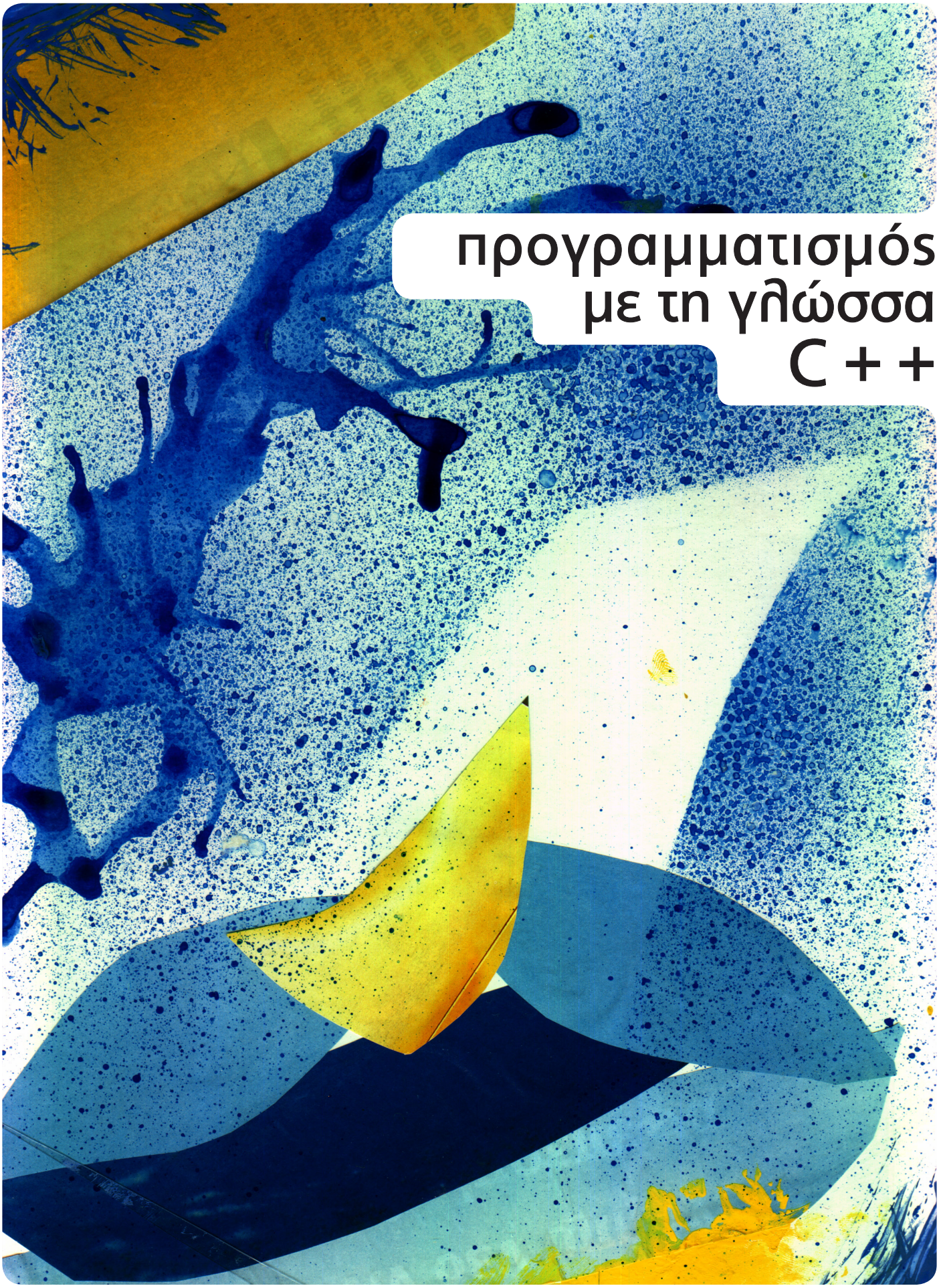
ΘΕΟΔΩΡΟΣ ΑΛΕΒΙΖΟΣ | Ομότιμος Καθηγητής Τ.Ε.Ι. Καβάλας



προγραμματισμός με τη γλώσσα C++

/ 2η ΕΚΔΟΣΗ | αθήνα 2014

Προγραμματισμός με τη Γλώσσα C++

Έκδοση Β

Θεόδωρος Αλεβίζος, Ph.D.
Ομότιμος Καθηγητής Τ.Ε.Ι. Καβάλας

ΥΠΟΥΡΓΕΙΟ ΕΘΝΙΚΗΣ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ ΕΠΕΑΕΚ
ΕΥΡΩΠΑΪΚΗ ΕΝΩΣΗ
ΣΥΓΧΡΗΜΑΤΟΔΟΤΗΣΗ
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
ΕΥΡΩΠΑΪΚΟ ΤΑΜΕΙΟ ΠΕΡΙΦΕΡΕΙΑΚΗΣ ΑΝΑΠΤΥΞΗΣ

Η ΠΑΙΔΕΙΑ ΣΤΗΝ ΚΟΡΥΦΗ
Επιχειρησιακό Πρόγραμμα
Εκπαίδευσης και Αρχικής
Επαγγελματικής Κατάρτισης

Αθήνα 2014

Εξώφυλλο: **Κωνσταντίνος Αλεβίζος**, με βάση ένα έργο της **Αφροδίτης-Ελένης Αλεβίζου.**

Πρόλογος

Η πρώτη έκδοση του παρόντος σχεδιάστηκε και γράφτηκε το ακαδημαϊκό έτος 1999-2000 για να καλύψει τις ανάγκες του νέου τότε Τμήματος Βιομηχανικής Πληροφορικής του ΤΕΙ Καβάλας. Το πρώτο μέρος εκδόθηκε από την (τότε) Μονάδα Εκδόσεων του ΤΕΙ Καβάλας ενώ το δεύτερο μέρος μοιράστηκε σε μορφή σημειώσεων.

Έκτοτε, τα μαθήματα προγραμματισμού διδάχθηκαν πολλές φορές από τον υπογραφόμενο και άλλους συναδέλφους, το μάθημα «Προγραμματισμός ΙΙ» χωρίστηκε σε «Τεχνικές Προγραμματισμού» και «Αντικειμενοστρεφή Προγραμματισμό», διάφορα προστέθηκαν (και κάποια αφαιρέθηκαν) και φτάσαμε στη παρούσα μορφή. Προφανώς δεν μπορούμε πια να μιλούμε για σχεδιασμένο και καλοζυγισμένο βιβλίο· πρόκειται για συμπίληση των εκπαιδευτικών υλικών που μοιράστηκαν (και μερικών που δεν μοιράστηκαν) όλα αυτά τα χρόνια από τον συγγραφέα.

Και ναι μεν φτάσαμε σε μια μορφή αρκετά διαφορετική από την αρχική σχεδίαση αλλά οι αλλαγές έγιναν για να καλυφθούν εκπαιδευτικές ανάγκες που εμφανίζονταν κατά τη διδασκαλία σε ομάδες φοιτητών που άλλαζαν κάθε εξάμηνο (και κατά την εργαστηριακή τους άσκηση). Έτσι, μπορούμε να πούμε ότι όλα αυτά που προστέθηκαν έκαναν υο παρόν πιο ενδιαφέρον και πιο χρήσιμο.

Για πολλούς από τους φοιτητές μου το πρόβλημα του παρόντος ήταν τα πολλά «μεγάλα» παραδείγματα! Λοιπόν ας βάλουμε τα πράγματα στη θέση τους:

- Αν –για κάθε νέο στοιχείο που εισάγεται– βάζουμε μόνον ένα προγραμματάκι 10 γραμμών τότε μιμούμαστε τα εγχειρίδια χρήσης των διαφόρων μεταγλωττιστών αλλά δεν διδάσκουμε προγραμματισμό.[1]
- Η διδασκαλία προγραμματισμού απαιτεί παραδείγματα επίλυσης προβλημάτων όπου χρησιμοποιούνται τα στοιχεία που παρουσιάζονται.
- Τέλος, μπορεί τα παραδείγματα να φαίνονται μεγάλα για ένα διδακτικό εγχειρίδιο αλλά είναι πολύ μικρά σε σχέση με τα «πραγματικά» προγράμματα.

Θα πρέπει ακόμη να δώσουμε μια απάντηση σε ένα άλλο ερώτημα: αφού από νωρίς παρουσιάζουμε τον τύπο *string* και τους τύπους ρευμάτων γιατί δεν παρουσιάζουμε και το *vector* και βασανιζόμαστε με τα **new** και τα **delete**;

- Η δυναμική μνήμη είναι ο μόνος πόρος που τον διαχειριζόμαστε με τον ίδιο τρόπο σε οποιοδήποτε Λειτουργικό Σύστημα. Έτσι, μπορούμε να δώσουμε παραδείγματα διαχείρισης πόρων.
- Γενικότερα, σε αρκετές περιπτώσεις, λέμε ποια είναι η απλή λύση αλλά υλοποιούμε κάποιαν άλλη για εκπαιδευτικούς λόγους.

Τμήματα του παρόντος χρησιμοποιήθηκαν στη διδασκαλία μαθημάτων Προγραμματισμού και από την κα Κ. Μήτσα, τον Δρ Λ. Μωυσιάδη και πολλούς έκτακτους εκπαιδευτικούς. Οι παρατηρήσεις τους ήταν πολύτιμος οδηγός στην παραγωγή εκπαιδευτικού υλικού και τους ευχαριστώ.

Το έργο του εξωφύλλου είναι μια ματιέρα της γραφίστριας κας Αφροδίτης-Ελένης Αλεβίζου. (Ο τίτλος της ήταν «Καλοκαίρι» αλλά όταν το είδα σκέφτηκα «Objects!...») Την ευχα-

[1] Αν σου αρέσει αυτός ο τρόπος παρουσίασης θα απολαύσεις τα:
`http://publib.boulder.ibm.com/infocenter/compbgpl/v9v111/index.jsp?topic=/com.ibm.xlcpp9.bg.doc/language_ref/compiler_pubs.htm`
`http://www.cplusplus.com/`

ριστώ όπως και τον αρχιτέκτονα = γραφιστή κ Κωνσταντίνο Αλεβίζο που δημιούργησε το εξώφυλλο.

Η παραγωγή και η ψηφιοποίηση του εκπαιδευτικού υλικού υποστηρίχθηκε κατά περιόδους από από το έργο «Ενίσχυση Σπουδών Πληροφορικής στο ΤΕΙ Καβάλας» κατηγορία πράξης 2.2.2.γ «Ενίσχυση των ΤΠΕ στη Τριτοβάθμια Εκπαίδευση» του ΕΠΕΑΕΚ ΙΙ.

Θ. Αλεβίζος
Αθήνα, Σεπτέμβριος 2014

Περιεχόμενα

Μέρος Α
Εισαγωγή στον Προγραμματισμό

κεφάλαιο

0

Επεξεργασία Στοιχείων – Προγραμματισμός

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα σιγουρέψουμε ότι εννοούμε το ίδιο πράγμα όταν αναφέρουμε μερικούς πολύ συνηθισμένους όρους της Πληροφορικής. Θα μάθουμε και μερικά πράγματα που θα μας απασχολούν συχνά στη συνέχεια.

Προσδοκώμενα αποτελέσματα:

- Θα καταλάβεις ότι όταν θέλουμε να γράψουμε ένα πρόγραμμα πρέπει πρώτα να καθορίσουμε με ακρίβεια τις προδιαγραφές του.
- Θα μάθεις τα συστατικά των προδιαγραφών.
- Θα μάθεις να διαβάζεις συντακτικούς ορισμούς όρων μιας γλώσσας προγραμματισμού.

Έννοιες κλειδιά:

- *αλγόριθμος*
- *πρόγραμμα*
- *προδιαγραφές προγράμματος*
- *προϋπόθεση (precondition)*
- *απαίτηση (postcondition)*
- *ορθότητα προγράμματος*
- *συμπερασματικοί κανόνες (inference rules)*
- *συντακτικά (syntax)*
- *νοηματικά (semantics)*
- *συμβολισμός BNF*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Επεξεργασία Στοιχείων:
Κατά την αλληλεπίδρασή μας με το περιβάλλον, ανάμεσα στα άλλα, δεχόμαστε και **πληροφορίες** (information) που πλουτίζουν τον γνωστικό μας κόσμο. Το μυαλό μας *επεξεργά-*

ζεται αυτές τις πληροφορίες, δηλαδή τις συσχετίζει μεταξύ τους και με γνώσεις που προϋπάρχουν, για να βγάλει συμπεράσματα που μας είναι χρήσιμα και να δημιουργήσει νέα γνώση. Η **επεξεργασία πληροφοριών** (information processing) είναι κάτι που γίνεται από τον άνθρωπο, ατομικώς ή συλλογικώς, με την βοήθεια μηχανών ή χωρίς αυτές, από καταβολής του ανθρώπινου γένους.

Όταν η επεξεργασία δεν γίνεται μόνο από τον άνθρωπο που έχει συλλέξει την πληροφορία, παρουσιάζεται η ανάγκη να *παραστήσει* την πληροφορία με κάποιο συμβολικό τρόπο για να τη μεταβιβάσει σε κάποιον άλλο ή στη μηχανή. Η παράσταση της πληροφορίας με έναν συμβολικό τρόπο μας δίνει τα **στοιχεία** ή **δεδομένα** (data). Έτσι, χρησιμοποιούμε και τον όρο **επεξεργασία στοιχείων** (data processing). Τα ακατέργαστα στοιχεία, που είναι το αντικείμενο της επεξεργασίας, λέγονται **στοιχεία εισόδου** (input data), ενώ τα αποτελέσματα της επεξεργασίας λέγονται **στοιχεία εξόδου** (output data).

Παράδειγμα 1

Μετρώντας, μερικές φορές, την τάση v στις άκρες ενός αγωγού και το ρεύμα i που τον διαρρέει, παίρνουμε μερικά (π.χ. 50) ζεύγη τιμών:

$$(v_k, i_k) \quad k = 0..49$$

Με κατάλληλη επεξεργασία (π.χ. μέθοδος ελαχίστων τετραγώνων), μπορούμε:

- να συναγάγουμε ότι το ρεύμα είναι ανάλογο της τάσης (νόμος του Ohm),
- να υπολογίσουμε την αντίσταση του αγωγού.

Τα ζεύγη τιμών (v_k, i_k) είναι τα στοιχεία εισόδου. Η τιμή της αντίστασης αλλά και ο νόμος του Ohm είναι τα στοιχεία εξόδου.

Παράδειγμα 2

Ένας εργάτης που επιβλέπει τη λειτουργία ενός ατμολέβητα, παρακολουθεί την πίεση μεσα στον λέβητα από ένα μανόμετρο. Όταν η πίεση μέσα στον λέβητα ξεπερνάει μια κρίσιμη τιμή, ανοίγει ειδικές βαλβίδες για να διαφύγει ατμός και να πέσει η πίεση.

Εδώ η ένδειξη του μανόμετρου είναι το στοιχείο εισόδου. Αλλά ποιό είναι το στοιχείο εξόδου; Μπορούμε να πούμε ότι είναι η ενέργεια του εργάτη. Τα πράγματα όμως γίνονται πιο καθαρά αν σκεφτούμε τη διαδικασία στο επίπεδο του νευρικού συστήματος και δούμε

- ως στοιχείο εισόδου το σήμα που στέλνει το μάτι πρός τον εγκέφαλο και
- ως στοιχείο εξόδου τη διαταγή που στέλνει ο εγκέφαλος προς το χέρι.

Όπως φαίνεται από το δεύτερο παράδειγμα, τα σύμβολα που χρησιμοποιούμε για την παράσταση των στοιχείων δεν χρειάζεται να είναι πάντοτε ορατά ή, γενικώς, αντιληπτά από μια αίσθηση.

Αυτό το παράδειγμα μας δείχνει πώς χρησιμοποιείται η επεξεργασία στοιχείων για **έλεγχο** (control) κάποιας διαδικασίας.

0.1 Αλγόριθμοι

Για την επίλυση των προβλημάτων μας και για την επεξεργασία στοιχείων, ειδικότερα, επινοούμε διάφορες μεθόδους. Μια τέτοια μέθοδος, που μπορεί να περιγραφεί *με πεπερασμένη ακολουθία καλά καθορισμένων βημάτων*, λέγεται **αλγόριθμος** (algorithm).

Ας πούμε, για παράδειγμα, ότι έχουμε δέκα ακέραιους αριθμούς, τους:

17 13 67 104 2 69 45 375 35 84

και θέλουμε να βρούμε ποιος είναι ο μέγιστος και τι σειρά έχει μέσα στη δεκάδα. Μια μεθοδος, που θα μπορούσες να ακολουθήσεις, περιγράφεται στο Σχ. 0-1.

Βήμα	Ενέργεια
1:	Ας πούμε ότι μέγιστος είναι ο 1ος (17)
2:	Ξεκίνα από το 2ο στοιχείο
3:	Αν τελείωσαν οι αριθμοί ΤΕΛΟΣ
	Όσο υπάρχουν και άλλοι κάνε τα εξής:
4:	Αν είναι μεγαλύτερος από αυτόν που είχες για μέγιστο τότε
5:	Θεώρησε ότι μέγιστος είναι αυτός ο αριθμός
6:	Προχώρησε στον επόμενο αριθμό
7:	Ξαναπήγαινε στο βήμα 3

Σχ. 0-1 Περιγραφή του αλγόριθμου «με λόγια».

Αυτή η περιγραφή δεν είναι και τόσο ικανοποιητική. Γιατί; Έχει πολλά λόγια! Βέβαια, είναι μεγάλη υπόθεση να περιγράφουμε έναν αλγόριθμο σε φυσική γλώσσα, αλλά, πολύ συχνά, όπως ήδη ξέρεις από την καθημερινή σου πείρα, οι ίδιες λέξεις και οι ίδιες εκφράσεις έχουν (έστω και ελαφρώς) διαφορετική σημασία για διαφορετικούς ανθρώπους. Μια λύση στο πρόβλημα αυτό είναι ο περιορισμός του λεξιλογίου και η χρήση καλά ορισμένων συμβόλων. Ας δοκιμάσουμε ξανά κάνοντας μερικές συμβάσεις για τον συμβολισμό:

- με το: **Βάλε** $X \leftarrow Y$
  εννοούμε: βάλε (αντίγραψε) την τιμή του Y ως τιμή του X,
- με το: **Όσο** <συνθήκη> **κάνε τα εξής:**
  εννοούμε: όσο ισχύει η συνθήκη να εκτελείς ξανά και ξανά όλα τα βήματα που ακολουθούν μέσα στο άγκιστρο (**{**) –με τη σειρά που γράφονται– και να ξαναελέγχεις τη συνθήκη.
- με το: **Αν** <συνθήκη> **τότε**
  εννοούμε: αν ισχύει η συνθήκη να εκτελέσεις μια φορά όλα τα βήματα που ακολουθούν μέσα στο άγκιστρο (**{**).

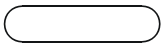
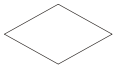
Ειδικώς για αυτήν την περίπτωση, ονομάζουμε a_k, $k = 0..9$ τους αριθμούς που έχουμε, m το μέγιστο που ψάχνουμε, k_m τη θέση του μέγιστου μέσα στη δεκάδα. Και να πώς γράφεται ο αλγόριθμός μας:

Βάλε $m \leftarrow a_0$, $k_m \leftarrow 0$, $k \leftarrow 1$
Όσο $k \le 9$ **κάνε τα εξής:**
{ **Αν** $a_k > m$ **τότε**
{ **Βάλε** $m \leftarrow a_k$, $k_m \leftarrow k$
Βάλε $k \leftarrow k + 1$

Τώρα ο αλγόριθμος έχει περιγραφεί κάπως πιο καθαρά, αλλά όχι τέλεια. Αν δεν έχεις πρόβλημα να καταλάβεις τον αλγόριθμο με τη δεύτερη ή την πρώτη διατύπωση, αυτό δεν οφείλεται στην τέλεια γλώσσα που χρησιμοποιούμε αλλά στην ανθρώπινη ευφυΐα.

Μια άλλη γλώσσα που χρησιμοποιείται ευρύτατα είναι αυτή των **λογικών διαγραμμάτων** ή **διαγραμμάτων ροής** (flowcharts). Είναι μια γλώσσα που επινοήθηκε για την περιγραφή διαδικασιών και αλγορίθμων. Για να σχεδιάσουμε ένα λογικό διάγραμμα, χρησιμοποιούμε ειδικά σύμβολα. Στον Πίν. 0-1 βλέπεις μερικά, ενώ στο Σχ. 0-2 βλέπεις πώς μπορούμε να ζωγραφίσουμε με λογικό διάγραμμα τον αλγόριθμο που δώσαμε παραπάνω.

Στόχος των παραπάνω δοκιμών ήταν να βρούμε μια γλώσσα που θα μπορεί να καταλάβει μια *μηχανή* πολύ λιγότερο ευφυής από τον άνθρωπο. Μια γλώσσα που θα μας επιτρέπει να περιγράψουμε τα βήματα ενός αλγορίθμου με *σαφήνεια* και *πληρότητα*. Οι *γλώσσες προγραμματισμού*, όπως η C++, που θα μάθουμε στο βιβλίο αυτό, είναι τέτοιες γλώσσες. Ο παραπάνω αλγόριθμος στη C++, γράφεται ως εξής:

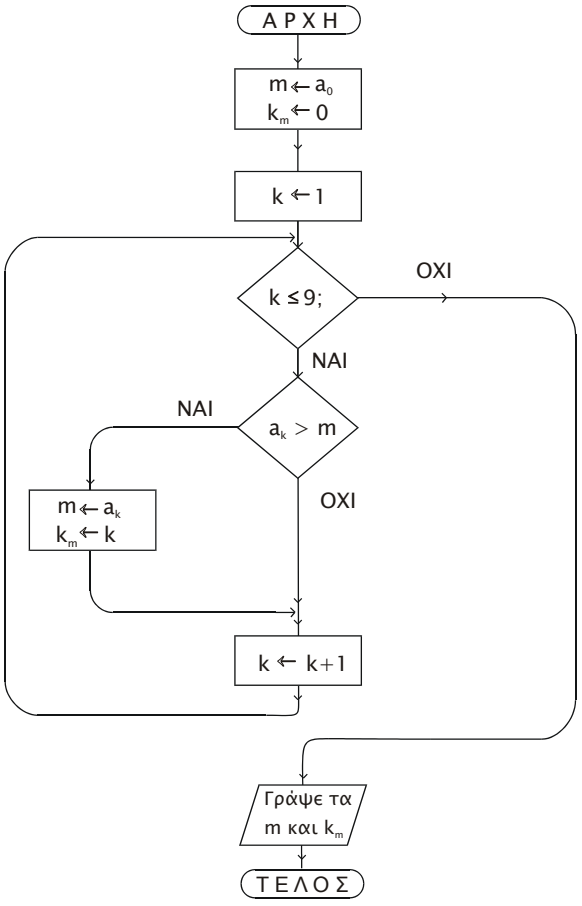
Σύμβολο	Χρήση
	για αρχή ή τέλος αλγορίθμου
	για διάβασμα (είσοδο) / γράψιμο (έξοδο) στοιχείων
	ρόμβος για αποφάσεις
	για περιγραφή επεξεργασίας
← ↑ → ↓	βέλη ροής

Πίν. 0-1: Σύμβολα για τα διαγράμματα ροής.

```
m = a[0];  km = 0;
for ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > m )  {  m = a[k];   km = k;  }
```

Αυτή είναι μια περιγραφή «που μπορεί να την καταλάβει ο υπολογιστής!» Το πώς την καταλαβαίνει θα το δούμε στις επόμενες παραγράφους.

Έστω λοιπόν ότι έχουμε βρει τη γλώσσα που θα περιγράφουμε τους αλγορίθμους μας και τη μάθαμε καλά· λύθηκε το πρόβλημά μας; Όχι βέβαια! Εκτός από τη δυνατότητα δια-



Σχ. 0-2 Το λογικό διάγραμμα του αλγόριθμου για την εύρεση του μέγιστου στοιχείου.

τύπωσης του αλγόριθμου μας ενδιαφέρει και ο ίδιος ο αλγόριθμος: θα πρέπει να είναι **ορθός** (correct) και **αποδοτικός** (efficient). Με τα θέματα αυτά θα ασχοληθούμε αργότερα.

0.2 Προγραμματα και Γλώσσες Προγραμματισμού

Χρειαζόμαστε λοιπόν γλώσσες που να μπορεί να «καταλάβει» ο ηλεκτρονικός υπολογιστής (ΗΥ)![1] Ένας αλγόριθμος διατυπωμένος σε γλώσσα που μπορεί να καταλάβει ο ΗΥ είναι ένα **πρόγραμμα** (program). Κάθε βήμα του προγράμματος λέγεται **εντολή** (statement, instruction). Οι γλώσσες που χρησιμοποιούμε για να γράφουμε τα προγράμματα λέγονται **γλώσσες προγραμματισμού** (programming languages).

Όπως κάθε γλώσσα, έτσι και οι γλώσσες προγραματισμού έχουν κανόνες για την χρήση τους:

- Οι **συντακτικοί** κανόνες (syntax) καθορίζουν το *πώς πρέπει να γράφεται* ένα πρόγραμμα.
- Οι **νοηματικοί** ή **σημαντικοί** (semantics) κανόνες καθορίζουν *το νόημα αυτών που γράφονται*.

Επειδή ο ΗΥ είναι πολύ απλοϊκός σε σύγκριση με το ανθρώπινο μυαλό, *οι γλώσσες προγραμματισμού έχουν πολύ αυστηρούς κανόνες* και πολύ μικρή (ή και καθόλου) ανοχή σε λάθη. Αν κάνεις κάποιο **συντακτικό** λάθος, όσο «μικρό» κι αν είναι κατά την γνώμη σου, ο ΗΥ δεν θα καταλάβει τι του ζητάς. Αν κάνεις κάποιο **σημαντικό** λάθος, λάθος στο τι του ζητάς να κάνει, ο ΗΥ δεν μπορεί να καταλάβει ότι «άλλο ήθελες να πεις» ή ότι «προφανώς πριν από αυτό έπρεπε να κάνει εκείνο». Τελικώς θα πάρεις λάθος αποτέλεσμα. Ο ΗΥ θα είναι ένας υπάκουος αλλά κουτός υπηρέτης σου: *θα εκτελεί* με πολύ καλή απόδοση *αυτό που τον διατάζεις, αλλά όχι αυτό που θα ήθελες να τον διατάξεις*.

Ο **ψηφιακός ΗΥ** (digital computer) παριστάνει τα πάντα με τα ψηφία του δυαδικού συστήματος: το "`0`" και το "`1`". Το πρόγραμμά μας, όπως και πολλά από τα στοιχεία που χρησιμοποιεί, αποθηκεύονται στην μνήμη του ΗΥ. Επομένως... Ναι, αυτό που κατάλαβες (ή φοβήθηκες): το πρόγραμμα είναι γραμμένο με 0 και 1. Μια τέτοια γλώσσα προγραμματισμού, με 0 και 1, λέγεται **γλώσσα μηχανής** (machine language). Ο προγραμματισμός σε μια τέτοια γλώσσα είναι δύσκολος.

Παράδειγμα

Στη γλώσσα μηχανής κάποιου υπολογιστή οι εντολές:

```
0100000000000111
0001000000001000     (A)
0101000000001001
```

θα μπορούσαν να σημαίνουν τα εξής:

1. πρόσθεσε τους ακέραιους που βρίσκονται στις θέσεις 7 και 8 της μνήμης,
2. αποθήκευσε το άθροισμα στην θέση 9.

Τα τέσσερα πρώτα ψηφία της κάθε μιας από τις παραπάνω εντολές δίνουν τη διαταγή (εντολή), το τι πρέπει να γίνει. Τα υπόλοιπα δείχνουν μια θέση της μνήμης –μια **διεύθυνση** (address). Αν ξέρεις το δυαδικό σύστημα, μπορείς να «δεις» τα 7, 8 και 9 για τα οποία μιλάμε παραπάνω.

Συνήθως, για να διευκολυνθεί η δουλειά του προγραμματιστή, η εισαγωγή του προγράμματος κλπ μας δίνεται η δυνατότητα να γράφουμε τις εντολές με συμβολικά ονόματα και τις διευθύνσεις στο δεκαεξαδικό ή το οκταδικό ή το δεκαδικό σύστημα. Μας δίνεται δηλαδή η δυνατότητα να γράψουμε τις παραπάνω εντολές με πιο βολικό τρόπο, π.χ.:

---

[1] ή απλώς **υπολογιστής** (computer).

```
LOAD    7
ADD     8
STORE   9
```

Παρ' όλο που οι μεγάλες δυσκολίες δεν αντιμετωπίστηκαν, αυτές οι εντολές έχουν κάποιο νόημα, τουλάχιστον για τους αγγλομαθείς. Παρακάτω θα δεις ότι έχουμε και άλλα εργαλεία που κάνουν τα πράγματα ακόμη καλύτερα.

Μια **συμβολική γλώσσα** (assembly language) έχει κατά βάση τις εντολές της γλώσσας μηχανής του ΗΥ, αλλά επιτρέπει χρήση συμβολικών διευθύνσεων. Είναι το «επόμενο βήμα» μετά τη γλώσσα μηχανής, για την ευκολία του προγραμματιστή. Οι συμβολικές γλώσσες προσφέρουν βέβαια και άλλες ευκολίες στον προγραμματιστή και χρησιμοποιούνται από πολλούς.

Τώρα όμως τα πράγματα αλλάζουν: Ο υπολογιστής μπορεί να εκτελέσει μόνο εντολές σε γλώσσα μηχανής· έτσι, δεν μπορεί να εκτελέσει αμέσως ένα πρόγραμμα που είναι γραμμένο σε συμβολική γλώσσα. Θα πρέπει προηγουμένως να **μεταφραστεί** σε γλώσσα μηχανής. *Αυτήν τη μετάφραση την κάνει κάποιο πρόγραμμα.* Το πρόγραμμα αυτό, που λέγεται **συμβολομεταφραστής** (assembler), υπάρχει έτοιμο στον ΗΥ. Ο συμβολομεταφραστής παίρνει, ως στοιχεία εισόδου, τις εντολές του προγράμματος σε συμβολική γλώσσα και παράγει, ως αποτέλεσμα, το ισοδύναμο πρόγραμμα σε γλώσσα μηχανής.[2]

Οι συμβολικές γλώσσες, παρ' όλο που είναι πιο βολικές από την γλώσσα μηχανής, δεν παύουν να είναι πιο κοντά στον υπολογιστή παρά στο χρήστη του, τον άνθρωπο. Στο παράδειγμα που δώσαμε παραπάνω, μας πηγαίνουν από

```
        LOAD  0007          LOAD  A
το      ADD   0008    στο   ADD   B
        STORE 0009          STORE X
```

Θα ήταν όμως πολύ καλύτερο αν μπορούσαμε, αντί για τα παραπάνω να γράφουμε:

```
X ← A + B
```

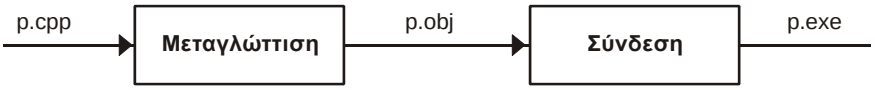
ή κάτι παρόμοιο.

Οι **γλώσσες προγραμματισμού υψηλού επιπέδου** (high level programming languages) μας δίνουν τη δυνατότητα να γράψουμε τους αλγορίθμους μας με τρόπο πιο «φιλικό», πιο οικείο στον άνθρωπο. Αυτές, είναι γλώσσες προγραμματισμού που μοιάζουν, λιγότερο ή περισσότερο, με τις ανθρώπινες γλώσσες (συνήθως τα αγγλικά) και έχουν σχέση με τα ανθρώπινα προβλήματα και τις μεθόδους που έχουμε για να τα λύνουμε. Οι περισσότερες έχουν σχεδιαστεί ώστε να ανταποκρίνονται καλύτερα σε προβλήματα ορισμένου τύπου. Για επιστημονικά προβλήματα έγιναν οι γλώσσες FORTRAN, ALGOL, APL και άλλες. Οι γλώσσες COBOL, RPG είναι σχεδιασμένες για εμπορικές εφαρμογές. Η PL/I, και η Ada μπορούν να δουλέψουν με προβλήματα πολλών τύπων. Η BASIC σχεδιάστηκε το 1965 για διδασκαλία προγραμματισμού και γνώρισε μεγάλη επιτυχία στους mini –και στους μικρο– υπολογιστές, αλλά χρησιμοποιείται απ' όλους για όλα! Η Pascal, που είναι και αυτή μια γλώσσα υψηλού επιπέδου, σχεδιάστηκε επίσης για διδασκαλία προγραμματισμού, αλλά χρησιμοποιείται στην παραγωγή.

Η C++ είναι μια γλώσσα που έχει γίνει πολύ δημοφιλής στους επιστήμονες της Πληροφορικής –αλλά και των άλλων θετικών επιστημών– και στους τεχνικούς· φαίνεται να αντικαθιστά τη FORTRAN στα επιστημονικά και τεχνικά προγράμματα. Το μεγαλύτερο μέρος του λογισμικού συστημάτων γράφεται σήμερα στη γλώσσα αυτή· το ίδιο ισχύει και για το λογισμικό εφαρμογών. Θα μας απασχολήσει σε επόμενη παράγραφο (και στα υπόλοιπα κεφάλαια του βιβλίου).

---

[2] Αν εξαιρέσουμε τις περιπτώσεις που έχουμε πολύ απλόν υπολογιστή, το πιο πιθανό είναι ότι το «ισοδύναμο πρόγραμμα σε γλώσσα μηχανής» θα παραχθεί κατά τη *φόρτωση* από τον **φορτωτή** (loader), αλλά ας τα αφήσουμε αυτά για μιαν άλλη φορά…



Σχ. 0-3 Από το αρχικό πρόγραμμα (p.cpp) στο τελικό ή εκτελέσιμο (p.exe).

Όπως το πρόγραμμα που γράφεται σε συμβολική γλώσσα έτσι και το πρόγραμμα που γράφεται σε γλώσσα υψηλού επιπέδου δεν μπορεί να εκτελεστεί από τον ΗΥ. Πρέπει πρώτα να γίνει η μετάφρασή του σε γλώσσα μηχανής. Η μετάφραση αυτή γίνεται από κάποιο άλλο πρόγραμμα που γενικώς λέγεται **μεταφραστής.**

Ένα είδος μεταφραστή είναι ο **μεταγλωττιστής** (compiler). Ο μεταγλωττιστής είναι ένα *πρόγραμμα* που παίρνει –ως στοιχεία εισόδου– το πρόγραμμά μας, γραμμένο σε γλώσσα υψηλού επιπέδου· από την επεξεργασία του βγάζει –ως αποτέλεσμα– το «ισοδύναμο» πρόγραμμα (πιθανότατα με κάποιες «βελτιώσεις») σε γλώσσα μηχανής. Αυτό που γράφει ο προγραμματιστής είναι το **αρχικό** ή **πηγαίο πρόγραμμα** (source program) και αυτό που παράγει ο μεταγλωττιστής είναι το **αντικειμενικό πρόγραμμα** (object program).

Συνήθως, το πρόγραμμα που βγάζει ο μεταγλωττιστής, δεν είναι έτοιμο για εκτέλεση. Ο μεταγλωττιστής βάζει μέσα στο πρόγραμμα «οδηγίες» για εκτέλεση μερικών προγραμμάτων, που υπάρχουν έτοιμα σε «βιβλιοθήκες» του ΗΥ. Αυτά τα προγράμματα θα πρέπει να «συνδεθούν» με το αντικειμενικό πρόγραμμα ώστε να προκύψει το τελικό **εκτελέσιμο** (executable) πρόγραμμα. Αυτή τη δουλειά την κάνει ένα άλλο πρόγραμμα που λέγεται **συνδέτης** (linker ή linkage editor). Αυτό φαίνεται διαγραμματικώς στο Σχ. 0-3.

Ένας άλλος τύπος μεταφραστή είναι ο **ερμηνευτής** (interpreter). Ο ερμηνευτής δεν παράγει αντικειμενικό πρόγραμμα, αλλά μεταφράζει μια προς μια τις εντολές και τις δίνει για εκτέλεση.

0.3 Το Σωστό Πρόγραμμα

Ας πούμε ότι γράφεις το πρόγραμμα που είδαμε στην §0.2

```
      . . .
m = a[0];  km = 0;
fot ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > m) then {  m = a[k]  km = k;  }
      . . .
```

Αλλά, κάνεις μερικά λαθάκια[3]: αντί για "**for**" έβαλες "**fot**", μετά τη συνθήκη της "**if**" έβαλες το "**then**" και μεταξύ των εντολών "`m = a[k]`" και "`km = k`" δεν έβαλες ένα "**;**". Αυτά είναι παραδείγματα *συντακτικών λαθών*, που, γενικώς, δεν δημιουργούν μεγάλο πρόβλημα: θα τα βρει ο μεταγλωττιστής, θα σου δείξει πού (περίπου) είναι και θα τα διορθώσεις.

Ας δούμε τώρα κάτι άλλο:

```
      . . .
m = a[0];  km = 0;
for ( k = 11; k <= 9; k = k+1 )
   if ( a[k] > m )  {  m = a[k];  km = k;  }
      . . .
```

Εδώ δεν υπάρχουν συντακτικά λάθη. Αλλά υπάρχει κάτι άλλο, χειρότερο: ζητάμε το *k* –αυξανόμενο ξανά και ξανά κατά 1– να πάρει τιμές από 11 μέχρι 9. Αυτό δεν γίνεται και το αποτέλεσμα είναι: η "`if (a[k] > m) { m = a[k]; km = k; }`" δεν θα εκτελεσθεί ούτε μια φορά και έτσι οι *m* και *km* θα μείνουν με τις αρχικές τιμές τους.

Στην περίπτωση αυτή τα πράγματα είναι άσχημα: Οι εντολές περνούν χωρίς πρόβλημα από τον μεταγλωττιστή. Η μόνη ένδειξη ότι κάτι δεν πάει καλά είναι τα παράλογα αποτελέσματα. Αυτό εδώ είναι ένα *νοηματικό* λάθος και είναι σαφώς σοβαρότερο από το προη-

[3] Το γιατί είναι λάθη θα το μάθεις αργότερα· προς το παρόν δέξου ότι αυτό που δώσαμε στην §0.2 ήταν σωστό και δες τις διαφορές που υπάρχουν.

γούμενο. Θα μπορούσε κανείς να πει: «Σιγά το λάθος! Από απροσεξία, έμεινε το πλήκτρο πατημένο και πέρασαν δύο άσοι αντί για ένας. Είναι προφανές ότι εννοείται 1.» Τίποτε δεν είναι προφανές· όπως είπαμε, ο υπολογιστής *θα υπακούσει με ακρίβεια σε αυτό που έγραψες και όχι σε αυτό που θα ήθελες* να γράψεις.

Δες και το παρακάτω:

```
   . . .
m = a[0];  km = 0;
for ( k = 11; k <= 9; k = k+1 )
  if ( a[k] > M )  {  m = a[m];  km = k;  }
   . . .
```

Εδώ υπάρχει ένα άλλο πολύ σοβαρό λάθος: "`m = a[m]`". Το "`a[m]`" δεν έχει νόημα, διότι βάζουμε στο *m* ως τιμή έναν από τους αριθμούς που εξετάζουμε, ενώ –όπως θα έχεις καταλάβει– μέσα στις αγκύλες πρέπει να υπάρχει η σειρά του αριθμού.

Αυτό το λάθος μπορεί να γίνει από κάποιον που δεν έχει καταλάβει τι ακριβώς παριστάνουν οι μεταβλητές ή τον τρόπο που δουλεύει ο αλγόριθμος. Θα μπορούσαμε να συνεχίσουμε δίνοντας και άλλα τέτοια παραδείγματα, αλλά καλύτερα να σταματήσουμε και να ρωτήσουμε: τι θα πει «*το πρόγραμμα έχει λάθος;*»

Θα πεις: «Ωραία ερώτηση! Καλά, κουβέντα θ' ανοίξουμε τώρα; Αν δεν μου δώσει αποτέλεσμα: «μέγιστο το 375 και στη θέση 7» τότε ο αλγόριθμος έχει λάθος!» Ναι, αλλά η φιλοδοξία μας είναι να κάνουμε έναν αλγόριθμο που να δουλεύει γενικώς και όχι μόνο για τη συγκεκριμένη δεκάδα. Η απάντηση στην ερώτησή μας είναι:

- ***Ένα πρόγραμμα χαρακτηρίζεται ορθό ή λαθεμένο σε σχέση με συγκεκριμένες προδιαγραφές.***

Ποιες είναι οι προδιαγραφές για το παράδειγμά μας; Ας τις δούμε:

1. «Έχουμε δέκα ακέραιους αριθμούς» Εδώ υπάρχει μια σημαντική πληροφορία: Το ότι έχουμε να εξετάσουμε ακέραιους αριθμούς σημαίνει ότι μπορούμε να τους συγκρίνουμε με τον "`>`", πράγμα πολύ ουσιώδες για τη μέθοδο που περιγράψαμε. Αν είχαμε δέκα υπολογιστές ή δέκα δίσκους μουσικής ή δέκα μιγαδικούς αριθμούς δεν θα μπορούσαμε να κάνουμε σύγκριση με τον "`>`" και δεν θα είχε νόημα «ο μέγιστος».
2. «Ποιος είναι ο μέγιστος (από αυτούς)» Αυτό σημαίνει ότι θέλουμε να βρούμε έναν αριθμό που να είναι μεγαλύτερος από (ή ίσος με) οποιονδήποτε από τους αριθμούς που μας δόθηκαν. Μια καλύτερη διατύπωση είναι: να μην είναι μικρότερος από οποιονδήποτε από τους αριθμούς που μας δόθηκαν. Α, έτσι. Αν πάρουμε τον 500 μας κάνει; Όχι! Θα πρέπει να είναι ένας από τους αριθμούς που μας δόθηκαν.
3. «Τι σειρά έχει (ο μέγιστος) μέσα στη δεκάδα» Δηλαδή η απάντηση στην ερώτηση αυτή θα πρέπει να είναι ένας ακέραιος αριθμός μεταξύ 0 και 9. Και ακόμη, αν *m* ο μέγιστος και k_m η σειρά του στη δεκάδα θα πρέπει να έχουμε $m = a[k_m]$.

Η 1. είναι μια ιδιότητα των στοιχείων που έχουμε να επεξεργαστούμε και μπορούμε να τη χρησιμοποιήσουμε στον αλγόριθμό μας· λέγεται **συνθήκη** ή **κατηγόρημα εισόδου** (input condition/predicate) ή **προϋπόθεση** (precondition). Οι 2. και 3. είναι ιδιότητες που χαρακτηρίζουν τα αποτελέσματα της επεξεργασίας· αποτελούν τη **συνθήκη** ή **κατηγόρημα εξόδου** (output condition ή predicate) ή **απαίτηση** (postcondition).

Παρόμοια πράγματα έχουμε στην τεχνολογία και στα μαθηματικά. Σύγκρινε με τα παρακάτω:

- «Να κατασκευασθεί θερμαντικό σώμα που να αποδίδει 2000 *kcal/h* όταν τροφοδοτηθεί με τάση 220 *V* (ενεργή τιμή).»
- «Να βρεθούν πραγματικοί x_1 και x_2 τέτοιοι ώστε $ax_i^2 + bx_i + c = 0$, όπου *a*, *b*, *c* πραγματικοί τέτοιοι ώστε: $a \neq 0$ και $b^2 - 4ac \geq 0$.»
- «Δίνεται το ευθύγραμμο τμήμα *AB* και μια ευθεία *ε* που δεν είναι κάθετη στο *AB*. Να βρεθεί σημείο *M* της *ε* που ισαπέχει από τα *A* και *B*.»

Στο πρώτο παράδειγμα προϋπόθεση είναι $V_{εν} = 220\ V$, ενώ η απαίτηση είναι: απόδοση 2000 *kcal/h*.

Στο δεύτερο η προϋπόθεση είναι: $a, b, c \in \mathbb{R}$ και $a \neq 0$ και $b^2 - 4ac \geq 0$. Απαίτηση: $x_1, x_2 \in \mathbb{R}$ και $ax_1^2 + bx_1 + c = 0$ και $ax_2^2 + bx_2 + c = 0$.

Στο τρίτο η προϋπόθεση είναι: το ευθύγραμμο τμήμα *AB* δεν είναι κάθετο στην ευθεία· και η απαίτηση: σημείο *M* της *ε* τέτοιο ώστε: *MA* = *MB*.

Ο ηλεκτρολόγος που θα αναλάβει να κατασκευάσει το θερμαντικό σώμα δεν θα κάνει οτιδήποτε στην τύχη. Από τις προδιαγραφές θα υπολογίσει την αντίσταση που χρειάζεται· υπολογίζοντας τη μέγιστη ένταση ρεύματος θα επιλέξει τα σωστά καλώδια και την κατάλληλη ασφάλεια κ.ο.κ. Ο ηλεκτρολόγος θα μπορεί –με οδηγό τους υπολογισμούς που έκανε με βάση τις προδιαγραφές– να αποδείξει ότι η συσκευή του συμμορφώνεται με αυτές (τις προδιαγραφές).

Οποιοσδήποτε (σχεδόν) προσπαθήσει να λύσει το δεύτερο πρόβλημα, δεν πρόκειται να αρχίσει να δοκιμάζει αριθμούς στην τύχη. Θα χρησιμοποιήσει τους γνωστούς τύπους, που ισχύουν με την προϋπόθεση των προδιαγραφών. Αν ζητήσουμε, θα μπορεί να αποδείξει ότι οι τύποι είναι σωστοί.

Στο τρίτο πρόβλημα, δεν πρόκειται φυσικά να δοκιμάζουμε σημεία της *ε* με την ελπίδα ότι θα βρούμε κάποιο που να ισαπέχει των *A* και *B*. Θα πάρουμε τη μεσοκάθετο στο *AB* και θα βρούμε το σημείο τομής της με την *ε*. Εύκολα μπορούμε να αποδείξουμε ότι αυτό είναι το σημείο που ψάχνουμε.

Διάλεξε όποια αντιστοιχία θέλεις, αλλά –όπως οποιαδήποτε λύση σε κάποιο κατασκευαστικό πρόβλημα–

- ***το πρόγραμμα είναι ένα προϊόν για το οποίο πρέπει να υπάρχουν προδιαγραφές,***
- ***το πρόγραμμα γράφεται έτσι ώστε να συμμορφώνεται με τις προδιαγραφές,***
- ***πρέπει να αποδεικνύεται η ορθότητα του προγράμματος (δηλαδή η συμμόρφωση με τις προδιαγραφές).***

Πώς θα γράφονται οι προδιαγραφές; Αφού οι προδιαγραφές θα χρησιμοποιηθούν για την απόδειξη ορθότητας του προγράμματος θα πρέπει να διατυπώνονται με μαθηματική αυστηρότητα. Και με μαθηματικό συμβολισμό; Συνήθως αυτά πάνε μαζί.

Πώς μπορούμε να γράψουμε προδιαγραφές για το παράδειγμα του μέγιστου; Βασιζόμαστε στο ξεκαθάρισμα που κάναμε:

Προϋπόθεση: $\mathrm{a}[k] \in \mathbb{Z}$, για κάθε $k \in [\,0\,..\,9\,]$.

Απαίτηση: Υπάρχουν $m \in \mathbb{Z}$ και $k_m \in [\,0\,..\,9\,]$ τέτοια ώστε:

$(m = \mathrm{a}[k_m])$ και $(m \geq \mathrm{a}[k]$ για κάθε $k \in [\,0\,..\,9\,])$.

Κάναμε την εξής σύμβαση: με το $[\,\nu_1\,..\,\nu_2\,]$ $(\nu_1, \nu_2 \in \mathbb{Z},\ \nu_1 \leq \nu_2)$ συμβολίζουμε το υποσύνολο των ακεραίων που έχουν τιμές $\geq \nu_1$ και $\leq \nu_2$. Παρομοίως ορίζονται και τα $(\,\nu_1\,..\,\nu_2\,]$, $[\,\nu_1\,..\,\nu_2\,)$ και $(\,\nu_1\,..\,\nu_2\,)$ σε αντιστοιχία με τα υποσύνολα (διαστήματα) $(\alpha_1, \alpha_2]$, $[\alpha_1, \alpha_2)$ και (α_1, α_2) της ευθείας των πραγματικών.

Το πρόβλημα με τις ρίζες της δευτεροβάθμιας εξίσωσης, όπως το δώσαμε παραπάνω, είναι ένα *θεώρημα* που πρέπει να αποδειχτεί. Παρομοίως, ένα θεώρημα που πρέπει να αποδειχτεί είναι και το πρόβλημα για την εύρεση του μεγίστου. *Το πρόγραμμα είναι μια κατασκευαστική απόδειξη ύπαρξης*. Γενικώς, το πρόβλημα του προγραμματισμού μπορεί να διατυπωθεί ως εξής:

- ***Δίνονται τα στοιχεία $x_1,\ldots x_m$, που ικανοποιούν τη συνθήκη $\phi(x_1,\ldots x_m)$. Απόδειξε ότι υπάρχουν $y_1,\ldots y_n$ που ικανοποιούν τη συνθήκη $\psi(x_1,\ldots x_m, y_1,\ldots y_n)$.***

Ή αλλιώς:

- ***Γράψε ένα πρόγραμμα που θα παίρνει τα στοιχεία $x_1,\ldots x_m$, που ικανοποιούν τη συνθήκη $\phi(x_1,\ldots x_m)$ και θα υπολογίζει τα $y_1,\ldots y_n$ που θα πρέπει να ικανοποιούν τη συνθήκη $\psi(x_1,\ldots x_m, y_1,\ldots y_n)$.***

Εδώ θα πρέπει να αναγνώρισες στη $\phi(x_1,... x_m)$ την προϋπόθεση και στην $\psi(x_1,... x_m, y_1,... y_n)$ την απαίτηση.

Στη σύγχρονη Τεχνολογία Λογισμικού θα βρεις τον όρο **προγραμματισμός με συμβόλαιο** (programming by contract) για το αυτονόητο: τη σύνθεση προγράμματος με βάση συγκεκριμένες προδιαγραφές (που είναι το «συμβόλαιο»).

Για να βρούμε το σημείο που ισαπέχει από τα άκρα ευθύγραμμου τμήματος, χρησιμοποιούμε τον κανόνα και το διαβήτη για να φέρουμε τη μεσοκάθετο. Παραλλήλως αποδεικνύουμε ότι το σημείο που θα βρούμε είναι αυτό που ζητάμε. Το ίδιο πρέπει να κάνουμε και στο πρόγραμμά μας: δίνουμε εντολές στον υπολογιστή και με την εκτέλεσή τους προχωρούμε στον καθορισμό των τιμών των m και k_m από επεξεργασία των τιμών των a[k]. Θα πρέπει όμως να αποδείξουμε ότι τελικώς: (m = a[k_m]) και ($m \geq$ a[k] για κάθε $k \in$ [0 .. 9]) και $k_m \in$ [0 .. 9].

Για την απόδειξη ορθότητας της γεωμετρικής κατασκευής χρησιμοποιούμε θεωρήματα, αξιώματα και συμπερασματικούς κανόνες που σε ορισμένες περιπτώσεις έχουν σχέση με τη χρήση των οργάνων, π.χ.: «με τον κανόνα γράφω τη μοναδική ευθεία που περνάει από δύο σημεία», «όλα τα σημεία της γραμμής που γράφει το ένα σκέλος του διαβήτη ισαπέχουν από σημείο που βρίσκεται το άλλο σκέλος», «αφού κάθε σημείο της μεσοκαθέτου ισαπέχει από τα άκρα και το σημείο που η μεσοκάθετος τέμνει την ε ισαπέχει από τα άκρα» κλπ.

Για την απόδειξη της ορθότητας του προγράμματος θα χρησιμοποιούμε κατ' αρχήν αξιώματα και συμπερασματικούς κανόνες που έχουν σχέση με τις εντολές που ζητούμε να εκτελεστούν. Με άλλα λόγια: Κάθε εντολή έχει συγκεκριμένο νόημα, κάνει συγκεκριμένα πράγματα που μπορούμε να τα περιγράψουμε αυστηρώς με *αξιώματα* και *συμπερασματικούς κανόνες*. Με βάση αυτά μπορούμε να αποδείξουμε ότι το πρόγραμμά μας είναι σωστό.

Αν αποδείξουμε ότι

- αν ένα πρόγραμμα E αρχίσει να εκτελείται με στοιχεία εισόδου $x_1,... x_m$ που ικανοποιούν την $\phi(x_1,... x_m)$ (προϋπόθεση) και ότι
- αν η εκτέλεσή του τελειώσει κανονικώς θα μας δώσει στοιχεία εξόδου $y_1,... y_n$ που να ικανοποιούν την $\psi(x_1,... x_m, y_1,.... y_n)$ (απαίτηση)

τότε λέμε ότι το πρόγραμμά μας είναι **μερικώς ορθό** (partially correct) και γράφουμε συμβολικώς[4]:

$$\phi(x_1,... x_m) \{ E \} \psi(x_1,... x_m, y_1,.... y_n)$$

Αν τώρα έχεις αποδείξει τη μερική ορθότητα του προγράμματός σου και αν ακόμη αποδείξεις ότι *η εκτέλεση του προγράμματος θα τερματισθεί*, έχεις αποδείξει ότι το πρόγραμμά σου είναι **ολικώς ορθό** (totally correct). Συμβολικώς γράφουμε:

$$\phi(x_1,... x_m) < E > \psi(x_1,... x_m, y_1,.... y_n)$$

Όπως θα δεις στη συνέχεια, το πρόβλημα του τερματισμού έχει σχέση με τον τερματισμό εκτέλεσης των επαναληπτικών εντολών: σε ένα πρόγραμμα C++ θα πρέπει να αποδειχτεί ότι όλες οι **while**, **for** και **dowhile** καθώς και οι *αναδρομικές κλήσεις* υποπρογραμμάτων, που υπάρχουν στο πρόγραμμα, *δεν θα εκτελούνται επ' άπειρον*.

Για να κάνεις τις αποδείξεις ορθότητας προγράμματος σου χρειάζεται ο Κατηγορηματικός Λογισμός 1ης Τάξης (First Order Predicate Calculus)[5]. Εδώ, σε ένα βιβλίο εισαγωγής στον προγραμματισμό, δεν θα σου ζητήσουμε τόσα πολλά. Θα περιοριστούμε στον Προτασιακό Λογισμό (Propositional Calculus) με τις έννοιες και τα σύμβολα του οποίου θα πρέπει να είσαι πιο εξοικειωμένος. Σε κάθε περίπτωση, διάβασε το Παρ. Α για να δεις και τον συμ-

---

[4] Αλλού θα βρεις το ίδιο πράγμα γραμμένο ως εξής:

$$\{ \phi(x_1,... x_m) \} E \{ \psi(x_1,... x_m, y_1,.... y_n) \}$$

Όπως θα δεις, ο συμβολισμός που χρησιμοποιούμε είναι βολικός για τη C++.

[5] ή λογικές ανώτερης τάξης.

βολισμό που θα χρησιμοποιούμε στη συνέχεια. Φυσικά, δεν μπορούμε να αποφύγουμε τα «*για κάθε ...*» και «*υπάρχει ... τέτοιο ώστε ...*», που υπάρχουν άλλωστε και στο παράδειγμα με το μέγιστο· βάλαμε λοιπόν και μερικά πράγματα για αυτά, αλλά ελπίζουμε ότι καταλαβαίνεις σωστά τα παραπάνω με το νόημα που έχουν στην ελληνική γλώσσα.

0.3.1 Συμπερασματικοί Κανόνες

Για αξιώματα και θεωρήματα έχεις δει αρκετά στα μαθηματικά. Εδώ ας κάνουμε μια παρένθεση για να πούμε δύο λόγια για τους **συμπερασματικούς κανόνες** (inference rules). Όπως λέει και το όνομά τους, είναι κανόνες που σου λένε πώς μπορείς να βγάζεις συμπεράσματα όταν κάνεις τις αποδείξεις σου. Να ένα παράδειγμα από τον Κατηγορηματικό Λογισμό:

- Αν για κάθε x ισχύει η $P(x)$

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η $P(a)$ για τυχόν a.

Αυτό γράφεται συμβολικώς ως εξής:

$$\frac{\forall x \bullet P(x)}{P(a)}$$

Εδώ θα παρατηρήσεις ότι α) αυτό είναι αυτονόητο και τετριμμένο β) το χρησιμοποιείς χωρίς να ξέρεις ότι υπάρχει. Και τα δύο είναι σωστά: α) όπως τα αξιώματα, έτσι και οι συμπερασματικοί κανόνες είναι αυτονόητοι και τετριμμένοι (ή τουλάχιστον έτσι μας φαίνεται), β) ήδη αναφέραμε τη χρήση αυτού του κανόνα στο γεωμετρικό παράδειγμα που δώσαμε παραπάνω.

Ας δούμε ακόμη έναν κανόνα από τον Προτασιακό Λογισμό:

- Αν ισχύει η πρόταση P και
- Αν ισχύει η πρόταση $P \Rightarrow Q$, δηλαδή η P συνεπάγεται την Q,

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η Q.

Συμβολικώς (modus ponens):

$$\frac{P, P \Rightarrow Q}{Q}$$

Τώρα θα δούμε δύο συμπερασματικούς κανόνες που είναι χρήσιμοι για την απόδειξη ορθότητας προγραμμάτων.

Ας πούμε ότι έχεις να λύσεις το εξής πρόβλημα (θα το δούμε παρακάτω)[6]: Γράψε πρόγραμμα E τέτοιο ώστε:

$$(a == a_0) \;\&\&\; (b == b_0)\ \{\ E\ \}\ (a == b_0) \;\&\&\; (b == a_0)$$

Γράφεις το πρόγραμμα, αλλά στην απόδειξή του παίρνεις:

$$(a == a_0) \;\&\&\; (b == b_0)\ \{\ E\ \}\ (a == b_0) \;\&\&\; (b == a_0) \;\&\&\; (S == a_0)$$

Είναι σωστό το πρόγραμμα E ή όχι;

Είναι σωστό, διότι αφού απόδειξες ότι ισχύει η $(a == b_0) \;\&\&\; (b == a_0) \;\&\&\; (S == a_0)$ θα ισχύει και η $(a == b_0) \;\&\&\; (b == a_0)$, αφού ισχύει η $(A \;\&\&\; B) \Rightarrow A$.

Αυτό μας λέει ο παρακάτω συμπερασματικός **κανόνας του επακόλουθου** (rule of consequence):

- Αν μετά την εκτέλεση του προγράμματος E με προϋπόθεση P ισχύει η Q και
- Αν από η Q συνεπάγεται την R

τότε

[6] "==" σημαίνει «είναι ίσο με» και "&&" σημαίνει «και». Δες το Παράρτ. Α.

- Αν το πρόγραμμα E εκτελεσθεί με προϋπόθεση P μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P\{E\}\,Q,\ Q \Rightarrow R}{P\,\{E\}\,R} \qquad (E1)$$

Όπως θα δεις στη συνέχεια, πιο χρήσιμος θα μας είναι ένας άλλος παρόμοιος συμπερασματικός κανόνας:

- Αν η P συνεπάγεται την Q και
- Αν μετά την εκτέλεση του προγράμματος E με προϋπόθεση Q ισχύει η R

τότε

- Αν το πρόγραμμα E εκτελεσθεί με προϋπόθεση P μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P \Rightarrow Q,\ \ Q\,\{E\}\,R}{P\,\{E\}\,R} \qquad (E2)$$

Στις αποδείξεις μας θα χρησιμοποιούμε τους δύο αυτούς κανόνες –κυρίως τον ($E2$)– χωρίς να τους αναφέρουμε πάντοτε.

0.4 Από τις Προδιαγραφές στο Πρόγραμμα

Αν ενδιαφέρεσαι σοβαρώς για την Πληροφορική και την Τεχνολογία Λογισμικού, θα πρέπει να δώσεις ιδιαίτερη προσοχή στο θέμα «προδιαγραφές». Πολύ γρήγορα θα καταλάβεις ότι αυτό που λέμε «προγραμματισμός» ή «ανάπτυξη εφαρμογής» είναι στην πραγματικότητα επεξεργασία προδιαγραφών.

Πώς δουλεύει ο Μηχανικός Λογισμικού (software engineer); Από την ανάλυση των απαιτήσεων της εφαρμογής που έχει να αναπτύξει, διατυπώνει προδιαγραφές για τα προγράμματα και τις βάσεις δεδομένων που πρέπει να γίνουν.

Ας πάρουμε τώρα ένα από αυτά τα προγράμματα. Αρχικώς προδιαγράφεται όπως είδαμε παραπάνω:

$$\phi(x_1,\ldots x_m) < E > \psi(x_1,\ldots x_m, y_1,\ldots y_n)$$

Στη συνέχεια το αρχικό πρόβλημα διασπάται σε δύο ή περισσότερα υποπροβλήματα E_1, ..., E_N:

$\phi(x_1,\ldots x_m)$
$< E_1 >$
$\omega_1(x_1,\ldots x_m, y_1,\ldots y_n, z_1,\ldots, z_p)$
:
$\omega_{N-1}(x_1,\ldots x_m, y_1,\ldots y_n, z_1,\ldots, z_p)$
$< E_N >$
$\psi(x_1,\ldots x_m, y_1,\ldots y_n)$

Τα $z_1,\ldots, z_p$ είναι βοηθητικά αντικείμενα (μεταβλητές) που χρειάζονται για την ανάπτυξη του προγράμματος.

Όπως βλέπεις κάθε υποπρόβλημα έχει προϋπόθεση την απαίτηση του προηγούμενου[7]. Αυτή η διαδικασία συνεχίζεται με τη διάσπαση των E_1, ..., E_N σε άλλα «μικρότερα» κ.ο.κ. μέχρι να φτάσουμε –κατ' αρχήν– σε προβλήματα που λύνονται με μια εντολή (στην πραγματικότητα σταματούμε όταν καταλήξουμε σε προβλήματα που η λύση τους είναι απλή). Τότε έρχεται η ώρα της **κωδικοποίησης** (coding) σε κάποια γλώσσα προγραμματισμού.

Πώς καταλαβαίνουμε ότι η διαδικασία της διάσπασης των προβλημάτων σε απλούστερα έφτασε σε προβλήματα που λύνονται με μια εντολή; Σύμφωνα με όσα λέμε, αυτό θα πρέπει να φαίνεται από τις προδιαγραφές των «υποπροβλημάτων»· άρα θα πρέπει να

[7] Η διάσπαση που βλέπεις εδώ είναι *σειριακή*· μπορεί όμως να γίνει και *παράλληλη*.

έχουμε προδιαγραφές για τις εντολές. Πράγματι, το νοηματικό μέρος της γλώσσας μας δίνει ακριβώς αυτές τις προδιαγραφές των εντολών και τις χρησιμοποιούμε για να γράφουμε τα προγράμματά μας και για να αποδεικνύουμε ότι είναι σωστά.

Με τον καιρό, όσο θα γράφεις μεγαλύτερα και δυσκολότερα προγράμματα, θα καταλάβεις ότι το σημαντικό κομμάτι του προγραμματισμού είναι η διατύπωση των προδιαγραφών. Οι νεόκοποι προγραμματιστές κάνουν το λάθος να μην ξεχωρίζουν τη διατύπωση προδιαγραφών από την κωδικοποίηση πράγμα που έχει πολύ κακές επιπτώσεις στα προγράμματα που γράφουν. Οι παλιοί προγραμματιστές είχαν μια συνταγή για να τονίζουν αυτήν την ανάγκη: «*think first, code later*» ή, στα ελληνικά: «*πρώτα σκέψου και άφησε για αργότερα το γράψιμο των εντολών*».

Η διαδικασία που περιγράψαμε παραπάνω, λέγεται **βήμα-προς-βήμα ανάλυση** (step-by-step refinement) και στηρίζεται στο συμπερασματικό **κανόνα της σύνθεσης** (rule of composition):

- Αν μετά την εκτέλεση του προγράμματος E_1 με προϋπόθεση P ισχύει η Q και
- Αν μετά την εκτέλεση του προγράμματος E_2 με προϋπόθεση Q ισχύει η R

τότε

- Αν με προϋπόθεση P εκτελεσθούν πρώτα το E_1 και μετά το E_2, μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P\{E_1\}Q,\ Q\{E_2\}R}{P\{E_1;E_2\}R} \qquad (\Sigma)$$

Τελειώνοντας (προς το παρόν) να πούμε ότι υπάρχουν ειδικές γλώσσες για τη διατύπωση προδιαγραφών, όπως είναι η Z (Diller 1990), (Spivey 1988), η VDM (Andrews & Ince 1991), (Jones 1990) κ.ά., που στηρίζονται στη Μαθηματική Λογική (Κατηγορηματικό Λογισμό). Η Z είναι μια γλώσσα που δημιουργήθηκε αποκλειστικώς για τη διατύπωση προδιαγραφών λογισμικού (αλλά και υλικού). Η VDM (Vienna Development Method) είναι μέθοδος ανάπτυξης λογισμικού που περιλαμβάνει και εργαλεία για την αυστηρή διατύπωση προδιαγραφών. Για ειδικές κατηγορίες προβλημάτων έχουν αναπτυχθεί ειδικές γλώσσες.

0.5 Αποδοτικότητα Προγράμματος

Δες πώς αλλιώς μπορούμε να γράψουμε τον αλγόριθμο του μεγίστου:

```
km = 0;
for ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > a[km] )  {  km = k;  }
m = a[km];
```

Εδώ ψάχνουμε τη θέση του μεγίστου· όταν τη βρούμε μπορούμε εύκολα να πάρουμε και το μέγιστο. Βλέπεις ότι αλγόριθμος είναι λίγο πιο «οικονομικός» από τον αρχικό (σύγκρινέ τους για να βρεις τις διαφορές.)

Γενικώς: ένα καλό πρόγραμμα δεν θα πρέπει να χρησιμοποιεί περισσότερη μνήμη ούτε να ζητάει την εκτέλεση περισσότερων εντολών από όσο χρειάζεται. Πρέπει δηλαδή να μην κάνει σπατάλη στη χρήση μνήμης και υπολογιστικού χρόνου.

Πολλοί προγραμματιστές, θέλοντας να συμμορφωθούν με αυτήν την αρχή, καταφεύγουν σε διάφορα *τεχνάσματα* με αποτέλεσμα: ένα *πρόγραμμα που δεν φαίνεται εύκολα πώς δουλεύει* και επομένως:

- δεν μπορεί να αποδειχτεί ότι είναι σωστό (και συχνά δεν είναι σωστό),
- δεν μπορεί να τροποποιηθεί.

Το πρόβλημα της βελτίστοποίησης του προγράμματος δεν λύνεται με τεχνάσματα αλλά *με έναν καλύτερο αλγόριθμο* που:

- αποδεικνύεται η ορθότητά του και
- δεν είναι σπάταλος σε υπολογιστικό χρόνο και χρήση μνήμης.

Βέβαια, όταν έχεις τον αλγόριθμο, υπάρχουν πολλοί τρόποι για να τον γράψεις σε κάποια γλώσσα προγραμματισμού. Οι τρόποι αυτοί διαφέρουν μεταξύ τους στις λεπτομέρειες. Ο καλός προγραμματιστής θα προσέξει τις λεπτομέρειες ώστε να αποφύγει διάφορες σπατάλες· αυτό λέγεται **μικροβελτιστοποίηση** (microoptimization). Η μικροβελτιστοποίηση είναι επιθυμητή *αλλά δεν έχει πρώτη προτεραιότητα*. Η πρώτη προτεραιότητα είναι το σωστό πρόγραμμα, δηλαδή το πρόγραμμα που αποδειγμένα ανταποκρίνεται στις προδιαγραφές του!

Σε μια εισαγωγή στον προγραμματισμό δεν ασχολείται κανείς και πολύ με τέτοια θέματα. Πάντως στη συνέχεια θα δεις μερικά απλά παραδείγματα που θα σου δείχνουν τι θα πει «καλύτερος αλγόριθμος» και τι θα πει «καλύτερη υλοποίηση» στις λεπτομέρειες.

0.6 Η Γλώσσα C++

Η γλώσσα C++ σχεδιάστηκε από τον B. Stroustrup με στόχο να δοθεί η δυνατότητα **αντικειμενοστρεφούς προγραμματισμού** (object-oriented programming) στους χρήστες της C. Είναι, κατ' αρχήν, υπερσύνολο της C (Kernighan & Ritsie 1978).

Η περιγραφή της γλώσσας δίνεται στο (Stroustrup 1997). Για την τυποποίησή της εργάσθηκαν σε συνεργασία οι επιτροπές ANSI X3J16 (του οργανισμού τυποποίησης των ΗΠΑ) και ISO WG21 (του διεθνούς οργανισμού τυποποίησης). Το πρότυπο (ISO/IEC 14882:1998) δόθηκε στη δημοσιότητα το 1998· θα δεις να αναφέρεται ως C++98. Επειδή το 2003 εγκρίθηκε μια τροποποίηση του προτύπου (ISO/ IEC 14882:2003) θα το δεις και ως C++03. Τον Σεπτέμβριο 2011 εγκρίθηκε το πρότυπο ISO/IEC 14882: 2011 και έχει το ανεπίσημο όνομα C++11.[8]

Η C++ έχει όλη την αποδοτικότητα της C όταν διαχειρίζεται δομές του υλικού (δυαδικό ψηφίο, ψηφιολέξη κλπ) ενώ από την άλλη μεριά δίνει στον προγραμματιστή τη δυνατότητα να υλοποιήσει **κλάσεις**. Ακόμη παρέχει και άλλες δυνατότητες που ευκολύνουν τη δουλειά του.

Στο C++11 υπάρχει πλήρες οπλοστάτιο για *πολυνηματικό σύνδρομο* προγραμματισμό ώστε να μπορούμε να γράψουμε προγράμματα που να εκμεταλλεύονται τις δυνατότητες που δίνουν τα σύγχρονα ΛΣ και οι σύγχρονοι επεξεργαστές.

Ήδη υπάρχουν αρκετοί μεταγλωττιστές που συμμορφώνονται σε μεγάλο ποσοστό με το πρότυπο C++03. Δύο από αυτούς δίνονται δωρεάν:

- Η Borland προσφέρει δωρεάν[9] τον μεταγλωττιστή (v.5.5) που χρησιμοποιεί στον C++ Builder. Συμμορφώνεται σε μεγάλο βαθμό με το πρότυπο της C++. Το πρόβλημα είναι ότι δεν έχει περιβάλλον ανάπτυξης και θα πρέπει να γράφεις τα προγράμματά σου στο Notepad και να τα περνάς με command line.
- Ο **g++** της GNU: Μπορείς να τον βρεις ενσωματωμένο σε περιβάλλον ανάπτυξης εφαρμογών (IDE) επίσης δωρεάν: α) Dev C++[10], β) Code::Blocks Studio[11]. Οι πειραματικές εκδόσεις του g++ που συμμορφώνονται με το C++11 υπάρχουν ήδη στο διαδίκτυο.[12]

---

[8] Το C++11 με ορισμένες τροποποιήσεις-συμπληρώσεις αναφέρεται ήδη ως C++14.

[9] `http://dn.codegear.com/article/20633`

[10] `http://www.bloodshed.net/`

[11] `http://www.codeblocks.org/`

[12] `http://gcc.gnu.org/projects/cxx0x.html`

0.7 Ο Συμβολισμός BNF

Για να έχουμε περισσότερη καθαρότητα και ακρίβεια στην περιγραφή του συντακτικού της γλώσσας, σε ορισμένες περιπτώσεις, θα χρησιμοποιούμε μια μορφή του γνωστού **συμβολισμού BNF** (Backus - Naur Form).

Ας ξεκινήσουμε με ένα

Παράδειγμα 1

Οι ακόλουθοι κανόνες δημιουργούν (ορίζουν) μια οντότητα με το όνομα *ψηφίο*, που μπορεί να είναι οποιοδήποτε από τα σύμβολα 0 ... 9· στη συνέχεια χρησιμοποιούμε το *ψηφίο* για να ορίσουμε την οντότητα *δεκαεξαδικό ψηφίο*.

ψηφίο = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
δεκαεξαδικό ψηφίο = *ψηφίο* |
"a" | "b" | "c" | "d" | "e" | "f" |
"A" | "B" | "C" | "D" | "E" | "F";

Όπως βλέπεις, ένας ορισμός οντότητας γράφεται ως εξής:

- γράφουμε το όνομα της οντότητας, π.χ. *ψηφίο*,
- μετά παραθέτουμε το "=" και
- τέλος γράφουμε τον κανόνα που ορίζει τη συντακτική δομή και τερματίζεται με ένα ";".

Ο κανόνας γράφεται ως εξής:

- όποτε εμφανίζεται ένα σύμβολο από το αλφάβητο της γλώσσας –**τελικό σύμβολο** (terminal symbol)– περικλείεται σε εισαγωγικά "...", π.χ. "2" ή "3",
- όταν κάποια οντότητα μπορεί να δημιουργηθεί με διάφορους τρόπους, οι διαφορετικές επιλογές διαχωρίζονται με το σύμβολο "|",
- για να καθορίσουμε ότι κάποια αντικείμενα εμφανίζονται με κάποια συγκεκριμένη σειρά τα παραθέτουμε χωρισμένα με κόμματα (",").

Παράδειγμα 2

Ο ακόλουθος κανόνας περιγράφει οποιονδήποτε από τους ορμαθούς "00", "01", ... "99", με βάση το ψηφίο που ορίσαμε παραπάνω.

διψήφιος αριθμός = *ψηφίο*, *ψηφίο*;

Για να μην ταλαιπωρούμαστε με πολύ γράψιμο χωρίς νόημα θα χρησιμοποιούμε και την παρακάτω σύμβαση συντόμευσης:

- αν δεν υπάρχει κίνδυνος παρεξήγησης τότε αν έχουμε μια ακολουθία (τελικών) συμβόλων που διαχωρίζονται με το σύμβολο "|" θα γράφουμε: το πρώτο, αποσιωποιητικά και το τελευταίο.

Έτσι, οι κανόνες του παραδ. 1 γράφονται:

ψηφίο = "0" | ... | "9";
δεκαεξαδικό ψηφίο = *ψηφίο* | "a" | ... | "f" | "A" | ... | "F";

Στους κανόνες μπορεί να υπάρχει και **αναδρομή** (recursion), δηλαδή στον ορισμό να χρησιμοποιούμε το οριζόμενο. Δες το:

Παράδειγμα 3

Να, τώρα, οι συντακτικοί κανόνες για να γράφουμε δεκαδικούς αριθμούς χωρίς πρόσημο:

δεκαδικός χωρίς πρόσημο = *ακέραιος χωρίς πρόσημο* |
κλασματικό μέρος |
ακέραιος χωρίς πρόσημο, *κλασματικό μέρος*;
κλασματικό μέρος = ".", *ακέραιος χωρίς πρόσημο*;

ακέραιος χωρίς πρόσημο = *ψηφίο* | *ακέραιος χωρίς πρόσημο*, *ψηφίο*;
ψηφίο = "**0**" | ... | "**9**";

Όπως βλέπεις, λέμε ότι ένας *ακέραιος χωρίς πρόσημο* μπορεί να είναι *ψηφίο* ή *ακέραιος χωρίς πρόσημο* ακολουθούμενος από *ψηφίο*. Δηλαδή: το "**133**" είναι *ακέραιος χωρίς πρόσημο*; Για να δούμε:

- Κατ' αρχάς δεν είναι *ψηφίο*. Θα πρέπει λοιπόν να είναι της μορφής *ακέραιος χωρίς πρόσημο*, *ψηφίο*. Πράγματι, το "**3**" είναι *ψηφίο*. Αν το υπόλοιπο ("**13**") είναι *ακέραιος χωρίς πρόσημο* είμαστε εντάξει.
- Και πάλι, το "**13**" δεν είναι *ψηφίο*. Θα πρέπει να είναι της μορφής *ακέραιος χωρίς πρόσημο*, *ψηφίο*. Το "**3**" είναι *ψηφίο*. Αν το υπόλοιπο ("**1**") είναι *ακέραιος χωρίς πρόσημο* είμαστε εντάξει.
- Το "**1**" όμως είναι *ψηφίο* άρα είναι *ακέραιος χωρίς πρόσημο*. Άρα και το "**13**" είναι *ακέραιος χωρίς πρόσημο*, επομένως και το "**133**" είναι *ακέραιος χωρίς πρόσημο*.

Αυτό είναι ένα παράδειγμα *αναδρομής στα αριστερά*. Αλλού μπορεί να δεις τον παραπάνω αναδρομικό ορισμό ως εξής:

ακέραιος χωρίς πρόσημο = *ψηφίο* | *ψηφίο*, *ακέραιος χωρίς πρόσημο*;

Αυτή είναι *αναδρομή στα δεξιά*. Εδώ θα χρησιμοποιούμε συνήθως αναδρομή στα αριστερά.

Τέλος, μέσα σε αγκύλες βάζουμε κάτι που μπορεί να υπάρχει μπορεί και όχι.

Παράδειγμα 4

Αντί να γράψουμε:

δεκαδικός = *πρόσημο*, *δεκαδικός χωρίς πρόσημο* |
δεκαδικός χωρίς πρόσημο;
πρόσημο = "**+**" | "**-**";

γράφουμε:

δεκαδικός = [*πρόσημο*] *δεκαδικός χωρίς πρόσημο*;
πρόσημο = "**+**" | "**-**";

Συχνά θα βλέπεις συντακτικούς ορισμούς οντοτήτων που δεν έχουν συγκεκριμένο φυσικό νόημα. Θα πρόκειται για ενδιάμεσους ορισμούς που χρησιμεύουν μόνον για να ολοκληρωθεί μια ακολουθία συντακτικών ορισμών. Μην τρομάζεις λοιπόν.

κεφάλαιο

1

Υπολογισμοί με Σταθερές

Ο στόχος μας σε αυτό το κεφάλαιο:

Να γράψουμε πρόγραμμα που θα δουλεύει! Μόνο με σταθερές; Ναι! Μόνο με σταθερές, αλλά θα έχει μερικά βασικά χαρακτηριστικά που έχουν όλα τα προγράμματα.

Προσδοκώμενα αποτελέσματα:

Τελικώς θα μπορείς να γράψεις προγραμματάκια που θα κάνουν υπολογισμούς και θα βγάζουν τα αποτελέσματα μαζί με περιγραφικά κείμενα. Θα μάθεις όμως ότι για να γράψεις ένα πρόγραμμα C++ θα πρέπει να «στήσεις ένα περιβάλλον» μέσα στο οποίο θα βάλεις τις εντολές για τους υπολογισμούς.

Έννοιες κλειδιά:

- *εντολή εξόδου*
- *cout*
- *endl*
- *ορμαθοί χαρακτήρων*
- *αριθμητικές σταθερές*
- *συναρτήσεις*
- *οδηγία "`include`"*
- *σχόλια*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Το Σύνολο Χαρακτήρων:

Κάθε γλώσσα έχει το αλφάβητό της, δίνοντας έτσι, σε αυτούς που τη χρησιμοποιούν, τη δυνατότητα να τη γράφουν. Ακόμη, στον γραπτό λόγο χρησιμοποιούμε και άλλα σύμβολα όπως αριθμούς, σημεία στίξης κλπ.

Τα παραπάνω ισχύουν και για τις γλώσσες προγραμματισμού. Για να δώσουμε στον ΗΥ ένα πρόγραμμά μας, πρέπει να το κωδικοποιήσουμε, χρησιμοποιώντας το αλφάβητο της γλώσσας και με βάση τους συντακτικούς κανόνες της. Το **σύνολο χαρακτήρων** (character set) της γλώσσας είναι μια διεύρυνση της έννοιας του αλφάβητου και καθορίζει με ακρίβεια το ποιούς χαρακτήρες μπορούμε να χρησιμοποιούμε όταν γράφουμε τα προγράμματά μας. Από την άλλη μεριά, αυτούς τους χαρακτήρες (τουλάχιστον) πρέπει να αναγνωρίζει σωστά ο ΗΥ που θα δώσουμε το πρόγραμμά μας.

Ας δούμε πρώτα μερικούς χαρακτήρες που χρησιμοποιούν όλες οι γλώσσες προγραμματισμού.

Κατ' αρχάς τα **γράμματα** (letters) του *Λατινικού* αλφάβητου, κεφαλαία:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

και πεζά:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Από εδώ και πέρα με το *γράμμα* θα εννοούμε *κεφαλαίο ή μικρό γράμμα του Λατινικού αλφαβήτου*. Όταν θέλουμε να αναφερθούμε σε γράμμα του Ελληνικού αλφαβήτου θα το τονίζουμε.

Πριν αφήσουμε τα γράμματα θα τονίσουμε ότι

- ***Η C++ ξεχωρίζει τα μικρά γράμματα από τα κεφαλαία!***[1]

παρ' όλο που ακόμη δεν μπορείς να καταλάβεις τι ακριβώς σημαίνει αυτό.

Ακόμη, οι γλώσσες χρησιμοποιούν και τα **ψηφία** (digits) του δεκαδικού συστήματος:

```
0 1 2 3 4 5 6 7 8 9
```

Πολύ συχνά θα συναντήσεις τον όρο **αλφαριθμητικός χαρακτήρας** (alphanumeric character). Με αυτόν εννοούμε κάποιον χαρακτήρα που είναι είτε γράμμα είτε ψηφίο.

Ένας άλλος χαρακτήρας που χρειάζονται οι γλώσσες προγραμματισμού είναι το **διάστημα** (space) ή **κενό** (blank). Φυσικά, το κενό είναι ένα μέρος του κειμένου όπου δεν υπάρχει κανένα γραφικό σύμβολο. Παρ' όλα αυτά μερικές φορές θα θέλουμε να τονίσουμε την ύπαρξή του. Τότε θα χρησιμοποιούμε το σύμβολο "␣". Αλλού θα δεις για την ίδια χρήση το σύμβολο "**ƀ**".

Παράδειγμα

```
TPITH␣1␣IAN␣2013
TPITHƀ1ƀIANƀ2013
```

Εκτός από τα γράμματα, τα ψηφία και το κενό, οι γλώσσες προγραμματισμού χρησιμοποιούν ορισμένους **ειδικούς χαρακτήρες**, π.χ.:

```
( ) . , + - * / =
```

Κάθε ΗΥ διαθέτει το δικό του σύνολο χαρακτήρων και συνήθως αυτό το σύνολο είναι διαθέσιμο σε αυτόν που γράφει ένα πρόγραμμα σε οποιαδήποτε γλώσσα. Έτσι λοιπόν, μπορεί να έχουμε στην διάθεσή μας περισσότερους χαρακτήρες από αυτούς που είδαμε. Αντίθετα, σε μερικές περιπτώσεις μπορεί να έχουμε περιορισμούς. Για τον λόγο αυτόν, το πρότυπο της κάθε γλώσσας προτείνει πάγιες αντικαταστάσεις αν κάποιοι χαρακτήρες δεν είναι διαθέσιμοι.

---

[1] Οι περισσότερες γλώσσες προγραμματισμού δεν τα ξεχωρίζουν.

1.1 Το Αλφάβητο (Σύνολο Χαρακτήρων) της C++

Αφού είδαμε τους χαρακτήρες που χρησιμοποούν όλες οι γλώσσες προγραμματισμού, ας δούμε τις ιδιαιτερότητες της C++. Όπως καταλαβαίνεις η διαφορά βρίσκεται στους **είδικούς χαρακτήρες**. Η C++ χρησιμοποιεί –περα από αυτούς που χρησιμοποιούν όλες οι γλώσσες– και τους:

```
< > % : ; ? ^ & | ~ ! \ " ' _ { } [ ] #
```

Στη C++, κάθε ειδικός χαρακτήρας παίζει κάποιο ιδιαίτερο ρόλο.

Για υπολογιστές που έχουν «φτωχό» σύνολο χαρακτήρων το πρότυπο της γλώσσας προτείνει πάγιες αντικαταστάσεις για τους χαρακτήρες δεν είναι διαθέσιμοι· αντικαθίστανται από **διγραφικές** (digraph) ή **τριγραφικές** (trigraph) ακολουθίες ή από λέξεις. Για παράδειγμα:

ο `{` αντικαθίσταται από τους `<%` και ο `}` από τους `%>`
ο `[` αντικαθίσταται από τους `<:` και ο `]` από τους `:>`
ο `#` αντικαθίσταται από τους `%:`

Στο Παράρτημα D υπάρχει πίνακας αυτών των αντικαταστάσεων.

1.2 Το Πρώτο Πρόγραμμα

Το πρώτο πρόγραμμα που θα δούμε είναι πολύ απλό και δεν κάνει πολλά πράγματα.

```
#include <iostream>

int main()
{
   std::cout << " Να το πρώτο μου πρόγραμμα" << std::endl;
   std::cout << " ΔΟΥΛΕΥΕΙ!" << std::endl;
   std::cout << " Είμαι στο τρένο της Τεχνολογίας"<< std::endl;
   std::cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << std::endl;
}
```

Γράψε *προσεκτικά* το πρόγραμμα αυτό με τον κειμενογράφο σου, μεταγλώττισέ το και ζήτα από τον ΗΥ σου να το εκτελέσει. Αποτέλεσμα που θα δεις στην οθόνη σου:

```
Να το πρώτο μου πρόγραμμα
ΔΟΥΛΕΥΕΙ!
Είμαι στο τρένο της Τεχνολογίας
ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
```

Σύγκρινε το αποτέλεσμα με το πρόγραμμα και μάντεψε! Με την εντολή "`std::cout << "`*κείμενο*`"`" ζητάς να γραφεί το κείμενο στην οθόνη του υπολογιστή σου. Με το "`<< std::endl`", στη συνέχεια, ζητάς να τελειώσει η γραμμή και ό,τι γραφεί στη συνέχεια να γραφεί στην επόμενη γραμμή. Τα άλλα τι είναι; Στην επόμενη παράγραφο τα εξηγούμε.

1.2.1 Να «Διώξουμε» το "`std::`"!

Πριν κάνουμε οτιδήποτε άλλο ας δούμε τι είναι αυτό το "`std::`" –που εμφανίζεται κάθε τόσο– και πώς μπορούμε να το διώξουμε.

Από το επόμενο κεφάλαιο θα καταλάβεις ότι τα πρόγράμματά σου θα έχουν πολλά ονόματα, όπως είναι τα *cout* και *endl*. Τα περισσότερα από αυτά θα είναι δικής σου επιλογής. Αργότερα θα δεις ότι για πολλά πράγματα που θέλεις να κάνεις υπάρχουν ήδη λύσεις σε βιβλιοθήκες προγραμμάτων, που φυσικά θα θέλεις να χρησιμοποιήσεις στα προγράμματα που θα γράφεις. Δεν αποκλείεται καθόλου ορισμένα ονόματα να χρησιμοποιούνται στο πρόγραμμά σου και σε κάποια βιβλιοθήκη για να παραστήσουν διαφορετικά αντικείμενα.

Η C++ για να διεκολύνει την κατάσταση δίνει τον εξής μηχανισμό: Μπορείς να δηλώσεις έναν **ονοματοχώρο** (namespace) μέσα στον οποίο δηλώνεις διάφορα ονόματα για

κάποιες χρήσεις. Αν λοιπόν έχεις το όνομα "*nm*" δηλωμένο σε δύο διαφορετικούς ονοματοχώρους *A*, *B* για δύο διαφορετικά προγραμματιστικά αντικείμενα μπορείς να τα ξεχωρίζεις γράφοντας `A::nm` και `B::nm`.

Το προγραμματιστικό περιβάλλον της C++ έχει έναν ονοματοχώρο για τα ονόματα των δικών του αντικειμένων, τον *std* (από το *st*andard). Έτσι, γράφοντας `std::cout` εννοούμε: το «το *cout* που έχει δηλωθεί στον ονοματοχώρο *std*.»

Πρόσεξε τώρα έναν άλλον τρόπο γραφής του προγράμματος:

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Αυτό το πρόγραμμα δουλεύει μια χαρά! Οι δύο δηλώσεις "`using`" λένε: «χρησιμοποιώ το *std*`::`*cout* ως *cout* και το *std*`::`*endl* ως *endl*».

Υπάρχει και άλλος τρόπος, πιο «τεμπέλικος»:

```
#include <iostream>

using namespace std;

int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Το "`using namespace std`" σημαίνει ότι χρησιμοποιώ ονόματα από τον *std*. Άρα όλα τα «άγνωστα» ονόματα είναι από εκεί.

Εδώ θα χρησιμοποιούμε τον τελευταίο τρόπο.

Για ονοματοχώρους θα μιλήσουμε εκτενώς αργότερα.

1.3 Πρόγραμμα

Όπως είδες και πιο πάνω, ένα απλό πρόγραμμα C++ είναι μια ακολουθία από εντολές που εκτελούνται η μια μετά την άλλη. Ας δούμε τους συντακτικούς κανόνες της C++ για τη σύνταξη ενός προγράμματος.

Ένα απλό πρόγραμμα αποτελείται από μια *συνάρτηση*, που με τη σειρά της αποτελείται από:

- μια επικεφαλίδα: "`int main()`",
- το *σώμα* της συνάρτησης.

Όπως θα δούμε αργότερα, μέσα στις παρενθέσεις, μπορεί να υπάρχουν οι παράμετροι της συνάρτησης.

Το **σώμα** (body), είναι το μέρος που δίνουμε στον υπολογιστή τις οδηγίες που θέλουμε να εκτελέσει. Προς το παρόν θα έχει μια απλή δομή: θα ξεκινάει με το σύμβολο "`{`" και θα τελειώνει με το σύμβολο "`}`". Ανάμεσα σε αυτά υπάρχουν εντολές που κάθε μια τελειώνει με το χαρακτήρα "`;`".

- ***Από δυο εντολές που γράφονται διαδοχικώς, η εντολή που γράφεται πρώτη θα εκτελεστεί, χρονικώς, πριν από την εντολή που γράφεται δεύτερη.***

Ας δούμε τι αποτέλεσμα θα είχε η αντιμετάθεση των δυο τελευταίων εντολών του προγράμματός μας:

```
#include <iostream>
using namespace std;
int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
}
```

Αποτέλεσμα:

```
 Να το πρώτο μου πρόγραμμα
 ΔΟΥΛΕΥΕΙ!
 ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
 Είμαι στο τρένο της Τεχνολογίας
```

Ανάμεσα σε δυο εντολές, μπορεί να υπάρχουν οσαδήποτε κενά και οσεσδήποτε αλλαγές γραμμής. Αυτό δεν θα αλλάξει το αποτέλεσμα. Το παράδειγμά μας θα μπορούσε να γραφεί το ίδιο σωστά ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
cout<<" Να το πρώτο μου πρόγραμμα"<<endl;cout<<
" ΔΟΥΛΕΥΕΙ!"<<
endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

ή ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;

   cout << " ΔΟΥΛΕΥΕΙ!" << endl;

   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;

   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Οι πεπειραμένοι προγραμματιστές χρησιμοποιούν τα κενά και τις αλλαγές γραμμής για να κάνουν τα προγράμματά τους πιο *ευανάγνωστα*.

Και εκείνο το

```
#include <iostream>
```

τι είναι; Είναι μια οδηγία προς το μεταγλωττιστή να περιλάβει στο σημείο αυτό ένα αρχείο, με το όνομα `iostream` (μερικά περιβάλλοντα μπορεί να έχουν διαφορετική κατάληξη) που έρχεται μαζί με το μεταγλωττιστή. Αυτό είναι απαραίτητο για να μπορέσει ο μεταγλωττιστής να καταλάβει τα "`cout`", "`<<`", "`endl`". Αν θέλεις να καταλάβεις –κάπως– τι γίνεται, διάβασε την επόμενη παράγραφο· αλλιώς θα τη γράφεις έτσι κι ας μην καταλαβαίνεις τι συμβαίνει.

Τα "`main`", "`cout`" και "`endl`" είναι **ονόματα** (identifiers). Είναι προκαθορισμένα για μερικά χρήσιμα αντικείμενα που χρησιμοποιούμε στα προγράμματά μας. Αργότερα θα δούμε πώς μπορούμε να ορίσουμε δικά μας ονόματα.

1.4 * Η Οδηγία "include"

Ας κάνουμε το εξής πείραμα: Φύλαξε σε ένα αρχείο, με το όνομα **entoles.txt**, τα εξής:

```
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
```

ενώ στο **prwto.cpp** φύλαξε τα:

```
#include <iostream>
using namespace std;
int main()
{
#include "entoles.txt"
}
```

Τώρα ζήτησε να μεταγλωττισθεί το **prwto.cpp** και να εκτελεσθεί το **prwto.exe**.[2] Αποτέλεσμα; Αυτό που ήδη ξέρεις:

```
Να το πρώτο μου πρόγραμμα
ΔΟΥΛΕΥΕΙ!
Είμαι στο τρένο της Τεχνολογίας
ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
```

Βγάζεις κάποιο συμπέρασμα για την include; Αποτέλεσμα της

```
#include "entoles.txt"
```

είναι να αντικατασταθεί, πριν από τη μεταγλώττιση, από το περιεχόμενο του αρχείου **entoles.txt**. Έτσι, το πρόγραμμα που πηγαίνει για μεταγλώττιση είναι αυτό που γράψαμε αρχικά, στην §1.2.

Δηλαδή και η "**#include <iostream>**" αντικαθίσταται από κάποιο αρχείο; Ναι, από το αρχείο **iostream**[3]· αν η Dev-C++ και είναι φυλαγμένη στο **c:\Dev-Cpp**, το **iostream** βρίσκεται στο **c:\Dev-Cpp\include\c++\3.4.2**. Μπορείς να το δεις στον κειμενογράφο σου[4]. Στο αρχείο αυτό υπάρχει ο ορισμός του *ρεύματος cout* και του *τελεστή* "**<<**".

Τώρα, θα ρωτήσεις: Γιατί το δικό μας αρχείο το γράψαμε μέσα σε αποστρόφους ενώ το **iostream** το βάζουμε μέσα σε "**<**" και "**>**"; Αυτό καθορίζει τη σειρά με την οποία θα ψάξει ο μεταγλωττιστής διάφορους καταλόγους για να βρει το αρχείο που ζητάμε.

Οι μεταγλωττιστές της C περιλαμβάνουν ένα πρόγραμμα, τον **προεπεξεργαστή** (preprocessor). Ο προεπεξεργαστής, εκτός των άλλων είναι αυτός που εκτελεί τις "**include**" και δημιουργεί το «συμπληρωμένο» αρχείο που θα επεξεργαστεί ο μεταγλωττιστής. Για αυτόν το λόγο η

#include <*όνομα αρχείου***>** ή

#include "*όνομα αρχείου***"**

λέγεται **οδηγία** (directive) **προς τον προεπεξεργαστή**.

Και ποια η διαφορά της πρώτης μορφής από τη δεύτερη;

- Στην πρώτη περίπτωση ο προεπεξεργαστής θα αναζητήσει το αρχείο σε ευρετήρια που είναι προκαθορισμένα για το κάθε περιβάλλον ανάπτυξης. Για παράδειγμα στη Dev-C++, που λέγαμε πιο πάνω, στο **c:\Dev-Cpp\include** και τα υποευρετηριά του.
- Στη δεύτερη περίπτωση η αναζήτηση θα γίνει κατ' αρχάς στο ευρετήριο που βρίσκεται το πρόγραμμά σου. Αν δεν το βρει θα συνεχίσει την αναζήτηση σαν να είχαμε δώσει "**#include <***όνομα αρχείου***>**".

---

[2] Ή όπως αλλιώς ονομάζει το τελικό (εκτελέσιμο) πρόγραμμα το ΛΣ που δουλεύεις.

[3] Μπορεί και **iostream.h** μπορεί και **iostream.hpp**.

[4] Αλλά, για το καλό σου, μην το πειράξεις! Είπαμε να το δεις μόνον!

1.5 Ορμαθοί Χαρακτήρων

Στο πρώτο μας πρόγραμμα, στις εντολές εκτύπωσης, χρησιμοποιήσαμε τα:

```
" Να το πρώτο μου πρόγραμμα"
" ΔΟΥΛΕΥΕΙ!"
" Είμαι στο τρένο της Τεχνολογίας"
" ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!"
```

Αυτά είναι παραδείγματα ορμαθών χαρακτήρων.

Ένας **ορμαθός χαρακτήρων**[5] (string literal) είναι μια ακολουθία χαρακτήρων που περικλείνεται μέσα σε διπλές αποστρόφους (**"**). Οι χαρακτήρες ενός ορμαθού δεν είναι απαραίτητο να ανήκουν στο σύνολο χαρακτήρων της C++. Αρκεί να υπάρχουν στο σύνολο χαρακτήρων του υπολογιστή μας.

Παραδείγματα

```
"α"   "A"   ";"   "α1"   "C++"
"Αυτά είναι νέα"   "12/15*()="    "   "
```

Κατ' αρχήν, *μέσα σ' έναν ορμαθό δεν μπορούμε να έχουμε αλλαγή γραμμής*. Δηλαδή, η αρχική και η τελική διπλή απόστροφος πρέπει να βρίσκονται στην ίδια γραμμή.

Μήκος (length) του ορμαθού είναι το πλήθος των χαρακτήρων που υπάρχουν ανάμεσα στις αποστρόφους.

Προσοχή! Το *διάστημα* (*κενό*) *είναι* χαρακτήρας και το μετράμε στο μήκος.

Οι ορμαθοί που δώσαμε πιο πάνω έχουν μήκη: 1, 1, 1, 2, 3, 14, 9, 3 αντιστοίχως.

Όπως καταλαβαίνεις υπάρχει ένα μικρό πρόβλημα όταν θελήσουμε να περιλάβουμε σε έναν ορμαθό τη διπλή απόστροφο (**"**). Το πρόβλημα λύνεται με την εξής σύμβαση: *Αν θέλουμε να περιλάβουμε μια διπλή απόστροφο σε έναν ορμαθό, τότε πρέπει να την γράψουμε ως* **\"**, που όπως είπαμε ονομάζεται **ακολουθία διαφυγής** (escape sequence). Φυσικά, στον υπολογισμό του μήκους, οι δυο αυτοί χαρακτήρες μετρούνται ως ένας. Με ακολουθίες διαφυγής εισάγονται και οι χαρακτήρες ****, **?**, **'**.

Παραδείγματα

```
"\""  "What\'s up man\?"  "T\' AHTOY TO NYXI"
```

Αυτοί οι ορμαθοί έχουν μήκη: 1, 14, 16 αντιστοίχως.

Στα παραδείγματα που δώσαμε παραπάνω, χρησιμοποιήσαμε και χαρακτήρες που δεν ανήκουν στο σύνολο χαρακτήρων της C++. Στους ΗΥ που έχουμε στην Ελλάδα υπάρχουν και τα ελληνικά γράμματα. Μπορείς να χρησιμοποιείς τους χαρακτήρες αυτούς αλλά να θυμάσαι ότι: *αν μεταφέρεις το πρόγραμμά σου σε άλλον ΗΥ είναι πολύ πιθανό να χρειαστεί να αλλάξεις αυτούς τους χαρακτήρες.*[6]

Ας δούμε ένα άλλο παράδειγμα όπου χρησιμοποιούμε ορμαθούς χαρακτήρων.

Παράδειγμα

```
#include <iostream>
using namespace std;
int main()
{
   cout << "   *   " << endl;
   cout << "  * *  " << endl;
   cout << " ***** " << endl;
   cout << "*     *" << endl;
}
```

[5] Μπορεί να το δεις και ως **συμβολοσειρά**.

[6] Μήπως βλέπεις ήδη το πρόβλημα σε διαφορετικά περιβάλλοντα του ίδιου υπολογιστή (αν δουλεύεις σε Windows);!

Αυτό το πρόγραμμα θα δώσει:

```
   *
  * *
 *****
*     *
```

Μπορείς να προκαλέσεις αλλαγή γραμμής όταν γράφεται (π.χ. στην οθόνη) ο ορμαθός, εισάγοντας σε αυτόν την *ακολουθία διαφυγής* "`\n`". Δηλαδή, η

```
   cout << "abc\ndef" << endl;
```

θα γράψει:

```
abc
def
```

1.5.1 Ορμαθοί Μεγάλου Μήκους

Ο περιορισμός «*μέσα σ' έναν ορμαθό δεν μπορούμε να έχουμε αλλαγή γραμμής*» περιορίζει και το μήκος του ορμαθού. Αν τώρα θέλεις να γράψεις έναν πολύ μακρύ ορμαθό έχεις δύο τρόπους:

Ο πρώτος τρόπος είναι να τον σπάσεις σε διαφορετικές γραμμές χρησιμοποιώντας τον χαρακτήρα '\'. Π.χ., η

```
   cout << "abc\
def" << endl;
```

θα γράψει:

```
abcdef
```

Όπως θα δεις και αργότερα, ο χαρακτήρας '\' είναι η λύση που δίνει η C++ όπου υπάρχει περιορισμός μη αλλαγής γραμμής.

Ο δεύτερος τρόπος είναι η πράξη της σύνδεσης: Αν στο πρόγραμμά σου γράψεις δύο ορμαθούς χαρακτήρων στη σειρά, η C++ θα σχηματίσει την **σύνδεσή** (concatenation) τους, που προκύπτει από τους χαρακτήρες του πρώτου ορμαθού ακολουθούμενους από αυτούς του δεύτερου. Π.χ. η εντολή:

```
   cout << "abc" "def" << endl;
```

θα δώσει:

```
abcdef
```

1.6 Έξοδος Αποτελεσμάτων

Η πρώτη εντολή που είδαμε ήταν η εντολή

`cout << "`*κείμενο*`" << endl`

Το αποτέλεσμά της είναι να γραφεί μια γραμμή. Πού; Μάλλον είδες αυτή τη γραμμή στην οθόνη του μικροϋπολογιστή σου (ή του τερματικού σου).

Κάθε ΗΥ έχει μια προκαθορισμένη συσκευή, ή αρχείο, όπου γίνεται η έξοδος των αποτελεσμάτων, εκτός αν ο χρήστης ορίσει κάτι άλλο. Είναι αυτό που τα εγχειρίδια για τον χρήστη αναφέρουν ως *default output device* (*file*). Τα πιο συνηθισμένα παραδείγματα είναι κάποια οθόνη ή ο εκτυπωτής. Αυτή η συσκευή (αρχείο), σε κάθε υλοποίηση της C++, συνδέεται με το πρόγραμμά σου με ένα προκαθορισμένο *ρεύμα* με το όνομα `cout`.

Να δούμε τώρα τη μορφή της εντολής. Η πιο απλή μορφή είναι:

```
cout << endl;
```

Το αποτέλεσμά της είναι να «γραφεί» μια κενή γραμμή.

Γενικώς, μετά τη λέξη *cout* βάζουμε τα *ορίσματά* της, που μεταξύ τους διαχωρίζονται με τους χαρακτήρες "`<<`". Προς το παρόν, τα ορίσματα μπορεί να είναι ορμαθοί χαρακτήρων.

Αποτέλεσμα της εκτέλεσης είναι να γραφεί μια γραμμή που περιέχει τα περιεχόμενα όλων των ορμαθών, με τη σειρά που δίνονται και χωρίς διαχωρισμό μεταξύ τους.

Παράδειγμα

Η εντολή:

```
cout << " ΜΗΛΟ" << "ΠΙΤΑ" << endl;
```

γράφει:

```
 ΜΗΛΟΠΙΤΑ
```

ενώ η εντολή:

```
cout << " AB" << "CD" << "EF" << endl;
```

γράφει:

```
 ABCDEF
```

Και τι θα γίνει αν δεν βάλουμε το `endl`; Τότε, δεν θα έχουμε αλλαγή γραμμής. Π.χ. οι:

```
cout << "abc";
cout << "def" << endl;
```

θα δώσουν:

```
abcdef
```

Αντί για το `endl` μπορεί να δεις να βάζουν το `'\n'`, π.χ.:

```
cout << " AB" << "CD" << "EF" << '\n';
```

Είναι *σχεδόν* το ίδιο πράγμα.

1.7 Αριθμητικές Πραγματικές Σταθερές

Τι είναι μια **σταθερά** (constant) στα μαθηματικά; Είναι μια ποσότητα που δεν αλλάζει η τιμή της. Π.χ.:

```
37,4    -0,1    -1,1    41,00   +10,00017
```

Ας δούμε τις συνιστώσες αυτών των σταθερών και το νόημά τους:

- Στην αρχή μπορεί να έχουμε ένα πρόσημο "`+`" ή "`-`". Αν δεν υπάρχει εννοείται το "`+`".
- Μετά, ακολουθούν τα ψηφία του ακέραιου μέρους. Στα παραδείγματά μας είναι: 37, 0, 1, 41, 10.
- Στη συνέχεια έχουμε την *υποδιαστολή* που διαχωρίζει το ακέραιο από το κλασματικό μέρος. Τη συμβολίζουμε με το "`,`".
- Τέλος υπάρχει το κλασματικό ή δεκαδικό μέρος, που είναι μια ακολουθία ψηφίων.

Οι θετικοί επιστήμονες και οι μηχανικοί σε μερικές περιπτώσεις χρησιμοποιούν την εξής σύμβαση: Αντί να γράψουν

0,000000173 γράφουν $1{,}73\times10^{-7}$

αντί να γράψουν:

3000000000 γράφουν 3×10^{9}

Αυτός ο τρόπος γραφής λέγεται **επιστημονική παράσταση** (scientific notation) ή **παράσταση κινητής υποδιαστολής** (floating point). Σε αντιδιαστολή, η συνηθισμένη γραφή λέγεται **παράσταση σταθερής υποδιαστολής** (fixed point).

Η C++ και οι περισσότερες γλώσσες προγραμματισμού ακολουθούν τους παραπάνω κανόνες με τις εξής διαφορές:

- σημειώνει την υποδιαστολή με τελεία ("`.`") αντί για κόμμα (αμερικανική γραφή αντί της ευρωπαϊκής),

- σημειώνει τη δύναμη του 10 με *eN* αντί για $\times 10^N$. Το *eN*, στην περίπτωση αυτή διαβάζεται –και σημαίνει– «επί 10 στη *N*». Π.χ. το 0.375e3 διαβάζεται «0.375 επί 10 στη 3η» (=375). Ο ακέραιος που ακολουθεί το *e* λέγεται **εκθέτης** (exponent).

Τα παραπάνω παραδείγματα γράφονται στη C++ ως εξής:

Αριθμητική	**C++**
37,4	`37.4`
–0,1	`-0.1`
–1,1	`-1.1`
41,00	`41.00`
10,00017	`10.00017`
$1{,}73\times10^{-7}$	`1.73e-7`
3×10^{9}	`3e9`

Το νόημα των «μεταφρασμένων» σταθερών είναι το ίδιο μ' αυτό που έχουν οι αντίστοιχες αριθμητικές σταθερές.

Τις σταθερές που είδαμε πιο πάνω ονομάζουμε **σταθερές κινητής υποδιαστολής** (floating point constants ή literals) ή *πραγματικές* σταθερές. Δίνουμε πρώτα το συντακτικό τους σε BNF:

σταθερά κινητής υποδιαστολής = *κλασματική σταθερά* [*τμήμα εκθέτη*] |
ακολουθία ψηφίων, *τμήμα εκθέτη*;
κλασματική σταθερά = [*ακολουθία ψηφίων*] "`.`", *ακολουθία ψηφίων* |
ακολουθία ψηφίων, "`.`";
τμήμα εκθέτη = "`e`" [*πρόσημο*] *ακολουθία ψηφίων* |
"`E`" [*πρόσημο*] *ακολουθία ψηφίων*;
πρόσημο = "`+`" | "`-`";
ακολουθία ψηφίων = *ψηφίο* | *ακολουθία ψηφίων*, *ψηφίο*;

Δηλαδή:

- Στην αρχή μπορεί να έχουμε μια κλασματική σταθερά, π.χ.:

```
1234.5678      .1357      2345.
```

Μια κλασματική σταθερά *είναι* σταθερά κινητής υποδιαστολής.

- Η κλασματική σταθερά μπορεί να ακολουθείται από ένα τμήμα εκθέτη που αποτελείται από ένα "`e`" ή "`E`", ένα πρόσημο "`+`" ή "`-`", που μπορεί να παραλείπεται και τέλος μια ακολουθία ψηφίων, π.χ.:

```
1234.5678e+12   1234.5678E-12   1234.5678e12
  .1357e+15       .1357E-15      .1357e15
 2345.e+15       2345.E-15      2345.e15
```

- Στην αρχή μπορεί, αντί για κλασματική σταθερά, να έχουμε μια ακολουθία ψηφίων. Στην περίπτωση αυτή *πρέπει* να υπάρχει εκθέτης:

```
2345e+15       2345E-15       2345e15
```

Και το πρόσημο πριν από τα ψηφία; Σαφέστατα μπορεί να υπάρχει, αλλά η `-1.7e-7` θεωρείται από τη C++ παράσταση.

Εδώ θα πρέπει να προλάβουμε μια πολύ συνηθισμένη παρανόηση:

♦ ***Το "e" δεν σημειώνει πράξη!***

Έτσι, το `5.36e-8` συμβολίζει τον *αριθμό* 0.0000000536 και *όχι την πράξη* $5.36/10^8$.

Τι θα πει αυτό δηλαδή; Αν το "`e`" ήταν πράξη θα μπορούσες να γράψεις

```
5.36e-8e2
```

για να συμβολίσεις την πράξη 0.0000000536 × 100. Αλλά το "`5.36e-8e2`" είναι *συντακτικό λάθος*.

Τέλος, για τις σταθερές κινητής υποδιαστολής ισχύει το εξής:

♦ ***Όταν γράφεις μια πραγματική σταθερά δεν επιτρέπεται να παρεμβάλλεις κενά ή αλλαγές γραμμής.***

Σύμφωνα με τα παραπάνω, τα παρακάτω είναι σταθερές κινητής υποδιαστολής (οι δύο μαζί με ένα πρόσημο):

```
5.67e+4      12.0    56700.0  0.00001  -0.0
    +0.70135e+1  7.0135  .12       19.
```

ενώ τα παρακάτω *δεν είναι:*

`23A`	έχει το γράμμα "`A`"
`3,14`	έχει το "`,`"
`7×10²`	τα "`×`" και "`²`" δεν αναγνωρίζονται
`3 . 14`	έχει κενά
`317.`	έχει αλλαγή γραμμής (και κενά)
`123`	

Μπορείς να ζητήσεις την εκτύπωση μιας αριθμητικής σταθεράς όπως ακριβώς κάνεις και με τους ορμαθούς χαρακτήρων. Τι θα πάρεις όμως; Για δες το

Παράδειγμα

```
#include <iostream>
using namespace std;
int main()
{
   cout << 5.67e+4 << endl;  cout << 12.0 << endl;
   cout << -56700.0 << endl; cout << 0.00001 << endl;
   cout << -0.0 << endl;     cout << +0.70135e+1 <<endl;
   cout << 7.0135 << endl;   cout << .12 << endl;
   cout << 19. << endl;
}
```

Αποτέλεσμα:

```
56700
12
-56700
1e-05
0
7.0135
7.0135
0.12
19
```

Τα αποτελέσματα είναι σωστά, αλλά δεν τυπώθηκαν όπως τα γράψαμε στο πρόγραμμά μας.

Από το τελευταίο παράδειγμα καταλαβαίνουμε ότι:

- Μπορούμε σε μια εντολή εκτύπωσης να βάζουμε ως ορίσματα και αριθμητικές σταθερές.
- Η τιμή της σταθεράς τυπώνεται όχι κατ' ανάγκη όπως τη γράψαμε στο πρόγραμμά μας. Αυτό γίνεται διότι η σταθερά «μεταφράζεται» στην εσωτερική παράσταση που χρησιμοποιεί η C++ για τις σταθερές κινητής υποδιαστολής και στη συνέχεια τυπώνεται με τους κανόνες που γίνεται η εκτύπωση τέτοιων τιμών.

1.7.1 Εσωτερική Παράσταση Πραγματικών Τιμών

Για να καταλάβεις πώς αποθηκεύονται οι τιμές κινητής υποδιαστολής στη μνήμη του ΗΥ, σκέψου ότι γράφονται σε μορφή κινητής υποδιαστολής αλλά στο δυαδικό σύστημα:

$$\sigma\ M \times 2^E$$

όπου το σ είναι πρόσημο και οι M, E δυαδικοί αριθμοί. Στη μνήμη αποθηκεύονται τα:

- σ: σε ένα δυαδικό ψηφίο (0 για το "+", 1 για το "-"),
- M: σε πλήθος δυαδικών ψηφίων που εξαρτάται από την υλοποίηση ή/και τον ΗΥ,
- E: σε πλήθος δυαδικών ψηφίων που εξαρτάται από την υλοποίηση ή/και τον ΗΥ.

Όπως βλέπεις κάθε τιμή κινητής υποδιαστολής πιάνει στη μνήμη τον ίδιο χώρο, που εξαρτάται από τον ΗΥ (ή το λογισμικό του) αλλά όχι από την τιμή που αποθηκεύεται. Αποτέλεσμα; το βλέπεις στο παρακάτω

Παράδειγμα

```
#include <iostream>
using namespace std;
int main()
{
   cout << 12.3456 << endl;
   cout << 12.34567890123456789 << endl;
}
```

Αποτέλεσμα:

```
12.3456
12.3457
```

Τι έγινε εδώ πέρα; Στη δεύτερη σταθερά, εμείς δώσαμε δεκαεννιά ψηφία και μας έβγαλε έξη! Μήπως κάτι δεν πάει καλά με το γράψιμο; Πράγματι, η C++ μας επιτρέπει να καθορίσουμε την **ακρίβεια** (precision) του αποτελέσματος· δες τι μπορούμε να κάνουμε:

```
#include <iostream>
using namespace std;
int main()
{
   cout << 12.3456 << endl;
   cout.precision(20);
   cout << 12.34567890123456789 << endl;
}
```

Όπως θα δούμε και αργότερα, με τη `cout.precision(20)` ζητούμε ακρίβεια είκοσι ψηφίων στα αποτελέσματα που παίρνουμε στο *cout*. Τι θα πάρουμε τώρα;

```
12.3456
12.345678901234567350
```

Μέχρι το 17ο ψηφίο πήγαμε καλά. Από εκεί και πέρα… Εδώ *φτάσαμε στα όρια της ακρίβειας της εσωτερικής παράστασης* και η ακρίβεια στο γράψιμο δεν μπορεί να μας βοηθήσει πια. Έχουμε δηλαδή **απώλεια σημαντικών ψηφίων** (loss of significant digits) κατά την εσωτερική παράσταση της τιμής.

Απώλεια σημαντικών ψηφίων έχουμε και όταν κάνουμε πράξεις με τιμές κινητής υποδιαστολής.

Ο τρόπος εσωτερικής παράστασης βάζει και έναν άλλο περιορισμό: υπάρχει κάποια μέγιστη (και κάποια ελάχιστη) τιμή που μπορεί να παρασταθεί. Αν, κατά κάποιο τρόπο, προκύψει κάποια τιμή έξω από τα προκαθορισμένα όρια έχουμε **υπερχείλιση** (overflow). Τι γίνεται στην περίπτωση αυτή; Εξαρτάται από τη συγκεκριμένη υλοποίηση της C++.

1.8 Ακέραιες Σταθερές

Σε πάρα πολλές περιπτώσεις μπορούμε να δουλέψουμε στα προγράμματά μας με ακέραιες τιμές μόνον. Ας ξεκινήσουμε από τα συντακτικό της **ακέραιης σταθεράς** (integer constant ή literal):

ακέραιη σταθερά = *δεκαδική σταθερά* | "`0`";
δεκαδική σταθερά = *μη-μηδενικό ψηφίο* | *δεκαδική σταθερά*, *ψηφίο*;
μη-μηδενικό ψηφίο = "`1`" | ... | "`9`";

Δηλαδή, μια ακέραιη σταθερά είναι (προς το παρόν) το 0 ή *μια ακολουθία ψηφίων που **δεν** αρχίζει από* 0.

Με το πρόσημο ισχύουν τα ίδια που είπαμε για τις πραγματικές σταθερές, π.χ. το `-375` έχει το νόημα που ξέρουμε, αλλά η C++ το βλέπει ως παράσταση[7].

Παραδείγματα

Τα:

```
12  0  417  375  2048
```

είναι ακέραιες σταθερές. Τα παρακάτω *δεν είναι ακέραιες σταθερές*:

`12.0` έχει την τελεία (“`.`”)
`1E100` έχει το “`E`”
`0,0` έχει το κόμμα (“`,`”)

Οι ακέραιες τιμές αποθηκεύονται με ακρίβεια στη μνήμη του ΗΥ, οι πράξεις τους γίνονται με ακρίβεια και είναι ταχύτερες από αυτές των τιμών κινητής υποδιαστολής. Φυσικά, οι ακέραιες τιμές δεν μπορούν να έχουν κλασματικό μέρος. Ακόμη, τα όριά τους είναι πιο στενά από αυτά που επιτρέπονται για τιμές κινητής υποδιαστολής.

Όπως τυπώνουμε ορμαθούς χαρακτήρων και τιμές κινητής υποδιαστολής μπορούμε να τυπώνουμε και ακέραιες τιμές. Π.χ. η:

```
cout << 12 << endl;
```

δίνει:

```
12
```

1.8.1 Οκταδικοί Ακέραιοι

Γιατί είπαμε «μια ακέραιη σταθερά είναι το 0 ή *μια ακολουθία ψηφίων που **δεν** αρχίζει από* 0»; Τι συμβαίνει με το 0;

Μια ακολουθία ψηφίων που αρχίζει από 0 είναι *οκταδική* (octal) *ακέραιη σταθερά*· έτσι, δεν μπορείς να γράψεις `08` και `09` (θα πάρεις ένα διαγνωστικό σαν “invalid octal constant”) αφού τα "`8`" και "`9`" δεν είναι ψηφία του οκταδικού συστήματος ενώ το `011` σημαίνει 9. Πράγματι, η εντολή

```
cout << 11 << "  " << 011 << endl;
```

θα δώσει:

```
11  9
```

1.8.2 Δεκαεξαδικοί Ακέραιοι

Η C++ σου επιτρέπει να γράφεις ακέραιες σταθερές στο *δεκαεξαδικό* (hexadecimal) σύστημα αρίθμησης. Αρχίζουν με “`0x`” ή “`0X`” και στη συνέχεια έχουν δεκαεξαδικά ψηφία: `0..9` και `a..f` (ή `A..F`). Για παράδειγμα η

```
cout << 0xA << " " << 0xff << " " << 0Xa12C << endl;
```

θα δώσει:

```
10 255 41260
```

1.9 Πράξεις – Αριθμητική Παράσταση

Ένα άλλο είδος ορίσματος για την εντολή εκτύπωσης είναι η **αριθμητική παράσταση** (arithmetic expression). Π.χ. με την εντολή:

```
cout << (1 + 1) << endl;
```

ζητούμε από τον ΗΥ να υπολογίσει το αποτέλεσμα της πράξης `1 + 1` και να το τυπώσει.

---

[7] Το πρόσημο είναι ένας *ενικός* τελεστής, δηλαδή τελεστής που *δρα σε ένα αντικείμενο*.

Πλαίσιο 1.1

Οι Αριθμητικοί Τελεστές της C++

Αριθμητική	C++
+	+
–	-
×	*
/	/
υπόλοιπο:	%

Υπολογισμός Αριθμητικής Παράστασης στη C++

1. Γίνονται οι πράξεις μέσα στις παρενθέσεις και υπολογίζονται οι κλήσεις συναρτήσεων,

2. Μετά γίνονται οι πράξεις *, /, %,

3. Στη συνέχεια γίνονται οι πράξεις +, -.

Οι αριθμητικές πράξεις στη C++ συμβολίζονται όπως και στα μαθηματικά, με μια διαφορά: συμβολίζουμε τον πολλαπλασιασμό με το "`*`" αντί για το "×". Ακόμη, ως σύμβολο της διαίρεσης χρησιμοποιούμε μόνον το "`/`" και όχι το "`:`".

Αν και τα δύο ορίσματα μιας πράξης είναι ακέραιοι το αποτέλεσμα της πράξης είναι ακέραιος, ενώ αν έστω και ένα είναι πραγματικός (π.χ. σταθερά κινητής υποδιαστολής) τότε το αποτέλεσμα είναι πραγματικός[8]. Αυτό, τουλάχιστον προς το παρόν, δεν σημαίνει και πολλά πράγματα εκτός από την περίπτωση της διαίρεσης. Δες το παρακάτω

Παράδειγμα

Το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   cout <<  (1 / 2) << "   " << (1. / 2) << endl;
}
```

δίνει:

```
0   0.5
```

Πώς εξηγείται; Το 0 είναι το αποτέλεσμα της *ακέραιης διαίρεσης* `1 / 2` και, όπως βλέπεις, *δεν* είναι 0.5. Αφού διαιρετέος και διαιρέτης είναι ακέραιοι, ακέραιο είναι και το αποτέλεσμα της διαίρεσης. Η δεύτερη διαίρεση, `1. / 2`, έχει διαιρετέο πραγματικό και έτσι το αποτέλεσμα είναι πραγματικός.

Ειδικά για τη διαίρεση ακεραίων, υπάρχει ένας ακόμη τελεστής ο "`%`" που κάνει μια πολύ γνωστή δουλειά: *μας δίνει το υπόλοιπο της ακέραιης διαίρεσης*. Σχετίζεται με τον "`/`" με τη –γνωστή από την αριθμητική «δοκιμή της διαίρεσης»– σχέση:

$$\Delta = \pi * \delta + \upsilon$$

όπου Δ ο διαιρετέος και δ ο διαιρέτης. Αλλιώς:

$$\Delta = (\Delta \ / \ \delta) * \delta + (\Delta \ \% \ \delta) \qquad (1)$$

Έτσι,

η πράξη: `5 / 2` δίνει 2

η πράξη: `5 % 2` δίνει 1

[8] Στο επόμενο κεφάλαιο θα δεις ότι ο κανόνας είναι πιο πολύπλοκος.

Όπως είναι φυσικό, είναι λάθος να βάλεις δεύτερο όρισμα σε οποιαδήποτε από τις δύο πράξεις το 0 (μηδέν).

Αν κάποιο από τα ορίσματα είναι αρνητικό τότε το αποτέλεσμα της "**%**" είναι τέτοιο ώστε να ισχύει η (1)· δηλαδή *μπορεί να είναι και αρνητικό*! Π.χ. το:

```
#include <iostream>
using namespace std;
int main()
{
   cout <<  (1 / 2) << "   " << (1 % 2) << "   "
        << ((1/2)*2 + (1%2)) << endl;
   cout <<  (1 / (-2)) << "   " << (1 % (-2)) << "   "
        << ((1/(-2))*2 + (1%(-2))) << endl;
   cout <<  ((-1) / 2) << "   " << ((-1) % 2) << "   "
        << (((-1)/2)*2 + ((-1)%2)) << endl;
   cout <<  ((-1) / (-2)) << "   " << ((-1) % (-2)) << "   "
        << (((-1)/(-2))*(-2) + ((-1)%(-2))) << endl;
}
```

θα δώσει:

```
0   1   1
0   1   1
0   -1   -1
0   -1   -1
```

Ήδη στο παραπάνω παράδειγμα βλέπουμε παραστάσεις με περισσότερες από μια πράξεις και με παρενθέσεις. Ας δούμε τι γίνεται σε τέτοιες περιπτώσεις. Π.χ. ας πούμε ότι έχουμε:

```
7*5 - 31.0/2 + 9*2/5.0
```

Πώς θα υπολόγιζες το αποτέλεσμα στην αριθμητική; Πρώτα θα υπολογίσεις τις τιμές των όρων:

```
7*5 = 35
31.0/2 = 15.5
9*2/5.0 = 3.6
```

Δηλαδή πρώτα θα κάνεις πολλαπλασιασμούς και διαιρέσεις. Στη συνέχεια θα κάνεις προσθέσεις και αφαιρέσεις:

```
35 - 15.5 + 3.6 = 23.1
```

Το ίδιο γίνεται και στη C++:

- πρώτα γίνονται οι πράξεις *****, **/**, **%**,
- στη συνέχεια γίνονται οι πράξεις **+**, **-**.

Ωραία, αλλά στην αριθμητική μπορούμε να παραβιάσουμε αυτή τη σειρά των πράξεων με τις παρενθέσεις, με αγκύλες, με άγκιστρα. Εδώ τι γίνεται; Το ίδιο πράγμα, αλλά χρησιμοποιούμε μόνον παρενθέσεις.

♦ ***Ό,τι υπάρχει μέσα στις παρενθέσεις υπολογίζεται πρώτο.***

Η παράσταση:

$$1 + \{ 1 + 4 \times [6 + 2 \times (9 - 3.5) / 2.2] \}$$

θα γραφεί στη C++:

```
1 + ( 1 + 4 * ( 6 + 2 * ( 9 - 3.5 ) / 2.2 ) )
```

Η διαφορά είναι αμελητέα. Μόνο πρόσεχε!

♦ ***Όσες παρενθέσεις ανοίγεις τόσες ακριβώς πρέπει να κλείνεις και –προ πάντων– στο σωστό σημείο.***

Με βάση τους κανόνες που είπαμε μέχρι τώρα, μπορείς να γράψεις σωστά στη C++ σχεδόν οποιαδήποτε αριθμητική παράσταση. *Όταν έχεις κάποια αμφιβολία για την σειρά εκτέλεσης των πράξεων, χρησιμοποίησε παρενθέσεις*, αφού ό,τι βρίσκεται μέσα σ' αυτές υπολογίζεται πρώτο.

Παράδειγμα

Έστω ότι θέλουμε να υπολογίσουμε το:

$$\frac{3.45}{5\times 3.14159}$$

Αν γράψουμε:

```
3.45 / 5 * 3.14159
```

κάνουμε *λάθος*. Όπως είπαμε πιο πριν, οι "`*`" και "`/`" έχουν την ίδια προτεραιότητα. Αν οι πράξεις γίνονται από αριστερά προς τα δεξιά, πρώτα θα υπολογιστεί το: 3.45/5 και ότι βρεθεί θα *πολλαπλασιαστεί* με το 3.14159. Δηλαδή, θα υπολογιστεί το:

$$\frac{3.45}{5}\times 3.14159$$

Χρησιμοποιώντας παρενθέσεις γράφουμε το σωστό:

```
3.45/(5 * 3.14159)
```

Τώρα, θα υπολογιστεί η τιμή της παράστασης μέσα στις παρενθέσεις και μετά θα διαιρεθεί το 3.45 με αυτό που βρήκαμε από τον πολλαπλασιασμό.

1.10 Οι Συναρτήσεις της C++

Στην προηγούμενη παράγραφο λέγαμε ότι τώρα μπορείς να μεταγράψεις σχεδόν κάθε μαθηματική παράσταση σε C++. Καλά που βάλαμε το «σχεδόν», γιατί εσύ θέλεις να μεταγράψεις τη:

$$\frac{\text{συν}(72^\circ)}{1+\sqrt{5}} \qquad \text{και τη} \qquad 2.5+\sqrt{5^2-3^2}$$

Κανένα πρόβλημα! Νάτες:

```
cos(72 * 3.14159 / 180) / (1 + sqrt(5))
2.5 + sqrt(5*5 - 3*3)
```

Τα καινούρια πράγματα εδώ είναι τα εξής: `cos(3.14159 * 72 / 180)`, `sqrt(5)`, `sqrt(5*5 - 3*3)`. Πρόκειται για κλήσεις δυο συναρτήσεων της C++ με τα ονόματα **`cos`**, **`sqrt`**, που μας δίνουν το συνημίτονο και την τετραγωνική ρίζα αντίστοιχα. Εντάξει! Αρχικά όμως είχαμε *συν*(72°). Πώς εμφανίστηκαν αυτά τα 3.14159/180; Λοιπόν: η συνάρτηση *cos()* της C++ περιμένει το όρισμά της σε ακτίνια. Το $\pi/180$ είναι ο παράγοντας μετατροπής.

Κάθε συνάρτηση της C++ έχει ένα όνομα με το οποίο τη χρησιμοποιούμε. Πού; Μέσα σε αριθμητικές παραστάσεις. Γράφουμε το όνομα της συνάρτησης που θέλουμε και στη συνέχεια μέσα σε παρενθέσεις το όρισμα ή τα ορίσματά της. Όλο αυτό είναι μια **κλήση συνάρτησης** (function call).

- ***Σε μια παράσταση, η κλήση συνάρτησης έχει την ίδια προτεραιότητα με μια παράσταση μέσα σε παρενθέσεις, δηλαδή υπολογίζεται στην αρχή, πριν από τις πράξεις.***

  Ο υπολογισμός γίνεται ως εξής:

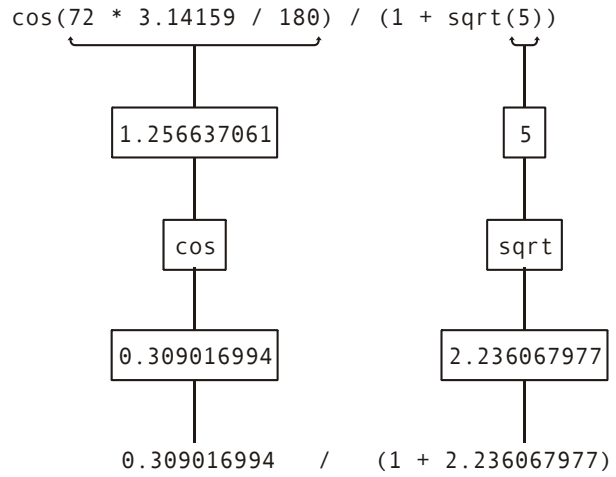
- Πρώτα υπολογίζεται η τιμή του ορίσματος,
- στη συνέχεια η τιμή της συνάρτησης και
- τέλος η τιμή αντικαθίσταται στη θέση της κλήσης συνάρτησης για να γίνουν οι άλλες πράξεις.

Στο Σχ. 1-1 βλέπεις πώς γίνονται τα παραπάνω για το πρώτο παράδειγμα.

Στο Παρ. Β βλέπεις όλες τις **συναρτήσεις** (functions) που υπάρχουν στη μαθηματική βιβλιοθήκη (math) της C++. Για να τις χρησιμοποιήσεις θα πρέπει στην αρχή του προγράμματός σου να βάλεις την οδηγία

```
#include <cmath>
```

Ας δούμε ένα παράδειγμα με μερικές από αυτές:



Σχ. 1-1 **Σειρά των πράξεων για τον υπολογισμό συνάρτησης.**

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   cout << sqrt( 2.0 ) << "    " << sqrt( 4.0*9 )<<endl;
   cout << exp( 1.0 ) << endl;
   cout << log( 1.0 ) <<"    "<<log( exp( 1.0 ) )<<endl;
   cout << sin( 3.141592653 / 3 ) << endl;
   cout << cos( 3.141592653 / 6 ) << endl;
   cout << atan( 1.0 ) << "  π = "<<4*atan( 1.0 )<<endl;
   cout << pow( 10.0,3.0 )<<"  "<<pow( 2.0, 0.5 )<<endl;
   cout << fabs( -7.0 ) <<  "    " << abs( 7 ) << endl;
}
```

1. ***sqrt:*** Μας δίνει την **τετραγωνική ρίζα** (**sq**uare **r**oot) του ορίσματος. Το αποτέλεσμά της είναι τιμή κινητής υποδιαστολής. Η παράσταση που θα μπει ως όρισμα, θα πρέπει να παίρνει τιμή *μη-αρνητική*. Η:

```
   cout << sqrt( 2.0 ) << "    " << sqrt( 4.0*9 )<<endl;
```

θα δώσει:

```
1.41421    6
```

2. ***exp:*** Το *exp*(*x*) μας δίνει το e^x, όπου *e* η βάση των φυσικών λογαρίθμων. Μπορείς να πάρεις από τον ΗΥ σου την τιμή του *e* με την:

```
   cout << exp( 1.0 ) << endl;
```

που θα σου δώσει:

```
2.71828
```

3. ***log:*** Μας δίνει τον **φυσικό λογάριθμο** του ορίσματος. Το όρισμά της πρέπει να έχει *θετική* τιμή. Είναι η αντίστροφη συνάρτηση της *exp()*. Π.χ. η:

```
   cout << log( 1.0 ) << "    " << log( exp(1.0) ) << endl;
```

θα σου δώσει:

```
0    1
```

4. ***sin:*** Μας δίνει το **ημίτονο** του ορίσματος (γωνίας). Το όρισμα θα πρέπει να είναι γωνία (τόξο) σε *ακτίνια*. Π.χ. η παρακάτω εντολή μας τυπώνει το ημίτονο των 60° (= *π*/3):

```
   cout << sin( 3.141592653 / 3 ) << endl;
```

αποτέλεσμα:

```
0.866025
```

5. *cos:* Μας δίνει το **συνημίτονο** του ορίσματος (γωνίας). Το όρισμα θα πρέπει να είναι γωνία (τόξο) σε *ακτίνια*. Π.χ. η παρακάτω εντολή μας τυπώνει το συνημίτονο των 30° (=π/6):

```
   cout << cos( 3.141592653 / 6 ) << endl;
```

αποτέλεσμα:

```
0.866025
```

6. *atan:* Μας δίνει το **τόξο εφαπτομένης** του ορίσματος. Δηλαδή, το τόξο –γωνία– σε ακτίνια, στο διάστημα (–π/2, π/2), που η εφαπτομένη του είναι ίση με το όρισμα.

Αν πάρεις υπόψη σου ότι $\varepsilon\phi(\pi/4) = 1$ και επομένως: $\pi = 4\tau o\xi\varepsilon\phi(1)$, μπορείς να δεις την τιμή του π, όπως την έχει η C++ που δουλεύεις. Η:

```
   cout << atan( 1.0 ) << "  π = " << 4*atan( 1.0 ) << endl;
```

θα δώσει:

```
0.785398  π = 3.14159
```

7. *pow:* Η *pow*(x, y) μας δίνει το x^y. Π.χ.

```
   cout << pow( 10. 0,3.0 ) << "  " << pow( 2.0, 0.5 ) <<endl;
```

αποτέλεσμα:

```
1000    1.41421
```

8. *abs, fabs:* Μας δίνουν την **απόλυτη τιμή** (**abs**olute value) του ορίσματος. Η *abs()* περιμένει ακέραιο όρισμα, ενώ η *fabs()* πραγματικό. Π.χ.

```
   cout << fabs(-7.0) <<  "    " << abs(7) << endl;
```

αποτέλεσμα:

```
7    7
```

1.10.1 "cmath" ή "math.h" ;

Σε πολλά μέρη είδες να γράφεται αντί για "`#include <cmath>`"

```
#include <math.h>
```

Ποιο είναι το σωστό; Το σωστό είναι `cmath` αλλά και το `math.h` θα δουλέψει.

Το `math` είναι κληρονομιά από τη C (ενώ το `iostream`, που χρησιμοποιούμε ήδη, είναι της C++). Ο κανόνας είναι ως εξής:

- Τα περιλαμβανόμενα αρχεία της C++ γράφονται χωρίς (οποιαδήποτε) κατάληξη, π.χ. γράφουμε:

```
#include <iostream>
```

- Τα περιλαμβανόμενα αρχεία της C γράφονται με πρόθεμα "`c`" (και χωρίς κατάληξη), π.χ.:

```
#include <cmath>
```

  Στην περίπτωση αυτή όλες οι δηλώσεις γίνονται στον ονοματοχώρο *std*, π.χ. στην πραγματικότητα έχουμε *std*`::`*fabs*, αλλά δεν έχουμε πρόβλημα αφού έχουμε βάλει το "`using namespace std`".

- Τα περιλαμβανόμενα αρχεία της C γράφονται με κατάληξη "`.h`" (χωρίς πρόθεμα), π.χ.:

```
#include <math.h>
```

  Αυτή η δυνατότητα είναι προσωρινή, αφού υπάρχουν ήδη πολλά προγράμματα που χρησιμοποιούν αυτή τη σύμβαση. Πάντως το πρότυπο της γλώσσας δίνει οδηγία να μην χρησιμοποιείται αυτό ο τρόπος.

1.11 Έλεγχος Εκτύπωσης

Ας δούμε τώρα πώς μπορούμε να ρυθμίσουμε την εμφάνιση των αποτελεσμάτων, ώστε να μην τα γράφει η C++ όπως θέλει. Προς το παρόν θα δούμε μερικά σχετικά εργαλεία· αργότερα θα μάθουμε κι άλλα.

Για κάθε τιμή που γράφεις στο *cout* μπορείς να δώσεις και ένα **πλάτος πεδίου** (field width). Δίνοντας:

```
cout.width(w);
```

όπου `w` ακέραιη τιμή > 0, ζητάς η επόμενη τιμή (αριθμητική ή ορμαθός χαρακτήρων) που θα γραφεί στο *cout να καταλάβει τουλάχιστον* `w` *θέσεις* σε μια γραμμή εκτύπωσης. Π.χ. οι εντολές:

```
cout.width( 4 );   cout << "abcdefghij";
cout.width( 4 );   cout << "ab";
cout.width( 6 );   cout << 1.1;
cout.width( 8 );   cout << 11;
cout.width( 10 );  cout << 0.0001 << endl;
```

θα δώσουν:

```
abcdefghij␣␣ab␣␣␣1.1␣␣␣␣␣␣11␣␣␣␣0.0001
```

και ας δούμε γιατί. Ζητούμε πλάτος πεδίου

- τουλάχιστον 4 για τον ορμαθό `"abcdefghij"`, που έχει μήκος 10. Γράφεται σε 10 θέσεις.
- 4 για τον ορμαθό `"ab"`, που έχει μήκος 2. Γράφεται σε 4 θέσεις, από τις οποίες οι δύο πρώτες (στα αριστερά) έχουν κενά.
- 6 για την πραγματική τιμή `1.1`, που χρειάζεται 3 θέσεις. Γράφεται σε 6 θέσεις, από τις οποίες οι 3 πρώτες (στα αριστερά) έχουν κενά.
- 8 για την ακέραιη τιμή `11`, που χρειάζεται 2 θέσεις. Γράφεται σε 8 θέσεις, από τις οποίες οι 6 πρώτες έχουν κενά.
- 10 για την ακέραιη τιμή `0.0001`, που χρειάζεται 6 θέσεις. Γράφεται σε 10 θέσεις, από τις οποίες οι 4 πρώτες έχουν κενά.

Για τις πραγματικές τιμές, αυτό που μας ενδιαφέρει συνήθως είναι η **ακρίβεια** (precision). Αν δώσεις την εντολή:

```
cout.precision( d );
```

όπου `d` θετική ακέραιη τιμή, ζητάς οι επόμενες πραγματικές τιμές να τυπώνονται με *d* το πολύ σημαντικά ψηφία. Π.χ. οι

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1234.5  123456.78  1.2345679e+08
```

Όπως βλέπεις, όλες οι τιμές γράφονται με 8 το πολύ σημαντικά ψηφία.

Ο καθορισμός ακρίβειας αναιρείται με νέα `cout.precision`. Η τιμή που καθορίζεται ερήμην είναι: 6.

Κάτι που ενοχλεί συχνά είναι ότι η C++ αποφασίζει με δικά της κριτήρια για το αν, κάποια τιμή, θα γράφεται σε μορφή σταθερής ή κινητής υποδιαστολής. Σου δίνει όμως τη δυνατότητα να το ελέγξεις. Αν δώσεις:

```
cout.setf( ios::scientific, ios::floatfield );
```

όλες οι πραγματικές τιμές που θα γράφεις στη συνέχεια (και μέχρι να δώσεις μια νέα `cout.setf`) θα γράφονται σε παράσταση κινητής υποδιαστολής. Π.χ. οι:

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1.23450000e+03  1.23456780e+05  1.23456789e+08
```

Πρόσεξε τώρα κάτι σε σχέση με την ακρίβεια: Αν έχουμε ακρίβεια *d* ψηφίων για κάθε τιμή θα τυπώνεται:

- το πρόσημο αν είναι "`-`",
- ένα ψηφίο πριν από την υποδιαστολή,
- *d* ψηφία μετά την υποδιαστολή,
- `e±99` ή `e±999` σε τέσσερις ή πέντε θέσεις.

Όπως καταλαβαίνεις, αν έχεις ορίσει ακρίβεια *d* ψηφίων, σου χρειάζονται από *d* + 6 μέχρι *d* + 8 θέσεις. Αν λοιπόν θέλεις να καθορίσεις και το πλάτος πεδίου θα πρέπει να φροντίσεις να είναι μεγαλύτερο.

Αν τώρα δώσεις:

```
cout.setf( ios::fixed, ios::floatfield );
```

οι εντολές:

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1234.50000000  123456.78000000  123456789.01230000
```

Όπως βλέπεις, και στην περίπτωση αυτή, η ακρίβεια (εδώ 8) καθορίζει το πλήθος των ψηφίων που θα γραφούν μετά την υποδιαστολή.

Αν θέλεις να επιστρέψεις στον αυτόματο καθορισμό παράστασης δώσε:

```
cout.setf( 0, ios::floatfield );
```

1.12 Κληρονομιά από τη C: *printf()*

Διαβάζοντας άλλα βιβλία (κυρίως για C αλλά μερικές φορές και για C++) θα δεις να χρησιμοποιείται για την έξοδο αποτελεσμάτων η *printf()*. Θα πούμε μερικά πράγματα για να καταλαβαίνεις αυτά που μπορεί να δεις.

Για να χρησιμοποιήσεις σε ένα πρόγραμμα την *printf()* θα πρέπει να περιλάβεις στο πρόγραμμά σου την `cstdio` αντί για την `iostream`.

Ας ξεκινήσουμε με ένα παράδειγμα χρήσης της *printf*:

```
#include <cstdio>
#include <cmath>

int main()
{
   printf( "%s%f) = %f\n", "τετραγωνική ρίζα(",
           2.0, sqrt(2.0) );
}
```

που δίνει:

```
τετραγωνική ρίζα(2.000000) = 1.414214
```

Όπως η C++ έχει το ρεύμα *cout* προς την οθόνη, η C έχει το ρεύμα "`stdout`". και η *printf()* γράφει σε αυτό. Σε γενικές γραμμές: *ό,τι κάνει ο "`<<`" προς το cout κάνει η printf() προς το stdout*.

Όπως βλέπεις το πρώτο όρισμα της *printf()* είναι ένας ορμαθός χαρακτήρων με «περίεργο περιεχόμενο»· είναι ο **ορμαθός μορφοποίησης** (format string) της *printf*. Στον ορμαθό μορφοποίησης υπάρχουν οι **προδιαγραφές μορφοποίησης** (format specifiers) που αρχίζουν πάντοτε με τον χαρακτήρα "`%`". Ας δούμε τι υπάρχει στο παράδειγμά μας.

- Το "`%s`" λέει: «γράψε όπως ξέρεις έναν ορμαθό χαρακτήρων που θα βρεις στη συνέχεια» (πρόκειται για τον: "`τετραγωνική ρίζα(`").
- Το "`%f`" λέει: «γράψε όπως ξέρεις μια πραγματική τιμή που θα βρεις στη συνέχεια σε μορφή σταθερής υποδιαστολής» (πρόκειται για την: `2.0`).
- Στη συνέχεια θα πρέπει να τυπωθούν όπως είναι τα "`)␣=␣`".
- Το "`%f`" λέει και πάλι: «γράψε όπως ξέρεις μια πραγματική τιμή που θα βρεις στη συνέχεια σε μορφή σταθερής υποδιαστολής» (πρόκειται για την: `sqrt(2.0)`).
- Τέλος, το "`\n`" λέει: «αφού γράψεις όλα τα προηγούμενα, άλλαξε γραμμή».

Γενικώς, για πραγματικές τιμές χρησιμοποιούμε την προδιαγραφή `%f` (σταθερή υποδιαστολή) ή `%e` (κινητή υποδιαστολή), για ακέραιη τιμή προδιαγραφή `%d`, για ορμαθούς χαρακτήρων προδιαγραφή `%s` και για έναν χαρακτήρα μόνον προδιαγραφή `%c` (αλλά και `%d`).

Μετά το "`%`" μπορείς να βάλεις έναν φυσικό αριθμό που καθορίζει το πλάτος πεδίου. Στις προδιαγραφές `%f` και `%e` ένας δεύτερος φυσικός καθορίζει τα ψηφία μετά την υποδιαστολή. Η:

```
printf( "%4s%4s%6.1f%8d%10.4f\n",
        "abcdefghij", "ab", 1.1, 11, 0.0001 );
```

θα δώσει:

```
abcdefghij␣␣ab␣␣␣1.1␣␣␣␣␣␣11␣␣␣␣0.0001
```

ενώ η:

```
printf( "%6.1e  %10.4e\n", 1.1, 0.0001 );
```

δίνει:

```
1.1e+00  1.0000e-04
```

Καλύτερα να χρησιμοποιείς τον μηχανισμό εξόδου της C++, αυτόν με το ρεύμα *cout*. Πάντως απόφυγε να χρησιμοποιείς και τους δύο μηχανισμούς στο ίδιο πρόγραμμα.

1.13 Σχόλια

Η C++ μας επιτρέπει να βάζουμε **σχόλια** (comments) μέσα στο πρόγραμμα που να δίνουν σ' αυτόν που το διαβάζει, διάφορες εξηγήσεις. Τα σχόλια υπάρχουν μόνο στο αρχικό πρόγραμμα και αγνοούνται από το μεταγλωττιστή όταν μεταγλωττίζει το πρόγραμμα.

Για την C++, σχόλιο είναι οτιδήποτε υπάρχει:

- σε μια γραμμή προγράμματος μετά το σύμβολο "`//`", π.χ:

```
cout << "1+1 = " << (1+1) << endl; // τα σχόλια περιττεύουν
// αυτό είναι άλλο ένα σχόλιο
```

- ανάμεσα στα σύμβολα "`/*`" και "`*/`". Π.χ.

```
/* ΚΙ ΑΥΤΟ ΕΙΝΑΙ ΕΝΑ ΣΧΟΛΙΟ */
cout << 3.14159 / 2 /* = π/2 */ << endl;
```

Όπως βλέπεις στο τελευταίο παράδειγμα μπορείς να βάζεις τα σχόλια και μέσα στις εντολές που δίνεις. Αυτό πάντως δεν είναι πάντοτε η καλύτερη ιδέα!

Τα σχόλια που θα βάζεις στα προγράμματά σου θα πρέπει να είναι προσεγμένα, ώστε να εξηγούν ό,τι χρειάζεται εξήγηση και να μη κουράζουν με ανοησίες για προφανή πράγματα.

1.14 Προβλήματα;

Σε βασανίζει ο μεταγλωττιστής όταν γράφεις τα προγράμματά σου; Για να δούμε τι προβλήματα μπορεί να έχεις.

Έστω ότι θέλεις να γράψεις ένα πρόγραμμα που να δίνει την τιμή της παράστασης: $1+\sqrt{2}$. Γράφεις:

```
int main()
{
   cout << (1 + sqrt(2)) << endl
}
```

Καλό δεν είναι; Καλό... Το δίνουμε για μεταγλώττιση και... 1ο λάθος:

```
Undefined symbol 'cout' in function int main()
```

Δηλαδή χρησιμοποιούμε το σύμβολο (όνομα) *cout* χωρίς να το έχουμε ορίσει! Το λάθος μας είναι ότι ξεχάσαμε να βάλουμε τα:

```
#include <iostream>
using namespace std;
```

Το διορθώνουμε και ξαναπροσπαθούμε. 2ο λάθος:

```
Function 'sqrt' should have a prototype in function int main()
```

Αυτή τη φορά το λάθος μας είναι ότι ξεχάσαμε να βάλουμε την οδηγία:

```
#include <cmath>
```

Το διορθώνουμε και προσπαθούμε για τρίτη φορά. 3ο λάθος:

```
Statement missing ; in function int main()
```

Ξεχάσαμε να βάλουμε "`;`" μετά το `endl`!

Το διορθώνουμε και όλα πάνε καλά. Να το πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   cout << (1 + sqrt(2)) << endl;
}
```

Στα επόμενα κεφάλαια θα καταλάβεις τι σημαίνουν όσα είπαμε μέχρι τώρα για τις οδηγίες `include`. Προς το παρόν θα πρέπει να κάνεις τα εξής:

- Αν ένα πρόγραμμά σου έχει εντολές `cout << ...` θα πρέπει να έχει στην αρχή την οδηγία `#include <iostream>`.
- Αν ένα πρόγραμμά σου χρησιμοποιεί οποιαδήποτε από τις αριθμητικές συναρτήσεις του Παραρτ. Β θα πρέπει να έχει στην αρχή την οδηγία `#include <cmath>`.

Ας δούμε τώρα ένα λάθος που κάνουν οι πρωτάρηδες. Για το παραπάνω πρόβλημα γράφεις:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   1 + sqrt(2);
}
```

Δίνεις το πρόγραμμα στο μεταγλωττιστή, όλα πάνε καλά, αλλά κανένα αποτέλεσμα· η οθόνη κενή! Τι γίνεται εδώ; Ζητήσαμε να υπολογιστεί η τιμή της παράστασης αλλά *δεν ζητήσαμε να γραφεί*! Θα έπρεπε να είχαμε γράψει:

```
cout << (1 + sqrt(2))...
```

Ασκήσεις

Α Ομάδα

1-1 Ποια από τα παρακάτω είναι πραγματικές σταθερές:

```
1,0
56.000
-234a
89.5
1e1
+1.01e+01
5.e-1
```

1-2 Ποια από τα παρακάτω είναι ακέραιες σταθερές:

```
1.0
1
-1
-1,0
78
999999
67.89
-8989
```

1-3 Ξεχώρισε από τις παρακάτω παραστάσεις, όσες είναι δεκτές ως προς το συντακτικό από τη C++.

Υπολόγισε τις τιμές τους.

```
7*12 + 3
7.*12 + 3
7.0*12 + 3
21 / 5 % 2
12 / 8
1. + 7/3
1.0 - 7 / 3
13 + 4A -12
58E14
7/(12 - 11E-3/7)
17 % 6/3
```

Β Ομάδα

1-4 Γράψε πρόγραμμα που θα υπολογίζει και θα τυπώνει τα: π, π^2, $\sqrt{\pi}$, e, e^2, $\sqrt{e}$ (e: η βάση των φυσικών λογαρίθμων).

1-5 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει τις τιμές των παρακάτω παραστάσεων:

$$\frac{1}{1+\sqrt{5}} \qquad 1+e^{1/2} \qquad \eta\mu(60^o)-\sqrt{2}$$

(e: η βάση των φυσικών λογαρίθμων).

1-6 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει τις τιμές των παραστάσεων:

$$\frac{\sqrt{2+\sqrt{3}}}{2} \qquad ln\left|\frac{1}{1-\sqrt{2}}\right| ln\left|\frac{1}{1+\sqrt{2}}\right| \qquad \frac{\sqrt{2}(\sqrt{3}+1)}{4}$$

($ln(x)$ ο φυσικός (νεπέρειος) λογάριθμος του x).

1-7 Γράψε προγράμματα που υπολογίζουν και τυπώνουν (μαζί με κατάλληλα σχόλια) τα ακόλουθα:

α) Το εμβαδό τριγώνου με βάση 10.5 *cm* και αντίστοιχο ύψος 8.7 *cm*.
β) Τον όγκο κυλίνδρου με περιφέρεια 9.2 *cm* και ύψος 5.3 *cm*.
γ) Την απόσταση δύο σημείων του επιπέδου με συντεταγμένες: (3,2) και (5,8).
δ) Την τιμή του πολυωνύμου x^2-2x+3 για $x = 1,2,3$

1-8 Γράψε πρόγραμμα που θα «ζωγραφίζει» τα αρχικά σου. Τα γράμματα θα πιάνουν πέντε γραμμές και πέντε στήλες το καθένα και θα σχηματίζονται από τον χαρακτήρα "*****" (δες το παράδειγμα της §1.6).

1-9 Στην §1.9 είπαμε ότι: «Αν και τα δύο ορίσματα μιας πράξης είναι ακέραιοι το αποτέλεσμα της πράξης είναι ακέραιος, ενώ αν έστω και ένα είναι πραγματικός τότε το αποτέλεσμα είναι πραγματικός.» Γράφουμε λοιπόν:

```
cout << (4./2) << "  " << (4/2) << endl;
```

και παίρνουμε αποτέλεσμα:

```
2  2
```

Και πού ξέρουμε ότι η πρώτη είναι πραγματική και η δεύτερη ακέραιη; Μπορείς να βρεις έναν τρόπο να πεισθούμε;
Υπόδ.: Δες αυτά που λέμε στην §1.11.

κεφάλαιο

2

Μεταβλητές και Εκχωρήσεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κατανοήσουμε την έννοια της *μεταβλητής* και μερικούς βασικούς τρόπους χειρισμού και χρήσης στο πρόγραμμα.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράψεις προγραμματάκια που θα έχουν περισσότερες δυνατότητες λόγω της χρήσης μεταβλητών.

Έννοιες κλειδιά:

- *μεταβλητή*
- *όνομα, τιμή* και *τύπος μεταβλητής*
- *μέγεθος και διεύθυνση μεταβλητής*
- *δήλωση μεταβλητής*
- *εντολή εκχώρησης*
- *εντολή εισόδου*
- *cin*
- *αριθμητικοί τύποι*
- *τυποθεώρηση*
- *σταθερές με όνομα*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Η Ανάγκη για Μεταβλητές:
Ωραία και απλά τα προγραμματάκια με τις σταθερές αλλά... δεν μας βοηθάνε και πολύ!

- Ας πούμε ότι γράφουμε εντολές για υπολογισμό μιας παράστασης· μετά θέλουμε να ξαναϋπολογίσουμε την τιμή της παράστασης αφού αλλάξουμε δύο τιμές. Πρέπει να ξαναγράψουμε τις εντολές. Δεν θα ήταν καλύτερο να γράψουμε την παράσταση χρησιμοποιώντας *μεταβλητές* –δηλαδή αντικείμενα που μπορούμε να αλλάξουμε την τιμή τους– αντί για σταθερές;
- Αν ο υπολογισμός είναι μεγάλος θέλουμε να τον κάνουμε κατά τμήματα και να φυλάγουμε κάπου –σε *μεταβλητές*– τα ενδιάμεσα αποτελέσματα.
- Αν όταν γράφουμε το πρόγραμμα δεν ξέρουμε ορισμένες τιμές καλό θα ήταν να γράψουμε τις εντολές υπολογισμού με βάση κάποιες *μεταβλητές* που οι τιμές τους θα ορίζονται (στοιχεία εισόδου) κατά τη διάρκεια της εκτέλεσης.

Χρειαζόμαστε λοιπόν μια οντότητα, ας την πούμε **μεταβλητή** (variable), που η *τιμή* της μπορεί να αλλάζει.

Σε μια διαδικασιακή γλώσσα προγραμματισμού υψηλού επιπέδου (όπως η C++, η C, η C#, η Pascal, η Algol, η Fortran κλπ) *μια μεταβλητή υλοποιείται με μια θέση της μνήμης, που το περιεχόμενό της, η* **τιμή** (value) *της μεταβλητής, μπορεί να αλλάζει με την εκτέλεση του προγράμματος*. Μέσα στο πρόγραμμά μας χειριζόμαστε τη μεταβλητή με το **όνομά** της (name). Ο τρόπος που οργανώνεται η μνήμη για να φιλοξενήσει την τιμή της μεταβλητής καθορίζεται από τον **τύπο** της (type). Ο τύπος έχει σχέση με το τι είδος τιμών μπορεί να φιλοξενήσει: ακέραιους, πραγματικούς, κείμενα κλπ.

Το *όνομα*, η *τιμή* και ο *τύπος* είναι τα τρία *χαρακτηριστικά υψηλού επιπέδου* μιας μεταβλητής. Υπάρχουν δύο ακόμη *χαρακτηριστικά χαμηλού επιπέδου*:

- Το **μέγεθός** της (size), δηλαδή ο αριθμός *ψηφιολέξεων* (bytes) που απαιτούνται για την αποθήκευσή της.
- Η θέση ή η **διεύθυνσή** της (address) στη μνήμη.

Κατ' αρχήν δεν μας ενδιαφέρει πόσες ψηφιολέξεις απαρτίζουν μια μεταβλητή ούτε σε ποια διεύθυνση της μνήμης υλοποιείται. Δυστυχώς, αργότερα, αυτό θα αποδειχθεί ψέμα!

2.1 Μεταβλητές και Τύποι

Υπάρχουν γλώσσες (π.χ. Fortran) που δεν απαιτούν από τον προγραμματιστή να δηλώνει όλες τις μεταβλητές που θα χρησιμοποιήσει. Η C++ απαιτεί να έχουμε *δηλώσει* μια μεταβλητή πριν τη χρησιμοποιήσουμε. Η C (όπως και η Pascal, η Ada κλπ) απαιτεί να δηλώνουμε τις μεταβλητές μας στην αρχή του προγράμματός μας. Αυτή είναι μια καλή συνήθεια για αρχάριους προγραμματιστές και προς το παρόν θα την ακολουθούμε.

♦ ***Με τη δήλωση μιας μεταβλητής καθορίζουμε τα τρία χαρακτηριστικά υψηλού επιπέδου μιας μεταβλητής. Μπορεί όμως να μην ορίσουμε την (αρχική) τιμή.***

Μια **δήλωση μεταβλητών** (variable declaration) αποτελείται από:

- το όνομα ενός *τύπου στοιχείων*,
- μια λίστα ονομάτων μεταβλητών,
- τον χαρακτήρα `';'`.

Παραδείγματα

Τα παρακάτω είναι δηλώσεις μεταβλητών:

```
double  pi, e, distance, length;
int  i, counter, j, number, integer;
```

Τα ίδια θα μπορούσαν να δηλωθούν και ως εξής:

```
int  i;
double  pi, e;
int  metrntns, j, number, integer;
double  distance, length;
```

Τα `int` και `double`, που βλέπεις στο παραπάνω παράδειγμα είναι ονόματα τύπων της C++. Τι είναι ένας τύπος;

- *Είναι ένα σύνολο τιμών*, μαζί με
- τις πράξεις που μπορούμε να κάνουμε σε αυτές τις τιμές.

Όταν λοιπόν δηλώνουμε, για παράδειγμα, "`int i`" εννοούμε ότι η τιμή της μεταβλητής *i* θα ανήκει *πάντοτε* στο `int`. Δηλαδή, οι μεταβλητές, με τη δήλωσή τους, αποκτούν μια ιδιότητα που παραμένει *αναλλοίωτη* όσο εκτελείται το πρόγραμμα. Κάθε τύπος έχει ένα δικό του τρόπο οργάνωσης της μνήμης. Δηλαδή, πόση μνήμη χρειάζεται για την παράσταση μιας τιμής και πώς ερμηνεύονται οι τιμές των δυαδικών ψηφίων που υπάρχουν εκεί.

Σημείωση: ▶

Στον τύπο `int` έχουμε ακέραιες τιμές ενώ στον τύπο `double` έχουμε πραγματικές. Στη συνέχεια τα λέμε εν εκτάσει.◀

Η C++ σου επιτρέπει μαζί με τη δήλωση να δώσεις στη μεταβλητή σου και αρχική τιμή:

Παραδείγματα

Θα μπορούσαμε να δώσουμε στις δηλώσεις μας αρχικές τιμές ως εξής:

```
double  pi, e, distance( 0.0 ), length( 1.0 );
int  i, counter( 0 ), j, number( 0 ), integer( 375 );
```

ή ως εξής:

```
double  pi, e, distance = 0.0, length = 1.0;
int  i, counter = 0, j, number = 0, integer = 375;
```

Από την άποψη χρήσης μνήμης. μπορούμε να πούμε ότι με τη δήλωση των μεταβλητών κάνουμε –πάντοτε– δύο πράγματα:

- ζητούμε μνήμη που θα χρησιμοποιηθεί από τις μεταβλητές του προγράμματός μας,
- απαιτούμε συγκεκριμένο τρόπο οργάνωσης αυτής της μνήμης –τον τρόπο που αντιστοιχεί στον συγκεκριμένο τύπο.

Αν θέλουμε, έχουμε τη δυνατότητα να

- δίνουμε αρχικές τιμές στις μεταβλητές που δηλώνουμε.

Τις δηλώσεις μεταβλητών επεξεργάζεται ο μεταγλωττιστής. Χωρίς να βρίσκεσαι πολύ μακριά από την πραγματικότητα, σκέψου αυτήν την επεξεργασία ως εξής: κατά τη διάρκεια της μεταγλώττισης, κάνει έναν πίνακα όπου βάζει, για κάθε μεταβλητή, την διεύθυνση στην μνήμη και τον τύπο της –δηλαδή, πώς είναι οργανωμένη αυτή η θέση. Ο πίνακας του Σχ. 2-1, θα μπορούσε να αντιστοιχεί στις δηλώσεις του προηγούμενου παραδείγματος. Αυτόν τον πίνακα χρησιμοποιεί ο ΗΥ στην διάρκεια της εκτέλεσης του προγράμματος για να ξέρει πού θα βρεί την κάθε μεταβλητή και πώς θα ερμηνεύσει τα δυαδικά ψηφία που θα βρεί αποθηκευμένα.

Όταν αρχίζει η εκτέλεση του προγράμματος:

- αν, για κάποια μεταβλητή, έχει καθοριστεί με τη δήλωση της και αρχική τιμή, αυτή αποθηκεύεται στην αντίστοιχη θέση της μνήμης,

όνομα	διεύθυνση	τύπος	μέγεθος	τιμή
counter	1245028	`int`	4	0
distance	1245044	`double`	8	0
e	1245052	`double`	8	`???`
i	1245032	`int`	4	`???`
integer	1245016	`int`	4	375
j	1245024	`int`	4	`???`
length	1245036	`double`	8	1
number	1245020	`int`	4	0
pi	1245060	`double`	8	`???`

Σχ. 2-1 Οι πληροφορίες που χρειάζονται για να ορίζουμε ή να παίρνουμε την τιμή μιας μεταβλητής: *διεύθυνση* και *τύπος*. Στην τελευταία στήλη βλέπεις την τιμή που θα έχουν οι μεταβλητές μας όταν αρχίζει η εκτέλεση του προγράμματος. Όσες πήραν (αρχική) τιμή κατά τη δήλωση έχουν την τιμή αυτή. Οι άλλες είναι αόριστες (τιμή: `???`).

- αν δεν έχει καθοριστεί τιμή κατά τη δήλωση η τιμή της δεν είναι ορισμένη· λέμε ότι η μεταβλητή είναι *αόριστη*.

Ως αρχική τιμή μιας μεταβλητής μπορείς να δώσεις, όχι μόνον μια σταθερά, αλλά και μια παράσταση που, όμως, θα περιέχει σταθερές ή μεταβλητές που η τιμή τους έχει καθοριστεί πιο πριν. Π.χ.:

```
int x( 1 ), y( 2 ), z( x+y );
double q( 5 ), r( 1+sqrt(q) );
```

Στη συνέχεια θα δούμε πώς μπορούμε να δώσουμε τιμή σε μια μεταβλητή στη διάρκεια της εκτέλεσης του προγράμματος.

2.1.1 Ονόματα (Αναγνωριστικά)

Οι γλώσσες προγραμματισμού δίνουν τη δυνατότητα στον προγραμματιστή να δίνει συμβολικά ονόματα στα διάφορα αντικείμενα της γλώσσας που χρησιμοποιεί. Συνήθως τα λέμε **ονόματα** ή **αναγνωριστικά** ή **ταυτότητες** (identifiers).

Όπως ήδη είπαμε:

όνομα μεταβλητής = *όνομα*;

Ένα όνομα της C++ σχηματίζεται από χαρακτήρες της γλώσσας ως εξής:

- Ο πρώτος χαρακτήρας είναι γράμμα του Λατινικού αλφαβήτου ή ο *χαρακτήρας της υπογράμμισης* (**"_"**, underscore).
- Οι υπόλοιποι χαρακτήρες είναι γράμματα του Λατινικού αλφαβήτου ή ψηφία, δηλ. αλφαριθμητικοί χαρακτήρες.

όνομα = *μη ψηφίο* | *όνομα*, *μη ψηφίο* | *όνομα*, *ψηφίο*;
μη ψηφίο = "_" | *γράμμα*;

Ισχύουν ακόμα και οι εξής περιορισμοί:

- Μέσα σε ένα όνομα δεν μπορούμε να έχουμε κενά ή αλλαγές γραμμής.
- Δεν επιτρέπονται ως ονόματα οι **λέξεις-κλειδιά** (keywords) της C++[1].

  Ακόμη: *απόφευγε τη χρήση αναγνωριστικών που περιέχουν διπλή υπογράμμιση* ("__").

[1] Για τις λέξεις-κλειδιά δες το Παράρ. C.

Υπάρχουν ακόμη μερικά ονόματα που η χρήση τους επιτρέπεται κατ' αρχήν αλλά στην πραγματικότητα απογορεύεται: είναι αυτά τα οποία ήδη χρησιμοποιούνται από τη C++, όπως τα: `sqrt`, `cout` κλπ. Έχεις, π.χ., δικαίωμα να δηλώσεις:

```
double sqrt;
```

αλλά, με τον τρόπο αυτό, το όνομα `sqrt` αναφέρεται σε μια μεταβλητή και όχι πια στη συνάρτηση για την τετραγωνική ρίζα[2]. Προς το παρόν λοιπόν, δέξου τον κανόνα:

- ***Δεν επιτρέπεται η χρήση των `main`, `sqrt` και άλλων προκαθορισμένων αναγνωριστικών της γλώσσας παρά μόνο για τις προκαθορισμένες χρήσεις τους.***

Παραδείγματα δεκτών αναγνωριστικών είναι τα εξής:

```
ALFA  OMEGA  _omikron  Nikos  prwto_programma
QWERTY  aSdFgH  program_1
```

Δεν είναι δεκτά όμως τα:

`12thprogram` :	αρχίζει από ψηφίο
`SYNTHESH$` :	περιέχει το χαρακτήρα '`$`'
`int` :	είναι λέξη-κλειδί
`Prwto Progr` :	περιέχει κενό
`MakryAnagnw ristiko` :	περιέχει αλλαγή γραμμής

Όπως λέγαμε και στο προηγούμενο κεφάλαιο, η C++ (όπως και η C) διαφέρει από την πλειοψηφία των γλωσσών προγραμματισμού στο εξής: *ξεχωρίζει τα πεζά από τα κεφαλαία γράμματα*. Έτσι, τα ονόματα: `Paradeigma`, `PARADEIGMA`, `PaRaDeIgMa`, `pARADEIGMa` είναι *διαφορετικά* για τη C++.

Η C++ δεν βάζει όριο για τον αριθμό χαρακτήρων που μπορεί να έχει ένα όνομα. Αλλά, τέτοια όρια μπορεί να βάζουν οι διάφορες υλοποιήσεις.

Ένας άλλος, εύλογος, περιορισμός είναι ο εξής: *κάθε όνομα αναφέρεται σε ένα μόνον αντικείμενο του προγράμματός σου*. Δεν μπορείς, για παράδειγμα, να δίνεις το ίδιο όνομα σε δυο μεταβλητές του προγράμματός σου[3].

Υπακούοντας στους παραπάνω κανόνες εφάρμοσε και τον παρακάτω:

- ***Κάθε όνομα θα πρέπει να έχει σχέση με το αντικείμενο που παριστάνει.***

Έστω π.χ. ότι θέλεις, μέσα σε κάποιο πρόγραμμα να υπολογίσεις την περιφέρεια ενός κύκλου από την ακτίνα του. Αντί να τα παραστήσεις με τα ονόματα *a*, *b* είναι πολύ καλύτερο να χρησιμοποιήσεις τα *perifereia*, *aktina* (φυσικά και τα *C* και *r* είναι μια χαρά για κάποιον που ξέρει λίγη Γεωμετρία).

Αυτός ο κανόνας αποβλέπει στο να κάνει τα προγράμματα **ευανάγνωστα** (readable). Ένα πρόγραμμα πρέπει να είναι καλογραμμένο όχι μόνο για να το διαβάζει εύκολα ένας τρίτος, αλλά και ο ίδιος ο προγραμματιστής που το σύνταξε.

Η περιγραφή του τι κάνει ένα πρόγραμμα και πώς το κάνει, λέγεται **τεκμηρίωση** (documentation) του προγράμματος. Ένα πρόγραμμα που γράφεται σωστά, δεν έχει ανάγκη από πολλές περιγραφές και λέμε ότι είναι **αυτοτεκμηριωμένο** (self-documented). Η τεκμηρίωση του προγράμματος είναι ένα σημαντικό κεφάλαιο του προγραμματισμού και θα αναφερόμαστε σ' αυτό ξανά και ξανά. Η σωστή επιλογή των αναγνωριστικών, είναι βασική αρχή αυτού του κεφαλαίου.

---

[2] Όπως θα δούμε, μπορείς να «ξαναβρείς» την τετραγωνική ρίζα ως "`std::sqrt`".

[3] Αυτός ο κανόνας θα ανατραπεί (μερικώς) και θα αντικατασταθεί από έναν πιο ακριβή, αργότερα.

2.2 Εκχώρηση – Μεταβλητές στις Παραστάσεις

Αφού είδαμε πώς δηλώνουμε τις μεταβλητές μας ας δούμε τώρα τι κάνουμε με αυτές στο πρόγραμμά μας. Αλλά πρώτα να υπενθυμίσουμε ότι:

♦ ***Για να χρησιμοποιηθεί κάποια μεταβλητή θα πρέπει να έχει δηλωθεί προηγουμένως.***

Αυτό είναι που απαιτεί η C++. Εμείς, όπως είπαμε, θα κάνουμε κάτι που απαιτείται ή συνηθίζεται στις περισσότερες γλώσσες προγραμματισμού: *θα βάζουμε τις δηλώσεις στην αρχή του προγράμματος*.

Οι δουλειές που κάνουμε με μια μεταβλητή είναι:

- αποθήκευση κάποιας πληροφορίας, δηλαδή: καθορισμός της τιμής μιας μεταβλητής,
- ανάκτηση και χρησιμοποίηση της αποθηκευμένης πληροφορίας, δηλ.: χρήση της μεταβλητής.

Ας ξεκινήσουμε με την πρώτη δουλειά και με ένα παράδειγμα: Θέλουμε να φυλάξουμε την τιμή της παράστασης $1+\sqrt{5}$ για να τη χρησιμοποιήσουμε στη συνέχεια. Αν έχουμε δηλώσει:

```
double x;
```

μπορούμε να δώσουμε την εντολή:

```
x = 1 + sqrt( 5 );
```

Τι σημαίνει;

- Υπολόγισε την τιμή της παράστασης: `1 + sqrt(5)`.
- Μετά εκχώρησε την τιμή στη μεταβλητή *x* (ή αλλιώς αποθήκευσε αυτήν την τιμή στη θέση της μνήμης που έχεις ονομάσει *x*).

Ένας άλλος τρόπος να το δούμε είναι ο εξής: Από εδώ και πέρα, όπου βρίσκεις το όνομα της μεταβλητής *x* θα το *αντικαθιστάς* με την τιμή που έχει η παράσταση `1 + sqrt(5)`. Γι' αυτό θα δεις ότι η εντολή αυτή *λέγεται* και **εντολή αντικατάστασης**.

Θα ονομάζουμε εντολές τέτοιου είδους **εντολές εκχώρησης** (assignment statements) και ας δούμε τα συντακτικά και τα νοηματικά τους. Προς το παρόν, μια εντολή εκχώρησης θα έχει την εξής μορφή:

όνομα μεταβλητής "=" *παράσταση*

όπου "=" είναι το σύμβολο (ή ο *τελεστής*) της εκχώρησης[4]. Η εκτέλεσή της γίνεται ως εξής:

- Υπολογίζεται η τιμή της παράστασης.
- Η τιμή μετατρέπεται στον τύπο της μεταβλητής.
- Η τιμή φυλάγεται ως τιμή της μεταβλητής.

Παραδείγματα

Οι παρακάτω είναι εντολές εκχώρησης:

```
pi = 4*atan(1);
e = exp(1);
number = 4*10 + 2*30;
integer = (100 % 51)*4 + 7;
```

Οι παρακάτω *δεν είναι εντολές εκχώρησης*:

```
x + 1 = 7;        x + y = 12;        sqrt(x) = 12;
```

(αριστερά από το "=" θα έπρεπε να υπάρχει μεταβλητή).

Ας δούμε τώρα πώς χρησιμοποιούμε μια τιμή που έχουμε αποθηκεύσει στη μνήμη: ποια είναι η θέση της μεταβλητής μέσα σε μια παράσταση; *Οι μεταβλητές μπαίνουν στην*

[4] Γιατί «θα ονομάζουμε»; Δεν είναι εντολή εκχώρησης; Η C++ δεν έχει εντολή εκχώρησης! Και αριστερά του "=" μπορεί να υπάρχει ολόκληρη παράσταση! Αργότερα θα καταλάβεις...

παράσταση όπως ακριβώς οι σταθερές –ως πρωτογενείς παραστάσεις. Έτσι, τα παρακάτω είναι παραστάσεις:

```
integer + length*3        i + j/2
sqrt(pi) + 1.0
log(e+1) - 1/pow(e,2)
```

Στον υπολογισμό της παράστασης κάθε μεταβλητή αντικαθίσταται από την τιμή που έχει εκείνη τη στιγμή. Δηλαδή, ο ΗΥ παίρνει το περιεχόμενο της αντίστοιχης θέσης της μνήμης χωρίς όμως και να το αλλάζει. Αν η παράσταση είναι μέσα σε μια "**cout <<** …", τότε η τιμή της παράστασης τυπώνεται συμφώνως με αυτά που ξέρουμε. Να λοιπόν ένα απλό προγραμματάκι:

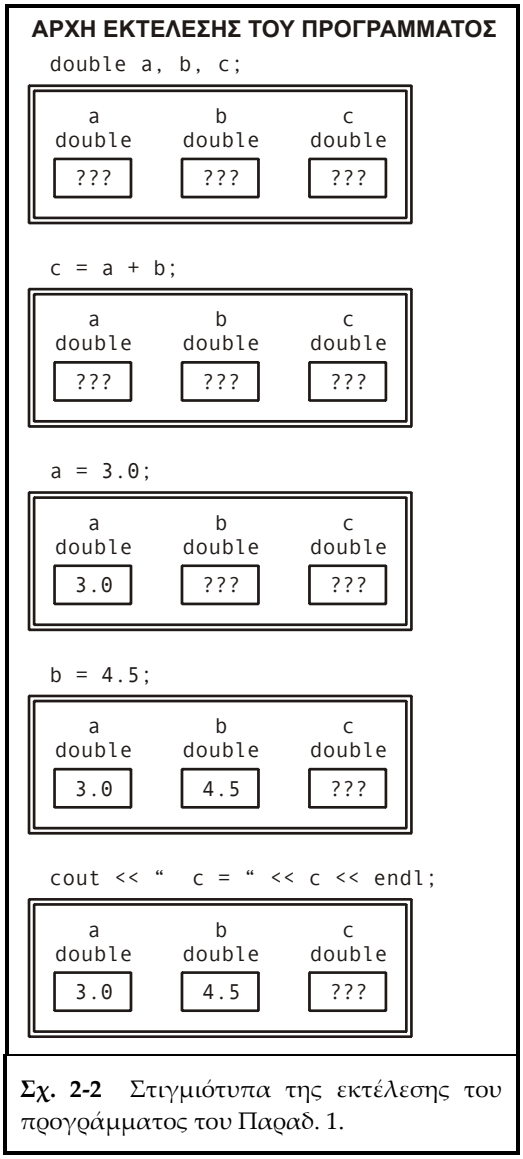
```
#include <iostream>
using namespace std;
int main()
{
   int k;

   k = 3;
   cout << " k = " << k << endl;
}
```

που δίνει:

```
 k = 3
```

Στην αρχή δηλώσαμε μόνο μια μεταβλητή τύπου **int** με το όνομα *k*. Με την πρώτη εντολή δίνουμε στη μεταβλητή *k* την τιμή 3. Στη **cout <<**… έχουμε δυο ορίσματα: τον ορμαθό **" k = "** και την παράσταση **k**. Από το πρώτο όρισμα τυπώνεται το «κείμενο» "␣k␣=␣" και από την παράσταση η τιμή της μεταβλητής *k*.

Ας δούμε δύο παραδείγματα, πολύ χαρακτηριστικά για την εντολή εκχώρησης:



Σχ. 2-2 Στιγμιότυπα της εκτέλεσης του προγράμματος του Παραδ. 1.

Παράδειγμα 1

Στα μαθηματικά μπορούμε να γράφουμε: $c = a + b$ και όταν πιο κάτω δώσουμε $a = 3$, $b = 4.5$, να είναι φανερό ότι $c = 7.5$. Αλλά, τι αποτέλεσμα θα δώσει το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   double a, b, c;

   c = a + b;
   a = 3;  b = 4.5;
   cout << " c = " << c << endl;
}
```

Αποτέλεσμα:

```
  c = 4.24613e-314
```

Τι είναι αυτό;! Για να δούμε: Όταν αρχίζει η εκτέλεση του προγράμματος, οι τιμές των *a*, *b*, *c* δεν είναι ορισμένες. *Αυτό δεν σημαίνει ότι είναι* 0· απλώς έχουν τυχαίες τιμές, που βγαίνουν από τις τυχαίες καταστάσεις στις οποίες έτυχε να βρεθούν τα αντίστοιχα δυαδικά ψηφία της μνήμης. Πρώτη εντολή του προγράμματος είναι η:

```
  c = a + b;
```

Και πώς εκτελείται;

Παρακολούθησε την εκτέλεση στο Σχ. 2-2. Ο ΗΥ «πηγαίνει» στις θέσεις της μνήμης *a*, *b*, παίρνει αυτά (τα «σκουπίδια» ή «**???**» όπως τα παριστάνουμε) που έχουν μέσα, τα προσθέτει και αποθηκεύει το αποτέλεσμα στην *c*. Οι εντολές "**a = 3; b = 4.5**" εκτελούνται *μετά* την "**c = a + b**" και δεν την επηρεάζουν.

Πάντως ο μεταλωττιστής έδωσε προειδοποίηση (warning): «**possible use of 'a' before definition in function main()**» (δυνατή χρήση της "a" πριν από τον ορισμό της στη συνάρτηση **main()**, παρόμοια προειδοποίηση και για τη 'b')

Τα διδάγματα από τα παραπάνω είναι τα εξής:

- Ο υπολογισμός μιας παράστασης γίνεται με τις τιμές που έχουν οι μεταβλητές όταν γίνεται ο υπολογισμός. *Οι κατοπινές αλλαγές τιμών στις μεταβλητές δεν επηρεάζουν υπολογισμούς που έγιναν.*
- Μη χρησιμοποιείς μεταβλητές που δεν έχεις ορίσει τις τιμές τους.

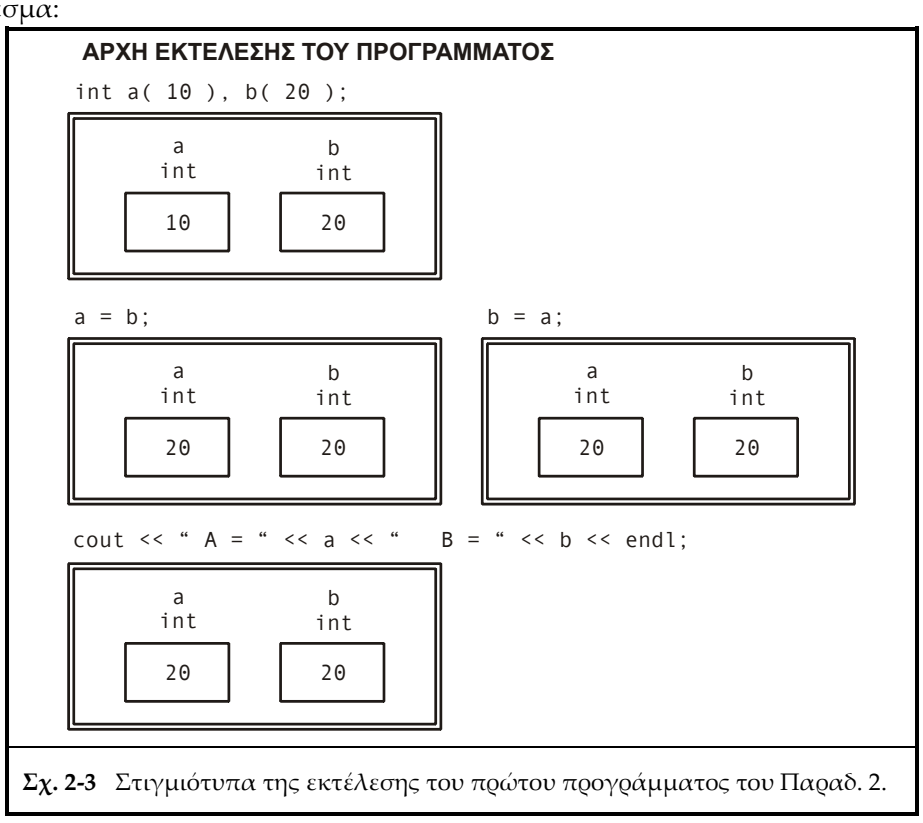
Παράδειγμα 2

Κάτι που θα χρειάζεται να κάνεις αρκετά συχνά στα προγράμματά σου, είναι να *αντιμεταθέτεις τις τιμές δύο μεταβλητών*. Ας δούμε πώς γίνεται κάτι τέτοιο.

Ας πούμε ότι θέλουμε να αντιμεταθέσουμε τις τιμές δύο μεταβλητών *a* και *b*. Η πρώτη ιδέα είναι: βάλε όπου *a* το *b* και όπου *b* το *a*. Ας δούμε τι θα βγάλει το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int a( 10 ), b( 20 );

   a = b;  b = a;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Αποτέλεσμα:



Σχ. 2-3 Στιγμιότυπα της εκτέλεσης του πρώτου προγράμματος του Παραδ. 2.

```
 a = 20   b = 20
```

Ατυχία! Τι έγινε; Όταν εκτελέστηκε η εντολή `a = b`, η *a* πήρε την τιμή 20, που είχε η *b*. Στη συνέχεια εκτελέστηκε η εντολή `b = a` και η *b* (ξανα)πήρε την τιμή 20 από την *a*. Δες και το Σχ. 2-3 που τα δείχνει με πιο παραστατικό τρόπο.

Τι πρέπει να κάνουμε; Να «φυλάξουμε» κάπου την τιμή της *a*, πριν εκτελεστεί η "`a = b`" και από κει να δώσουμε μετά την τιμή στη *b*. Για τη φύλαξη θα χρειαστούμε μια βοηθητική μεταβλητή *s*, του ίδιου τύπου με τις *a*, *b*. Να το σωστό πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int a( 10 ), b( 20 ), s;

   s = a;  a = b;  b = s;
   cout << " a = " << a << "   b = " << b << endl;
}
```

που μας δίνει:

```
 a = 20   b = 10
```

Το δίδαγμα και από εδώ είναι το εξής:

- ***Ο υπολογισμός μιας παράστασης γίνεται με τις τιμές που έχουν οι μεταβλητές όταν γίνεται ο υπολογισμός (η b πήρε την τιμή που είχε η a όταν εκτελέστηκε η εντολή `b = a`). Αν αλλάξεις την τιμή μιας μεταβλητής, η παλιά τιμή χάνεται (εκτός αν τη φυλάξεις εσύ σε κάποια άλλη θέση της μνήμης).***

Και τώρα μια ερώτηση: Γιατί είναι σωστό το πρόγραμμα που γράψαμε; Διότι δούλεψε το παράδειγμα; Ούτε να το σκέφτεσαι!

- ***Με ένα παράδειγμα ή με στιγμιότυπα εκτέλεσης μπορείς να ανακαλύψεις λάθη στο πρόγραμμά σου αλλά δεν μπορείς να αποδείξεις ότι το πρόγραμμα είναι σωστό.***

Στο επόμενο κεφάλαιο θα αποδείξουμε ότι το πρόγραμμα είναι σωστό χρησιμοποιώντας το *αξίωμα της εκχώρησης*.

2.3 Εισαγωγή Στοιχείων

Τα προγράμματα που είδαμε μέχρι τώρα, δουλεύουν με στοιχεία που ορίζονται μέσα σε αυτά και βγάζουν ορισμένα αποτελέσματα. Τέτοια προγράμματα είναι πολύ περιορισμένης χρησιμότητας γιατί δεν μπορούν να αλλάξουν τα στοιχεία με τα οποία γράφτηκαν.

Ένα πρόγραμμα μπορεί να επικοινωνεί με τον χρήστη του, *κατά τη διάρκεια της εκτέλεσής του*, με εντολές εισόδου που έχουν το συντακτικό:

`cin >>` *λίστα ορισμάτων*`;`

Προς το παρόν, η λίστα ορισμάτων θα απαρτίζεται από ονόματα μεταβλητών που διαχωρίζονται από "`>>`".

Παρομοίως με την έξοδο αποτελεσμάτων προς το ρεύμα `cout`, σε κάθε ΗΥ υπάρχει και μια προκαθορισμένη συσκευή –ή αρχείο– από όπου γίνεται είσοδος στοιχείων (default Input device (file))· σε κάθε υλοποίηση της C++, αυτή η συσκευή (αρχείο) συνδέεται με το πρόγραμμά μας με ένα προκαθορισμένο ρεύμα με το όνομα `cin`.

Αν δεν βάζεις την «τεμπέλικη» "`using namespace std`" να ξέρεις ότι θα πρέπει να δηλώνεις:

```
using std::cin;
```

Η "`cin >>` …" διαβάζει από το πληκτρολόγιο μια γραμμή με στοιχεία. Τι είναι μια γραμμή; Αν δουλεύεις σε τερματικό ενός ΗΥ (ή σε έναν μικροϋπολογιστή), μια γραμμή

είναι οι χαρακτήρες που γράφεις ανάμεσα σε δύο διαδοχικά πατήματα του πλήκτρου <enter>. Και τι περιέχει μια γραμμή με στοιχεία;

- Σε κάθε μεταβλητή της λίστας εισόδου πρέπει να αντιστοιχεί μια σταθερά (που μπορεί να έχει και πρόσημο) του ίδιου τύπου στη γραμμή με τα στοιχεία εισόδου.
- Το πρώτο στοιχείο της γραμμής αντιστοιχεί στην πρώτη μεταβλητή της λίστας εισόδου, το δεύτερο στοιχείο στη δεύτερη μεταβλητή κ.ο.κ.
- Το πλήθος των στοιχείων θα πρέπει να είναι τουλάχιστον ίσο με το πλήθος των μεταβλητών της λίστας εισόδου. Αν τα στοιχεία είναι λιγότερα από τις μεταβλητές, ο υπολογιστής θα περιμένει όλα τα στοιχεία, όσα <enter> και αν δώσεις. Αν είναι περισσότερα, όσα πλεονάζουν αγνοούνται από τη συγκεκριμένη εντολή.
- Τα στοιχεία θα πρέπει να διαχωρίζονται μεταξύ τους με ένα τουλάχιστον διάστημα. *Ο αριθμός των διαστημάτων που μεσολαβούν ανάμεσα στους αριθμούς δεν έχει σημασία*. Τα διαστήματα πριν από το πρώτο στοιχείο αγνοούνται.

  Το τι κάνει η εντολή εισόδου φαίνεται στο παρακάτω

Παράδειγμα 1

Γράφουμε το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int    k, j;
   double x, y;

   cin >> k >> j >> x >> y;
   cout << k << ' ' << j << ' '
        << x << ' ' << y << endl;
}
```

Αφού περάσει από το μεταγλωττιστή μας, το δίνουμε για εκτέλεση και βλέπουμε τον οθονοδείκτη (cursor) να περιμένει στην αρχή της πρώτης κενής γραμμής. Τώρα εκτελείται η εντολή `cin >> k >>`…. Ο ΗΥ περιμένει να του πληκτρολογήσουμε 4 τιμές. Πληκτρολογούμε λοιπόν:

```
17  -375      145.78   31e4
```

Στο τέλος πιέζουμε το πλήκτρο <enter>. Ο ΗΥ απαντάει αμέσως:

```
17 -375 145.78 310000
```

Η απάντηση ήρθε από την "`cout << k`…" όπου του ζητάμε να μας γράψει τις τιμές τεσσάρων μεταβλητών διαχωριζόμενες, ανά δύο, από ένα κενό. Τι έγινε όμως με τη "`cin >> k`…"; Ο πρώτος αριθμός που δώσαμε, το "`17`", έγινε τιμή της πρώτης μεταβλητής της λίστας εισόδου, δηλ. της *k*, ο δεύτερος, ο "`-375`", θα γίνει τιμή της *j*, ο τρίτος, ο "`145.78`", της *x* και ο τέταρτος, ο "`31e4`", της *y*, όπως φαίνεται παρακάτω:

```
  cin >> k >> j   >>    x   >>  y;
         ↑    ↑          ↑       ↑
        17  -375      145.78   31e4
```

Το ίδιο αποτέλεσμα παίρνουμε και με τα:

```
␣␣␣␣␣␣␣␣␣␣17␣-375␣␣␣␣␣␣␣␣␣145.78␣31E4
```

Αν όμως πληκτρολογήσουμε:

```
31e4    17  -375      145.78
```

κάνουμε λάθος! Η πρώτη τιμή προορίζεται για την *k*, που είναι τύπου `int` και εμείς δώσαμε "`31e4`"· η τιμή αυτή είναι ακέραιη, αλλά δεν είναι σταθερά τύπου `int`, όπως είδαμε στο προηγούμενο κεφάλαιο, §1.7, 1.8. Να λοιπόν τι διάβασε το πρόγραμμα όταν μεταγλωττίστηκε με τη gcc (Dev C++):

```
31 42 5.28418e-308 9.92631e-315
```

και να τι διάβασε όταν μεταγλωττίστηκε με τη Borland C++ v.5.5:

`31 0 2.12414e-314` (και runtime error)

Παρ' όλα αυτά, *σε μια μεταβλητή τύπου* `double` *μπορεί να αντιστοιχεί μια σταθερά τύπου* `int`.

Και τώρα ας ξαναεκτελέσουμε το πρόγραμμά μας, αλλά θα του δώσουμε ως στοιχεία εισόδου τα εξής:

`17  -375  145.78  31e4  123`

Αποτέλεσμα:

`17 -375 145.78 310000`

Τι έγινε με το "`123`"; Τίποτε απολύτως: το πρόγραμμα περίμενε να διαβάσει τέσσερις τιμές. Τις διάβασε και αγνόησε την πέμπτη! Αν ακολουθούσε άλλη εντολή εισόδου στη συνέχεια, θα ξεκινούσε την ανάγνωση από το "`123`".

Τι θα γίνει όμως αν βάλουμε τρεις τιμές; Ούτε να το σκέφτεσαι. Μπορεί να δίνεις κενά, να αλλάζεις γραμμή με το <enter>, αλλά ο ΗΥ είναι πιο επίμονος από σένα. Και θα επιμένει να πάρει όλες τις τιμές που περιμένει. Δοκίμασέ το!

Βάζοντας εντολές που διαβάζουν τιμές από το πληκτρολόγιο έχουμε τη δυνατότητα να καθορίσουμε (ή να αλλάξουμε) την τιμή μιας μεταβλητής την ώρα που εκτελείται το πρόγραμμα. Με την χρήση της μπορούμε να κάνουμε πιο «ευέλικτα» προγράμματα:

Παράδειγμα 2

Ας γράψουμε ένα προγραμματάκι που «διαβάζει» από το πληκτρολόγιο δύο ακέραιους και υπολογίζει το άθροισμά τους. Τώρα όμως έχουμε διδαχτεί από το προηγούμενο παράδειγμα: Θα βάλουμε το πρόγραμμά μας να δίνει κάποιο μήνυμα όταν θα περιμένει να πληκτρολογήσουμε στοιχεία εισόδου:

```
#include <iostream>
using namespace std;
int main()
{
   int a1, a2, sum;

   cout << "ΔΩΣΕ ΜΟΥ ΤΟΥΣ ΔΥΟ ΠΡΟΣΘΕΤΕΟΥΣ" << endl;
   cin >> a1 >> a2;
   sum = a1 + a2;
   cout << " (" << a1 << ") + (" <<  a2 << ") = "
        << sum << endl;
}
```

Μόλις αρχίσει η εκτέλεση του προγράμματός μας, θα δούμε πάνω στην οθόνη:

```
ΔΩΣΕ ΜΟΥ ΤΟΥΣ ΔΥΟ ΠΡΟΣΘΕΤΕΟΥΣ
_
```

Πληκτρολογούμε τα εξής:

`5  7`

και ο ΗΥ απαντάει:

`(5) + (7) = 12`

Είπαμε ότι το πρόγραμμά αυτό είναι ευέλικτο, γιατί τώρα *μπορούμε να ξαναζητήσουμε την εκτέλεσή του, χωρίς να χρειαστεί νέα μεταγλώττιση*, και αφού πληκτρολογήσουμε:

`57  -17`

να πάρουμε:

`(57) + (-17) = 40`

Όπως καταλαβαίνεις, αν αντί για τη `cin >> a1 >> a2` είχαμε τις: `a1 = 5;  a2 = 7` αυτό δεν θα ήταν δυνατό.

Αυτό ήταν εύκολο. Να δούμε κάτι πιο δύσκολο.

Παράδειγμα 3

Όπως ξέρεις, οι λύσεις της εξίσωσης: $ax^2 + bx + c = 0$ είναι οι:

$$x_{\pm}=\frac{-b\pm\sqrt{\Delta}}{2a} \quad \text{όπου: } \Delta = b^2 - 4ac$$

Το παρακάτω πρόγραμμα θα διαβάζει τα *a*, *b*, *c* και στη συνέχεια θα υπολογίζει και θα τυπώνει τις ρίζες της δευτεροβάθμιας εξίσωσης.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;
   double  x1, x2;
   double  delta;

   cout << "Δώσε τρεις πραγματικούς αριθμούς" << endl;
   cin >> a >> b >> c;
   cout << " a = " << a << "   b = " << b
        << "   c = " << c << endl;
   delta = b*b - 4*a*c;
   x1 = (-b + sqrt(delta))/(2*a);
   x2 = (-b - sqrt(delta))/(2*a);
   cout << " Λύση1 = " << x1
        << "   Λύση2 = " << x2 << endl;
}
```

Αφού τελειώσουμε με τη μεταγλώττιση, ζητούμε την εκτέλεση και βλέπουμε:

```
Δώσε τρεις πραγματικούς αριθμούς
_
```

Πληκτρολογούμε:

```
               1  3  -10
```

και παίρνουμε:

```
 a = 1   b = 3   c = -10
 Λύση1 = 2   Λύση2 = -5
```

Πολύ ωραία! Άλλη μια δοκιμή: Μόλις δούμε το μήνυμα: «Δώσε τρεις πραγματικούς αριθμούς» πληκτρολογούμε:

```
 2   3   4
```

και παίρνουμε (gcc - Dev C++):

```
 a = 2   b = 3   c = 4
 Λύση1 = -1.#IND   Λύση2 = -1.#IND
```

ή (Borland C++ 5.5):

```
 a = 2   b = 3   c = 4

sqrt: DOMAIN error

sqrt: DOMAIN error
 Λύση1 = +NAN   Λύση2 = +NAN
```

Τι είναι αυτό; Καλέσαμε (δύο φορές) τη συνάρτηση *sqrt()* με όρισμα εκτός του πεδίου ορισμού της. Δηλαδή, του ζητήσαμε να υπολογίσει την τετραγωνική ρίζα ενός αρνητικού αριθμού ($\Delta = 3^2 - 4\cdot2\cdot4 = -23$) και ο ΗΥ μας έβγαλε... τα είδες[5].

Σε επόμενο κεφάλαιο θα δούμε πώς, σε τέτοιες περιπτώσεις, μπορείς να παίρνεις τον έλεγχο στα χέρια σου (ή καλύτερα στο πρόγραμμά σου).

[5] Τι είναι το "**#IND**"; Θα τα πούμε αργότερα...

2.4 Σταθερές με Ονόματα

Στα επόμενα παραδείγματα θα χρησιμοποιήσουμε δύο πολύ γνωστές σταθερές: το π (= 3.14159...) και την επιτάχυνση της βαρύτητας κοντά στην επιφάνεια της Γής, *g* (= 9.81 *m/sec*²).

Μπορούμε βέβαια να γράφουμε

```
vP = -sqrt( 2*h*9.81 );
```

ή να απαλλάξουμε τον υπολογιστή από έναν πολλαπλασιασμό(!):

```
vP = -sqrt( h*19.62 );
```

Πόσο εύκολα θα καταλάβει κάποιος που διαβάζει το πρόγραμμα τι είναι αυτή η «μαγική σταθερά» "`9.81`" (και πολύ περισσότερο η "`19.62`"); Εκτός από αυτό, είναι πολύ πιθανό, σε μια διόρθωση ή επέκταση του προγράμματος να χρησιμοποιήσουμε ως τιμή της *g* το "`10`" ή το "`9.8`".

Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνουν τη δυνατότητα να χρησιμοποιήσουμε σταθερές με όνομα:

```
const double g( 9.81 ); // m/sec²
```

και να γράφουμε:

```
vP = -sqrt( 2*h*g );
```

Γιατί χρειαζόμαστε σταθερές με όνομα; Δεν θα μας έκανε μια μεταβλητή; Αντί για άλλη απάντηση, βάζουμε μια εντολή `g = 5.32` και δίνουμε το πρόγραμμά μας για μεταγλώττιση. Αποτέλεσμα: "`Cannot modify a const object in function main()`". Δηλαδή, ο μεταγλωττιστής δεν θα επιτρέψει μια κατά λάθος τροποποίηση τιμής της σταθεράς, πράγμα που δεν μπορεί να συμβαίνει για μια μεταβλητή.

Παρομοίως για το π μπορούμε να ορίσουμε:

```
const double pi( 4*atan(1) ); // = π
```

Εδώ βλέπεις ότι η τιμή της σταθεράς δίνεται από μια (σταθερή) παράσταση!

2.5 Οι Αριθμητικοί Τύποι της C++

Ο τύπος `int` της C++ είναι ένα υποσύνολο των ακεραίων· σε πολλές διαλέκτους της γλώσσας, είναι το σύνολο των ακεραίων αριθμών μεταξύ -2147483648 και 2147483647. Με το παρακάτω προγραμματάκι μπορείς να δεις τη μέγιστη και την ελάχιστη τιμή τύπου `int` στη C++ που χρησιμοποιείς:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
   cout << "INT_MIN = " << INT_MIN << "    INT_MAX = "
        << INT_MAX << endl;
}
```

Στη gcc (Dev C++) και τη Borland C++ θα μας δώσει:

```
INT_MIN = -2147483648    INT_MAX = 2147483647
```

Αν λοιπόν έχεις δηλώσει, όπως είδαμε παραπάνω:

```
   int number;
```

έχουμε τη σιγουριά ότι η *number* έχει πάντοτε ακέραιη τιμή ανάμεσα στη μεγαλύτερη και τη μικρότερη που μπορεί να παρασταθεί στον τύπο `int`.

Εκτός από τον `int` η C++ μας δίνει και άλλους τύπους με ακέραιες τιμές. Στον Πίν. 2-1 βλέπεις τους **ακέραιους** (integral) τύπους. Διαλέγεις και παίρνεις: Θέλεις μεγάλες ακέραιες τιμές; Χρησιμοποίησε τον `long int` και αν θέλεις ακομα μεγαλύτερες διάλεξε τον `long`

Όνομα	Τιμές από .. μέχρι	Δυαδικά ψηφία
`signed char` ή `char`	-128 .. 127	8
`short int` ή `short`	-32768 .. 32767	16
`int`	-2147483648 .. 2147483647	32
`long int` ή `long`	-2147483648 .. 2147483647	32
`long long int`	–9223372036854775808 .. 9223372036854775807	64
`unsigned char`	0 .. 255	8
`unsigned short int`	0 .. 65535	16
`wchar_t`	0 .. 65535	16
`unsigned int`	0 .. 4294967295	32
`unsigned long int`	0 .. 4294967295	32
`unsigned long long int`	0 .. 18446744073709551615	64
`bool`	`false` (0) .. `true` (1)	8

Πίν. 2-1 Ακέραιοι τύποι στη **C++**. Ο `char` σε άλλες υλοποιήσεις είναι ίδιος με τον `signed char` (-128 .. 127) και σε άλλες σαν τον `unsigned char` (0 .. 255). Ο `wchar_t` είναι ίδιος με τον `unsigned short int`. Για τον `bool` θα τα πούμε αργότερα. Οι ακέραιοι `long long int` σε 64 δυαδικά ψηφία υπάρχουν στο C++11.

`long int`.[6] Έχεις μικρές ακέραιες τιμές: ο `short` θα σου λύσει το πρόβλημα και με οικονομία μνήμης (2 ψηφιολέξεις αντί για τις 4 του `long int`). Για πολύ μικρές τιμές έχεις τον `char` που πιάνει μια ψηφιολέξη μόνο![7] Για μη αρνητικές τιμές προτίμησε `unsigned int` ή `unsigned long int` ή `unsigned long long int`.

Εδώ πρόσεξε το εξής: τα μεγέθη εξαρτώνται από την υλοποίηση. Η C++ απαιτεί γενικώς:

`short int` "≤" `int` "≤" `long int` "≤" `long long int`

όπου το "≤" μπορείς να το καταλάβεις ως *μικρότερος ή ίσος χώρος αποθήκευσης* ή *μικρότερη ή ίση μέγιστη τιμή*. Αυτό δεν αποκλείει το να είναι ταυτόσημοι ανά δύο (ή και οι τρεις). Έτσι, στη Borland C++ (και στη gcc) έχουμε `short int` "<" `int` "=" `long int`.

Ο τύπος `double` είναι ένα υποσύνολο των πραγματικών· για την ακρίβεια ένα υποσύνολο των **ρητών**. Αυτό είναι συνέπεια του ότι μια τιμή τύπου `double` αποθηκεύεται στη μνήμη του ΗΥ σε πεπερασμένο πλήθος δυαδικών ψηφίων. Γενικώς, μια πραγματική τιμή, αποθηκεύεται στον ΗΥ με προσέγγιση. Ο τρόπος αποθήκευσης εκτέθηκε πολύ συνοπτικώς στην §1.7.1.

Το παρακάτω πρόγραμμα θα μας δώσει τη μέγιστη και την ελάχιστη θετική τιμή του τύπου `double` (και κάτι ακόμη):

```
#include <iostream>
#include <cfloat>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << DBL_MIN
        << "    DBL_MAX = " << DBL_MAX << endl;
   cout << "DBL_EPSILON = " << DBL_EPSILON << endl;
}
```

Αποτέλεσμα:

```
DBL_MIN = 2.22507e-308    DBL_MAX = 1.79769e+308
DBL_EPSILON = 2.22045e-16
```

[6] Οι ακέραιοι σε 64 δυαδικά ψηφία υπάρχουν στο πρότυπο C++11. Θα τους βρεις στη gcc αλλά όχι στη Borland 5.5.

[7] Για τον `char`, τον `wchar_t` και τον `bool` θα τα πούμε στο Κεφ. 4.

Όνομα	Απόλυτη τιμή από .. μέχρι	Σημαντικά δεκ. ψηφία	Δυαδικά ψηφία
`double`	5.0e–324 .. 1.7e+308	15-16	64
`float`	1.5e–45 .. 3.4e+38	7-8	32
`long double`	1.9e–4951 .. 1.1e+4932	19-20	80

Πίν. 2-2 Οι τύποι κινητής υποδιαστολής της (Borland) C++.

Ένα χαρακτηριστικό ενός τύπου κινητής υποδιαστολής είναι ο ελάχιστος θετικός που αν προστεθεί στο 1 μας δίνει τιμή μεγαλύτερη από 1:

$$\varepsilon = min\{\, u : \mathbb{R} \mid u > 0 \wedge (1 + u)_{fp} \neq 1_{fp} \bullet u \,\}$$

(όπου το q_{fp} συμβολίζει την παράσταση της τιμής q στον `double`). Αυτό είναι το λεγόμενο **έψιλον** (epsilon) του τύπου κινητής υποδιαστολής. Για τον τύπο `double` της Borland C++:

$$\varepsilon = 2.220446e\text{-}16$$

Τι σημαίνει αυτό; Σημαίνει ότι όλες οι τιμές $1 + x$, για οποιοδήποτε $x \in [0,\varepsilon)$, αποθηκεύονται στον τύπο `double` όπως ακριβώς και το 1. Δες τις παρακάτω εντολές

```
cout << "DBL_EPSILON/2 = " << DBL_EPSILON/2 << endl;
cout << 1 + DBL_EPSILON/2 << endl;
```

που μας δίνουν:

```
DBL_EPSILON/2 = 1.11022e-016
1
```

Η δήλωση:

```
double  length;
```

μας εξασφαλίζει ότι η *length* θα έχει πάντοτε τιμή πραγματικό αριθμό από αυτούς που παριστάνονται στον `double`.

Η C++ έχει και άλλους τύπους **πραγματικών** ή **κινητής υποδιαστολής** (floating point) εκτός από τον `double`. Στον Πίν. 2-2 βλέπεις την ελάχιστη και τη μέγιστη απόλυτη τιμή για τον κάθε τύπο καθώς και το πλήθος των σημαντικών ψηφίων.

Όπως βλέπεις η C++ σου δίνει εργαλεία για να ξεπεράσεις –ακριβέστερα: για να *μετατοπίσεις*– τα προβλήματα των αριθμητικών τύπων.

Και ο τύπος μιας μεταβλητής καθορίζεται με τη δήλωσή της. Αν θέλουμε να καθορίσουμε τον τύπο μιας σταθεράς τι κάνουμε; Αυτό γίνεται με μια **κατάληξη** (suffix):

- σταθερά με κατάληξη "`l`" ή "`L`" είναι τύπου `long`,
- σταθερά με κατάληξη "`u`" ή "`U`" είναι τύπου `unsigned`,
- πραγματική σταθερά με κατάληξη "`f`" ή "`F`" είναι τύπου `float`,
- ακέραιη σταθερά χωρίς κατάληξη είναι τύπου `int`, ενώ πραγματική σταθερά χωρίς κατάληξη είναι τύπου `double`.

Παραδείγματα

```
123 (int)  123L (long)
123U (unsigned)  123LU (unsigned long)
1.23 (double)  1.23f (float)  1.23L (long double)
```

Ωραία λοιπόν, έχουμε τόσες επιλογές για τα δεδομένα μας. Και πώς διαλέγουμε τον τύπο; Δεν τα βάζουμε όλα `double` να τελειώνουμε; Όχι! Να γιατί:

- Η παράσταση των τιμών τύπου `double` (και `float` και `long double`) γίνεται *προσεγγιστικώς*, ενώ των τιμών ακέραιου τύπου με *ακρίβεια*.
- Οι πράξεις στους ακέραιους τύπους είναι ακριβείς, ενώ στους πραγματικούς τύπους (όπως ο `double`) δεν είναι, όπως είδες και πιο πάνω.

- Η επεξεργασία (π.χ. πράξεις) στοιχείων ακεραίων τύπων είναι ταχύτερη από αυτή των στοιχείων πραγματικού τύπου.

Αλλά:

- Στον τύπο `double` μπορούμε να παραστήσουμε *κλασματικές* τιμές ενώ στον τύπο `int` *μόνον ακέραιες*.
- Στον τύπο `double` μπορούμε να παραστήσουμε τιμές πολύ μεγαλύτερες, κατ' απόλυτη τιμή, από αυτές που μπορούμε να παραστήσουμε στον τύπο `int`.

Με βάση τα παραπάνω οδηγούμαστε στο συμπέρασμα ότι θα πρέπει να χρησιμοποιούμε ακέραιους τύπους όσο περισσότερο γίνεται. Μπορούμε λοιπόν να δώσουμε έναν πρώτο κανόνα για αποδοτικά προγράμματα:

♦ ***Όταν μια μεταβλητή θα παίρνει σίγουρα ακέραιες τιμές σε όλη την εκτέλεση του προγράμματος, ως τύπος της θα πρέπει να επιλέγεται κάποιος ακέραιος τύπος.***

2.6 Έξω από τα Όρια

Πολλοί οι τύποι λοιπόν, αλλά όλοι έχουν τα όριά τους. Έστω ότι έχουμε δηλώσει:

```
double d;
```

και στη συνέχεια έχουμε:

```
cin >> d;
cout << d << endl;
```

Όταν εκτελείται η `cin >> d` δίνουμε:

```
1e50000
```

Αποτέλεσμα:

```
4.24399e-314      (gcc - Dev C++)
+INF              (Borland C++)
```

Δηλαδή, στην περίπτωση αυτή η κάθε υλοποίηση αντιδρά με δικό της τρόπο.

Παρόμοια συμβαίνουν και με τους άλλους τύπους κινητής υποδιαστολής.

Με τους ακέραιους τύπους τα πράγματα είναι μάλλον χειρότερα. Αν η τιμή που δίνουμε είναι έξω από τα όρια του τύπου αποθηκεύεται κάποια τιμή που υπολογίζεται με αριθμητική υπολοίπων (modulo 2^n).[8]

2.7 Πότε Λύνεται το Πρόβλημα – Δύο Παραδείγματα

Ας σταματήσουμε για λίγο να δίνουμε νέα στοιχεία της γλώσσας κι ας προσπαθήσουμε με ένα παράδειγμα να κάνουμε μια επανάληψη σε όσα μάθαμε μέχρι τώρα.

Το πρόβλημα:

Από ύψος h αφήνεται να πέσει προς τη γή ένα σώμα. Να γραφεί πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του h και θα υπολογίζει και θα γράφει:

α) τον χρόνο που θα κάνει το σώμα μέχρι να φτάσει στην επιφάνεια της γής

β) η ταχύτητά του τη στιγμή της πρόσκρουσης.

Να αγνοηθεί η αντίσταση του αέρα. Επιτάχυνση βαρύτητας: g = 9.81 m/sec².

Αν πούμε *Oz* τον κατακόρυφο άξονα, με αρχή την επιφάνεια της γής και θετική φορά προς τα πάνω, έχουμε:

$$z=\frac{1}{2}at^2+v_0t+z_0$$

για τη θέση και:

[8] Αν αυτό σου λέει κάτι.

$$v = v_0 + at$$

για την ταχύτητα, όπου:

a : η επιτάχυνση,

v_0: η αρχική ταχύτητα,

z_0: η αρχική θέση.

Στην περίπτωσή μας:

$$a = -g \qquad v_0 = 0 \qquad z_0 = h$$

και οι παραπάνω εξισώσεις γίνονται:

$$z = -\frac{1}{2} g t^2 + h$$

για τη θέση και:

$$v = -gt$$

Όταν το σώμα προσκρούσει στην γή θα έχουμε $z = 0$ και τιμή της t θα είναι ακριβώς ο χρόνος πτώσης t_P. Και λύνοντας ως προς t_P την εξίσωση:

$$0 = -\frac{1}{2} g\, t_P^2 + h$$

παίρνουμε τον χρόνο πτώσης:

$$t_P = \sqrt{2h/g}$$

Η ταχύτητα την στιγμή της πρόσκρουσης είναι:

$$v_P = -g\, t_P = -\sqrt{2gh}$$

Αν λοιπόν δηλώσουμε:

```
const double g( 9.81 ); // m/sec²
double h, tP, vP;
```

έχουμε τις προδιαγραφές:

Προϋπόθεση: $(g == 9.81)$ && $(h \geq 0)$

Απαίτηση: $(t_P == \sqrt{2h/g})$ && $(v_P == -\sqrt{2gh})$

Εδώ όμως να τονίσουμε κάτι: Δεν θα πρέπει να σου έχει μείνει αμφιβολία ότι

- ***Όλη η δουλειά για την επίλυση του προβλήματος γίνεται όταν διατυπώνουμε τις προδιαγραφές.***

Το πρόγραμμα που θα γράψουμε από δω και πέρα είναι μόνο για τις πράξεις, που τις αφήνουμε στον υπολογιστή.

Ας έρθουμε τώρα στο πρόγραμμά μας. Το σχέδιο για το πρόγραμμα θα είναι:

```
Διάβασε το h
Υπολόγισε τα tP, vP
Τύπωσε τα tP, vP
```

Υποθέτουμε και πάλι ότι το πρόγραμμα θα εκτελεστεί σε διάλογο με το χρήστη. Γι' αυτό, πριν δοκιμάσει να διαβάσει το h, θα πρέπει να δώσει κατάλληλο μήνυμα προς τον χρήστη. Έτσι, η «*Διάβασε το h*» θα γίνει:

```
cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
```

Η «*Υπολόγισε τα t_P, v_P*» θα γίνει:

```
tP = sqrt(2*h/g);
vP = -sqrt(2*h*g);
```

σύμφωνα με αυτά που είπαμε παραπάνω.

Εδώ όμως πρόσεξε: Ο υπολογισμός του v_P μπορεί να γίνει και με την:

```
vP = -g*tP;
```

Η πρώτη είναι μεν σύμφωνη με αυτά που λένε τα βιβλία, αλλά χρειάζεται δύο πολλαπλασιασμούς και υπολογισμό μιας τετραγωνικής ρίζας. Η δεύτερη χρειάζεται μόνον έναν πολλαπλασιασμό. Θα προτιμήσουμε λοιπόν την «οικονομικότερη» δεύτερη.

Το τρίτο βήμα του σχεδίου «*Τύπωσε τα t_P, v_P*» μεταφράζεται εύκολα στις:

```
   cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << vP << " m/sec" << endl;
```

Να λοιπόν ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
// Υπολόγισε τα tP, vP
// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   tP = sqrt( 2*h/g );
   vP = -sqrt(2*h*g);
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
// Τύπωσε τα tP, vP
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << vP << " m/sec" << endl;
}
```

και ένα παράδειγμα εκτέλεσης. Αρχικώς το πρόγραμμα ζητάει:

```
Δώσε μου το αρχικό ύψος σε m: _
```

Απαντούμε:

```
Δώσε μου το αρχικό ύψος σε m: 80
```

και παίρνουμε:

```
Χρόνος Πτώσης = 4.03855 sec
Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
```

Ας δούμε άλλο ένα

Παράδειγμα

Να γραφεί πρόγραμμα που θα διαβάζει την ακτίνα και το ύψος ενός κυλίνδρου σε *m*, και θα υπολογίζει και θα γράφει:

α) το εμβαδό της βάσης σε m^2,

β) τον όγκο του κυλίνδρου σε *lt*.

Ένα πρόγραμμα για τη δουλειά αυτήν είναι απλό. Ας καταγράψουμε τι θέλουμε να κάνει:

```
Διάβασε την ακτίνα της βάσης και το ύψος.
Υπολόγισε το εμβαδό.
Υπολόγισε τον όγκο και μετάτρεψέ τον σε λίτρα.
Τύπωσε τα αποτελέσματα.
```

Για τον υπολογισμό του εμβαδού μας χρειάζεται το π και καλό είναι να το υπολογίσουμε από την αρχή.

Μια καλή αρχή για τα προγράμματά σου είναι να *αντηχούν* (echo) τα στοιχεία εισόδου μόλις τα διαβάσουν.

Μετά από αυτές τις παρατηρήσεις ξαναγράφουμε το σχέδιό μας:

```
Υπολόγισε το π.
Διάβασε ακτίνα βάσης και ύψος και τύπωσε τις τιμές τους.
Υπολόγισε το εμβαδό.
Υπολόγισε τον όγκο και μετάτρεψέ τον σε λίτρα.
Τύπωσε τα αποτελέσματα.
```

Στο πρόγραμμα θα χρειαστούμε μεταβλητές για:

- το π
- την ακτίνα
- το ύψος
- το εμβαδό
- τον όγκο.

Αν χρησιμοποιήσουμε τα ονόματα: *pi*, *r*, *h*, *s*, *v*, αντιστοίχως, έχουμε κάνει μια καλή επιλογή, μια και αυτά παριστάνουν τέτοια μεγέθη στην απλή γεωμετρία. Θα μας χρειαστεί ακόμη μια μεταβλητή για τον παράγοντα μετατροπής κυβικών μετρων σε λίτρα. Ας την ονομάσουμε *m3SeLt*, και ας της δώσουμε την τιμή 1000 με τη δήλωσή της. Όλες οι μεταβλητές μας θα είναι τύπου `double`.

Να πώς θα μπορούσε να είναι το πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   const double  pi( 4*atan(1) ); // = π
   const double  m3SeLt( 1000 ); // lt/m³
   double  r, h, s, v;
// Διάβασε ακτίνα βάσης και ύψος και τύπωσε τις τιμές τους
   cout << " Δώσε την ακτίνα βάσης σε m: ";   cin >> r;
   cout << " Δώσε το ύψος σε m: ";   cin >> h;
   cout << " Διαστάσεις Κυλίνδρου:";
   cout << " Ακτ.Βάσης = " << r << " m  Ύψος = "
                        << h << " m" << endl;
// Υπολόγισε το εμβαδό
   s = pi*r*r;       // βάση
// Υπολόγισε τον όγκο και μετάτρεψέ τον σε λίτρα
   v = s * h;        // όγκος
   v = m3SeLt * v;  // μετατροπή σε lt
// Τύπωσε τα αποτελέσματα
   cout << " Εμβαδό Βάσης = " << s << " m²" << endl;
   cout << " Όγκος = " << v << " lt" << endl;
}
```

Όταν αρχίσει η εκτέλεσή του, βλέπουμε στην οθόνη μας:

```
Δώσε την ακτίνα βάσης σε μέτρα: _
```

Σε απάντηση πληκτρολογούμε:

```
Δώσε την ακτίνα βάσης σε μέτρα: 1.2
```

και στη συνέχεια βλέπουμε:

```
Δώσε το ύψος σε μέτρα: _
```

απαντούμε:

```
Δώσε το ύψος σε μέτρα: 1.8
```

και παίρνουμε το αποτέλεσμα:

```
Διαστάσεις Κυλίνδρου: Ακτ.Βάσης = 1.2 m  Ύψος = 1.8 m
Εμβαδό Βάσης = 4.52389 m²
Όγκος = 8143.01 lt
```

Δες όμως και ένα άλλο παράδειγμα εκτέλεσης:

```
Δώσε την ακτίνα βάσης σε μέτρα: 1.2  1.8
Δώσε το ύψος σε μέτρα:  Διαστάσεις Κυλίνδρου: Ακτ.Βάσης = 1.2 m  Ύψος = 1.8 m
Εμβαδό Βάσης = 4.52389 m²
Όγκος = 8143.01 lt
```

Εδώ τι έγινε; Όταν μας ζητήθηκε η ακτίνα βάσης πληκτρολογήσαμε εκτός από αυτήν (1.2) και την τιμή του ύψους (1.8). Το 1.8 αγνοήθηκε από την `cin >> r;` αλλά έγινε δεκτό από την `cin >> h;`. Το αποτέλεσμα είναι φανερό: Ενώ υπολογίστηκαν σωστά οι τιμές,

έγινε άνω κάτω η εμφάνιση αφού δεν πήρε το δεύτερο <enter> που θα του δίναμε μαζί με το ύψος.

2.8 Τα Χαρακτηριστικά της Μεταβλητής στη C++

Είδαμε πιο πριν ότι αναφέροντας το όνομα μιας μεταβλητής μπορούμε να πάρουμε την τιμή της για να τη γράψουμε ή για να τη χρησιμοποιήσουμε σε πράξεις (παραστάσεις). Στις προηγούμενες παραγράφους όμως συζητήσαμε και για άλλα χαρακτηριστικά μιας μεταβλητής, όπως η *διεύθυνση*, το *μέγεθος* (πόσες ψηφιολέξεις καταλαμβάνει στη μνήμη) και τον *τύπο*. Η C++ μας δίνει τα εργαλεία για να δούμε και αυτά τα χαρακτηριστικά.

Στις επόμενες υποπαραγράφους θα δούμε αυτά τα εργαλεία της C++ και στο τέλος θα γράψουμε ένα πρόγραμμα που, με βάση τις δηλώσεις (παράδ. στην §2.1):

```
double  pi, e, distance( 0.0 ), length( 1.0 );
int  i, counter( 0 ), j, number( 0 ), integer( 375 );
```

θα μας δώσει τον πίνακα του Σχ. 2.1.

Προς το παρόν, μπορείς να χρησιμοποιήσεις αυτά τα εργαλεία για να «σκαλίζεις» τη μνήμη και να βλέπεις πώς μεταφράζει ο μεταγλωττιστής της C++ τα προγράμματά σου. Αργότερα θα δεις ότι είναι χρήσιμα σε πολλές περιπτώσεις.

2.8.1 Ο Τύπος – Ο Τελεστής "typeid"

Αν δώσεις:

```
cout << typeid(pi).name() << "  "
     << typeid(integer).name() << endl;
```

θα πάρεις[9]:

```
double  int
```

Δηλαδή: ο τύπος της *pi* είναι **double** και ο τύπος της *integer* είναι **int**.

Γενικώς, μπορείς να δώσεις:

typeid(*παράσταση* **).name()**

και να πάρεις τον τύπο του αποτελέσματος της <παράστασης>. Στο πρόγραμμά σου θα πρέπει να περιλάβεις (**#include**) το "**typeinfo**".

Να τώρα *Η λύση της άσκ. 1-8*: Η

```
cout << typeid(4./2).name() << "  "
     << typeid(4/2).name() << endl;
```

μας δίνει την απάντηση:

```
double  int
```

2.8.2 Το Μέγεθος – Ο Τελεστής "sizeof"

Στις τελευταίες στήλες των Πίν. 2-1, 2-2 βλέπεις τη μνήμη, σε ψηφιολέξεις και δυαδικά ψηφία αντίστοιχα, που χρειάζονται οι μεταβλητές διαφόρων τύπων. Δεν θα ήθελες να δεις αυτές τις τιμές για τη δική σου εγκατάσταση της C++; Ο τελεστής "**sizeof**" σου δίνει αυτήν τη δυνατότητα. Γράφουμε:

```
cout << (sizeof number) << "  "
     << (sizeof length) << endl;
```

και παίρνουμε:

```
4  8
```

[9] Το πρόγραμμα από τον gcc (π.χ. Dev C++) θα δώσει: **d i**.

που σημαίνει: το μέγεθος της μεταβλητής *number* είναι 4 ψηφιολέξεις (= 32 δυαδικά ψηφία) και της *length* είναι 8 ψηφιολέξεις.

Γενικώς, μπορείς να γράφεις:

"`sizeof`", *όνομα μεταβλητής* ή

"`sizeof`", "`(`", *παράσταση*, "`)`" ή

"`sizeof`", "`(`", *όνομα τύπου*, "`)`"

Αν δώσεις:

```
cout << (sizeof (4./2)) << "  "
     << (sizeof (4/2)) << endl;
cout << (sizeof (double)) << "  "
     << (sizeof (int)) << endl;
```

θα πάρεις[10]:

```
8  4
8  4
```

2.8.3 Η Διεύθυνση – Οι Τελεστές "&" και "*"

Μπορούμε να δούμε τη διεύθυνση μιας μεταβλητής με τον τελεστή "`&`". Αν δώσεις π.χ.:

```
cout << &number << "  " << number << endl;
```

θα πάρεις απάντηση:

```
0x343f2744  0
```

ή κάτι παρόμοιο. Η δεύτερη τιμή (0) είναι η τιμή της *number*, όπως την περιμένουμε. Η πρώτη είναι μια ακέραιη τιμή –γραμμένη στο *δεκαεξαδικό* σύστημα– και είναι η διεύθυνση, στη μνήμη, όπου αρχίζει η μεταβλητή *number*. Δοκίμασε και στον δικό σου υπολογιστή, αλλά μην περιμένεις να πάρεις την ίδια τιμή!

Έτσι:

- γράφοντας `number` παίρνουμε αυτό που μας ενδιαφέρει, δηλ. την τιμή της μεταβλητής `number`, ενώ
- γράφοντας `&number` παίρνουμε τη διεύθυνση μιας θέσης μνήμης όπου υπάρχει η πληροφορία που θέλουμε· παίρνουμε δηλαδή μια *παραπομπή* προς αυτό που μας ενδιαφέρει.

Λέμε λοιπόν ότι η `&number` είναι μια **παραπέμπουσα** ή **αναφερόμενη** (referencing) τιμή –αφού παραπέμπει ή αναφέρεται σε κάτι– ή τιμή-**βέλος** (pointer) –αφού κατευθύνεται προς (δείχνει) κάτι. Όπως θα δούμε αργότερα, στον προγραμματισμό με τη C++, τα βέλη χρησιμοποιούνται πάρα πολύ.

Ας πούμε ότι έχουμε μια τιμή-βέλος *p*, δηλαδή μια διεύθυνση· πώς μπορούμε να δούμε την τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η *p*; Ή, με άλλα λόγια, ποια είναι η *αντίστροφη πράξη* της "`&`"; Η C++ τη συμβολίζει με "`*`": η τιμή που είναι απόθηκευμενη στη θέση που μας δείχνει η *p* παριστάνεται με "`*p`". Αν, λοιπόν, πάρουμε τη "`*(&number)`" είναι σαν να παίρνουμε τη "`number`". Πράγματι, η:

```
cout << &number << "  " << number
     << "  " << *(&number) << endl;
```

θα δώσει:

```
0x343f2744  0  0
```

Δες και αυτό:

```
*(&number) = 4;
cout << &number << "  " << number
     << "  " << *(&number) << endl;
```

[10] Αυτό μπορείς να το θεωρήσεις και ως λύση στην άσκ. 1-8.

Αποτέλεσμα:

```
0x343f2744  4  4
```

«Μα, αριστερά του "=" δεν υπάρχει μεταβλητή!» Υπάρχει! Αφού είπαμε ότι «*Αν πάρουμε την* `*(&number)` *είναι σαν να παίρνουμε τη* `number`.»

Λέμε ότι ο τελεστής "`*`" **απο-παραπέμπει** (dereferences) την τιμή-βέλος στην οποία δρα.

Εδώ όμως μπορεί να υπάρχουν αντιρρήσεις: «ο τελεστής "`*`" ξέρουμε ότι κάνει πολλαπλασιασμό! Τι αποπαραπομπές και κουραφέξαλα μας λες;» Ναι, *ο "`*`" κάνει πολλαπλασιασμό όταν δρα σε δύο αριθμητικές τιμές· κάνει αποπαραπομπή όταν δρα σε μια τιμή-βέλος.*

2.8.4 Πώς Παίρνουμε τον Πίνακα

Και τώρα ας έρθουμε να γράψουμε το πρόγραμμα που θα βγάζει τον πίνακα του Σχ. 2-1 και μάλιστα με μια στήλη παραπάνω όπου θα γράφεται το μέγεθος σε ψηφιολέξεις. Εδώ θα γράψουμε τις εντολές που βγάζουν τις δύο πρώτες γραμμές και εσύ θα το συμπληρώσεις για να βγάλει τις υπόλοιπες.

Φυσικά, ξεκινούμε με τις δηλώσεις:

```
   double pi, e, distance( 0.0 ), length( 1.0 );
   int    i, counter( 0 ), j, number( 0 ), integer( 375 );
```

Στη αρχή θα πρέπει να τυπώσει το όνομα σε 8 θέσεις και στη συνέχεια τη διεύθυνση:

```
   cout.width( 8 );
   cout << "integer" << "  " << &integer << "  ";
```

Τώρα σειρά έχει ο τύπος, σε 6 θέσεις, και μετά το μέγεθος:

```
   cout.width( 6 );
   cout << typeid(integer).name() << "  "
        << (sizeof integer) << "  ";
```

Τέλος βγάζουμε και την τιμή:

```
   cout << integer << endl;
```

Να λοιπόν το πρόγραμμα:

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   double pi, e, distance( 0.0 ), length( 1.0 );
   int    i, counter( 0 ), j, number( 0 ), integer( 375 );

   cout << " όνομα\tδιεύθυνση\tτύπος\tμεγ\tτιμή" <<endl;
   cout.width( 8 );
   cout << "counter" << '\t' << &counter << '\t';
   cout.width( 6 );
   cout << typeid(counter).name() << '\t'
        << (sizeof counter) << '\t';
   cout << counter << endl;

   cout.width(8);
   cout << "distance" << '\t' << &distance << '\t';
   cout.width(6);
   cout << typeid(distance).name() << '\t'
        << (sizeof distance) << '\t';
   cout << distance << endl;
// εσύ γράφεις τα υπόλοιπα
}
```

που (συμπληρωμένο θα) δίνει:

```
όνομα  διεύθυνση    τύπος  μεγ    τιμή
 counter     0x12FF64        int 4      0
distance     0x12FF74     double 8      0
```

```
       e      0x12FF7C      double 8      2.12414e-314
       i      0x12FF68         int 4      1245072
 integer      0x12FF58         int 4      375
       j      0x12FF60         int 4      1
  length      0x12FF6C      double 8      1
  number      0x12FF5C         int 4      0
      pi      0x12FF84      double 8      2.122e-314
```

Παρατηρήσεις: ▶

- Δεν σου αρέσει ως πίνακας, ε; Πέρνα τον στο Excel ή σε πίνακα του Word και θα γίνει πανέμορφος.
- Οι τιμές των *e*, *i*, *j* και *pi* δεν πρέπει να σε εκπλήσσουν: αυτές οι μεταβλητές είναι αόριστες. Έτσι, ας πούμε για την *e*, ο ΗΥ «βλέπει» τα δυαδικά ψηφία στην αντίστοιχη διεύθυνση και τα «ερμηνεύει» ως τιμή τύπου `double`. ◀

2.9 Αλλαγή Τύπου

Για κάθε τύπο *T* της C++ υπάρχει μια συνάρτηση, με όνομα

`static_cast<`*T*`>`

που μετατρέπει τιμές άλλων τύπων σε τιμή τύπου *T* (αν αυτή η μετατροπή είναι δυνατή). Εδώ θα ασχοληθούμε με μετατροπές αριθμητικών τύπων. Ας δούμε ένα παράδειγμα:

Παράδειγμα

Η εντολή:

```
cout << static_cast<int>(9.916) << "  "
     << static_cast<int>(-9.916) << endl;
```

θα δώσει:

```
9  -9
```

Δηλαδή: οι τιμές "`9.916`" και "`-9.916`" μετατράπηκαν σε τιμές τύπου `int` από τη συνάρτηση `static_cast<int>`. Αυτό έγινε με *αποκοπή του κλασματικού μέρους*.

Η εντολή:

```
cout << static_cast<double>(901234567L) << "  "
     << static_cast<float>(901234567L) << endl;
```

θα δώσει:

```
9.01235e+08  9.01235e+08
```

Δηλαδή: η τιμή `901234567L` μετατράπηκε

- σε μια τιμή τύπου `double` από τη `static_cast<double>` και
- σε μια τιμή τύπου `float` από τη `static_cast<float>`.

Αλλά, η ακέραιη τιμή είχε 9 ψηφία ενώ από τη μετατροπή μας έμειναν 6. Μήπως υπάρχουν τα ψηφία αλλά δεν γράφονται; Ας δοκιμάσουμε να αλλάξουμε την ακρίβεια:

```
cout.precision(10);
cout << static_cast<double>(901234567L) << "  "
     << static_cast<float>(901234567L) << endl;
```

Αποτέλεσμα:

```
901234567  901234560
```

Βλέπουμε δηλαδή ότι *από τη μετατροπή* `long` *σε* `float` *μπορεί να έχουμε απώλεια σημαντικών ψηφίων*.

Να δούμε τώρα μια άλλη περίπτωση: Ας υποθέσουμε ότι έχουμε τύπο `int` με μέγιστη τιμή 2147483647· τι θα γίνει αν ζητήσω την

`static_cast<int>( 2345678901.23 )`

Η C++ μας λέει ότι το αποτέλεσμα είναι αόριστο. Γενικώς: *αν έχεις αριθμητικό τύπο T που περιλαμβάνει τιμές από* min_T *μέχρι* max_T *μπορείς να χρησιμοποιείς την* `static_cast<T>(x)` *μόνον αν*

$$min_T \le \texttt{static\_cast<}T\texttt{>}(x) \le max_T$$

Η αλλαγή παράστασης μιας τιμής από έναν τύπο σε έναν άλλο ονομάζεται (στατική) **τυποθεώρηση** (typecasting ή casting)[11].

Εδώ θα πρέπει να τονίσουμε ότι η τιμή που έχει μια μεταβλητή δεν μεταβάλλεται από τη στατική τυποθεώρησή της. Π.χ. οι εντολές:

```
double x( 123.457 );
int    i;
i = static_cast<int>( x );
cout << "  x = " << x << "    i = " << i << endl;
```

δίνουν:

```
x = 123.457    i = 123
```

Αντί για `i = static_cast<int>( x )` μπορούμε να γράψουμε:

```
i = int( x );
```

και να πάρουμε το ίδιο αποτέλεσμα. Γενικώς, το να γράφουμε:

$T(v)$ αντί για `static_cast<`T`>(`v`)`

είναι πολύ συνηθισμένο αλλά όχι κατ' ανάγκη καλύτερο.

Πρόσεξε τώρα μια (πολύ συχνή) χρήση της τυποθεώρησης: Από όσα ξέρουμε, μετά τη δήλωση:

```
int k( 10 ), n( 3 );
```

η εντολή:

```
cout << "  k/n = " << k/n << endl;
```

θα δώσει:

```
k/n = 3
```

Πώς μπορούμε να πάρουμε το ακριβές αποτέλεσμα; Έτσι:

```
cout << "  k/n = " << static_cast<double>(k)/n << endl;
```

ή έτσι:

```
cout << "  k/n = " << double(k)/n << endl;
```

που δίνουν:

```
k/n = 3.33333
```

2.9.1 Η Τυποθεώρηση στη C

Η C++ έχει κληρονομήσει από τη C και έναν άλλο τρόπο γραφής της τυποθεώρησης· γράφουμε:

$(T)\,v$ που είναι ισοδύναμο[12] με `static_cast<`T`>(`v`)`

Έτσι, μπορεί να δεις γραμμένο

```
(unsigned long int) -1234567890L
```

αντί για:

```
static_cast<unsigned long int>(-1234567890L)
```

---

[11] Υπάρχουν και άλλα είδη τυποθεώρησης, που θα τα δούμε αργότερα.

[12] Ε, όχι ακριβώς· ο συμβολισμός αυτός καλύπτει και άλλες περιπτώσεις τυποθεώρησης, που θα μάθουμε αργότερα.

2.10 * Οι «Συντομογραφίες» της Εκχώρησης

Η C++ μας δίνει μερικές «συντομογραφίες» πολύ συνηθισμένων εκχωρήσεων. Πολύ συχνά στα προγράμματά μας θέλουμε να γράψουμε: αύξησε το *x* κατά *b*· αυτό το γράφουμε, σύμφωνα με όσα ξέρουμε:

```
   x = x + b;
```

δηλαδή: πάρε την τιμή της *x* και την τιμή της *b*, πρόσθεσέ τις και το αποτέλεσμα να το αποθηκεύσεις ως νέα τιμή της *x*. Η C++ μας δίνει τη δυνατότητα να το γράψουμε ως:

```
   x += b;
```

Παρόμοιες συντομογραφίες υπάρχουν και για τις άλλες πράξεις:

`x += Π;` είναι ισοδύναμη με: `x = x + Π;`
`x -= Π;` είναι ισοδύναμη με: `x = x - Π;`
`x *= Π;` είναι ισοδύναμη με: `x = x * Π;`
`x /= Π;` είναι ισοδύναμη με: `x = x / Π;`
`x %= Π;` είναι ισοδύναμη με: `x = x % Π;`

Για τις "`/=`" και "`%=`" η παράσταση *Π* θα πρέπει να παίρνει τιμή μη μηδενική. Για την τελευταία: η *x* και η τιμή της *Π* θα πρέπει να είναι ακέραιου τύπου.

Ειδικώς για την περίπτωση που θέλουμε να αυξήσουμε ή να μειώσουμε την τιμή μιας μεταβλητής κατά 1, υπάρχουν και άλλες συντομογραφίες:

`++x;` είναι ισοδύναμη με: `x = x + 1;` ή `x += 1;`
`--x;` είναι ισοδύναμη με: `x = x - 1;` ή `x -= 1;`

Τέλος, υπάρχει και μια παραλλαγή της τελευταίας συντομογραφίας: οι `x++` και `x--` αυξάνουν/μειώνουν την τιμή της *x* κατά 1. Τη διαφορά τους από τις `++x` και `--x` θα τη μάθουμε αργότερα.[13]

Στο παρακάτω παράδειγμα μπορείς να δεις όλα τα παραπάνω:

```
#include <iostream>
using namespace std;
int main()
{
   int x, y;
   int a = 1, b = 2;

   x = 5; x += a+b; cout << x << "  ";
   x = 5; x -= a+b; cout << x << endl;
   x = 5; x *= a+b; cout << x << endl;
   x = 5; x /= a+b; cout << x << "  ";
   x = 5; x %= a+b; cout << x << endl;

   x = 5;  ++x;   cout << x << "  ";
   x = 5;  --x;   cout << x << endl;
   x = 5;  x++;   cout << x << "  ";
   x = 5;  x--;   cout << x << endl;
}
```

Αποτέλεσμα:

```
8  2
15
1  2
6  4
6  4
```

Θα αρχίσουμε να χρησιμοποιούμε τις συντομογραφίες αυτές στο Β' Μέρος του βιβλίου.

---

[13] Τώρα προσπάθησε να καταλάβεις τη σχέση που έχει η C++ με τη «μητρική» της γλώσσα C!

2.11 * Υπολογισμός Παράστασης

Στον πίνακα του Παρ. Ε βλέπεις τα χαρακτηριστικά των πράξεων της C++. Ας δούμε αυτές που έχουμε μάθει μέχρι τώρα:

- Πρώτα εκτελούνται υπολογισμοί μέσα σε παρενθέσεις (προτ. 0).
- Στη συνέχεια υπολογίζονται οι κλήσεις συναρτήσεων (προτ. 2).
- Μετά γίνονται πράξεις με τα πρόσημα (προτ. 3) με προσεταιριστικότητα από δεξιά προς τα αριστερά. Δηλαδή; Στη C++ μπορείς να γράψεις:

  `+ - - +12` ή `- + - -x`

  που υπολογίζονται ως:

  `+(-(-(+12)))` και `-(+(-(-x)))`

  αντιστοίχως.
- Μετά γίνονται πολλαπλασιασμοί, διαιρέσεις και υπολογισμοί υπολοίπου (προτ. 5) με προσεταιριστικότητα από αριστερά προς τα δεξιά. Αυτό σημαίνει ότι η παράσταση: `2*x/y*z/5` υπολογίζεται ως: `(((2*x)/y)*z)/5`. Αλλά προσοχή: *δεν σημαίνει* ότι αν έχεις την `(x*y)/(w/u)` θα γίνει πρώτα ο πολλαπλασιασμός `x*y` και μετά η διαίρεση `w/u`.
- Στη συνέχεια γίνονται προσθέσεις και αφαιρέσεις (προτ. 6) από αριστερά προς τα δεξιά, με την ίδια έννοια όπως παραπάνω.

Όταν υπολογίζεται μια αριθμητική παράσταση γίνονται και ορισμένες μετατροπές τύπων, που θα τις ονομάζουμε *Συνήθεις Αριθμητικές Μετατροπές* (ΣΑΜ). Ας πούμε ότι έχουμε κάποια πράξη *a θ b*. Τότε:

1. Αν κάποια από τις τιμές *a*, *b* είναι τύπου `long double`, τότε μετατρέπεται στον τύπο `long double` και η άλλη τιμή.
2. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `double`, τότε μετατρέπεται στον τύπο `double` και η άλλη τιμή.
3. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `float`, τότε μετατρέπεται στον τύπο `float` και η άλλη τιμή.
4. Αλλιώς, τα *a*, *b* (είναι ακέραιου τύπου) υφίστανται *προώθηση ακεραίων*, δηλαδή: αν κάποια (ή και οι δύο) από τις τιμές *a*, *b* είναι «μικρού ακέραιου τύπου» (`char`, `short int`, `signed` ή `unsigned`), αυτή μετατρέπεται σε τύπο `int` ή `unsigned int` (αν δεν μπορεί να παρασταθεί στον `int`) και στη συνέχεια

   αν κάποια από τις τιμές *a*, *b* είναι τύπου `unsigned long`, τότε

   μετατρέπεται στον τύπο `unsigned long` και η άλλη τιμή.
5. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `long int` και η άλλη `unsigned int`, τότε αν κάθε τιμή `unsigned int` μπορεί να παρασταθεί σε `long int`

   η τιμή `unsigned int` μετατρέπεται στον τύπο `long int`

   αλλιώς

   και οι δύο μετατρέπονται σε `unsigned long int`.
6. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `long`, τότε μετατρέπεται σε `long` και η άλλη τιμή.
7. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `unsigned`, τότε μετατρέπεται στον τύπο `unsigned` και η άλλη τιμή.
8. Αλλιώς και οι δυο τιμές είναι τύπου `int`.

Οι ΣΑΜ εφαρμόζονται όταν γίνονται οι πράξεις `+`, `-`, `*`, `/` μεταξύ αριθμητικών τιμών και καθορίζουν τον τύπο του αποτελέσματος. Ακέραιη προώθηση γίνεται στις ενικές πράξεις `+`, `-` (πρόσημα). Ο τύπος του αποτελέσματος είναι αυτός που προκύπτει από την προώθηση.

Δηλαδή, ο μεταγλωττιστής κάνει τις κατάλληλες μετατροπές τύπου ώστε να έχει τον ακριβέστερο υπολογισμό. Αλλά, στο τέλος, αυτό μπορεί να «χαλάσει» από κάποια εκχώρηση που μπορεί να ακολουθεί. Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
   short int a;

   a = SHRT_MAX + 10;
   cout << SHRT_MAX+10 << "  " << a << endl;
   cout << sizeof(SHRT_MAX+10) << "  " << sizeof(a) << endl;
}
```

που δίνει:

```
32777  -32759
4  2
```

Η πρώτη τιμή είναι ακριβής αλλά η δεύτερη όχι. Το τι έγινε το καταλαβαίνεις αν πάρεις υπόψη σου και τη δεύτερη γραμμή: Η τιμή `SHRT_MAX + 10` χρειάζεται 4 ψηφιολέξεις για να παρασταθεί αλλά η *a* έχει μόνο 2...

2.12 *scanf()*: Η Δίδυμη της *printf()*

Στη C, όπως γράφουμε με την *printf().*, διαβάζουμε με τη *scanf()*. Η αλλιώς: *ό,τι κάνει ο "`>>`" στο cin κάνει η scanf() στο stdin*, που είναι για τη C το ρεύμα από το πληκτρολόγιο προς το πρόγραμμά μας.

Ξαναγράφουμε το πρόγραμμα του Παραδ. 1 της §2.3 χρησιμοποιώντας τις *scanf()* και *printf()*:

```
#include <cstdio>
using namespace std;
int main()
{
   int    k, j;
   double x, y;

   scanf( "%d %d %lf %lf", &k, &j, &x, &y );
   printf( "%d %d %f %f\n", k, j, x, y );
}
```

Η σύνταξη της *scanf* μοιάζει με αυτήν της *printf* ως προς το ότι και οι δύο παίρνουν πρώτο όρισμα έναν *ορμαθό μορφοποίησης*. Τα υπόλοιπα ορίσματα διαφέρουν ως προς τον τύπο: η *scanf* περιμένει (παραστάσεις που οι τιμές τους είναι) *βέλη*, δηλαδή *διευθύνσεις στη μνήμη*. Σε κάθε προδιαγραφή μορφοποίησης αντιστοιχεί μια διεύθυνση. Πηγαίνοντας από αριστερά προς τα δεξιά βρίσκουμε

- Την προδιαγραφή "`%d`" που αντιστοιχεί στην πρώτη διεύθυνση "`&k`". Αυτό σημαίνει: θα διαβάσεις ψηφία που σχηματίζουν μια *ακέραιη* σταθερά· θα αποθηκεύσεις την τιμή (τύπου `int`) που θα προκύψει από τη μετατροπή σε εσωτερική παράσταση στη διεύθυνση `&`*k*.
- Παρόμοια ισχύουν για τη δεύτερη "`%d`" που αντιστοιχεί στη δεύτερη διεύθυνση "`&j`".
- Η πρώτη προδιαγραφή "`%lf`" που αντιστοιχεί στην τρίτη διεύθυνση "`&x`" και σημαίνει: θα διαβάσεις ψηφία που σχηματίζουν μια *πραγματική* σταθερά· θα αποθηκεύσεις την τιμή τύπου `double` που θα προκύψει από τη μετατροπή σε εσωτερική παράσταση στη διεύθυνση `&`*x*.
- Παρόμοια ισχύουν και για τη δεύτερη "`%lf`" που αντιστοιχεί στην τέταρτη διεύθυνση "`&y`".

Οι προδιαγραφές "**%d**" είναι γενικώς για ακέραιες τιμές. Μπροστά από το '**d**' μπορεί να υπάρχει ένας **τροποποιητής** (modifier) '**h**' αν θέλουμε η εσωτερική παράσταση να είναι **short int** ή '**l**' για να είναι **long int** (χωρίς τροποποιητή για **int**).[14]

Για πραγματικές τιμές έχουμε τις προδιαγραφές "**%f**" (ή "**%e**" ή "**%E**" ή "**%f**" ή "**%g**" ή "**%G**"). Μπροστά από το '**f**' μπορεί να υπάρχει τροποποιητής '**l**' αν θέλουμε η εσωτερική παράσταση να είναι **double** ή '**L**' για να είναι **long double** (χωρίς τροποποιητή για **float**).

Παρατήρηση: ▶

Το παραπάνω πρόγραμμα δεν είναι γραμμένο σε C, αλλά πρόγραμμα C++ που χρησιμοποιεί βιβλιοθήκες της C. Ο μεταγλωττιστής της C *δεν* θα το δεχτεί.

Να τι θα δεχθεί:

```
#include <stdio.h>

int main()
{
   int    k, j;
   double x, y;

   scanf( "%d %d %lf %lf", &k, &j, &x, &y );
   printf( "%d %d %f %f\n", k, j, x, y );
}
```

Φύλαξέ το στο αρχείο **test_C_IO.c** και δώσε το στον μεταγλωττιστή της C. Δεν θα σου φέρει αντίρρηση.◀

Ας ξαναγράφουμε και το πρόγραμμα του Παραδ. 3 της §2.3 με τις *scanf* και *printf*:

```
#include <cstdio>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;
   double  x1, x2;
   double  delta;

   printf( "Δώσε τρεις πραγματικούς αριθμούς\n" );
   scanf( "%lf %lf %lf", &a, &b, &c );
   printf( " a = %f   b = %f   c = %f\n", a, b, c );
   delta = b*b - 4*a*c;
   x1 = (-b + sqrt(delta))/(2*a);
   x2 = (-b - sqrt(delta))/(2*a);
   printf( " Λύση1 = %f   Λύση2 = %f\n", x1, x2 );
}
```

Πρόσεξε ότι εδώ προτιμήσαμε να βάλουμε όλα τα σταθερά κείμενα στον ορμαθό μορφόποιησης. Δηλαδή:

- Αντι να γράψουμε

```
   printf( "%s\n", "Δώσε τρεις πραγματικούς αριθμούς" );
```

γράφουμε

```
   printf( "Δώσε τρεις πραγματικούς αριθμούς\n" );
```

- Αντι να γράψουμε

```
   printf( "%s %f %s %f %s %f\n",
           " a =", a, " b =", b, " c =", c );
```

γράφουμε

```
   printf( " a = %f   b = %f   c = %f\n", a, b, c );
```

[14] Για ακέραιες τιμές χωρίς πρόσημο έχουμε την προδιαγραφή "**%u**" με τους ίδιους τροποποιητές.

2.13 Λάθη, Λάθη ...

Αν ακολουθείς τις οδηγίες μας και περνάς τα προγράμματα και πειραματίζεσαι θα πρέπει να έχεις αγανακτήσει ήδη με τον μεταγλωττιστή σου και τα λάθη που βρίσκει. Κάνε υπομονή. Όσο προχωράς θα ελαττωθούν τα λάθη απροσεξίας καθώς και τα σοβαρότερα.

- ***Είναι βασικό να μάθεις να βρίσκεις και να διορθώνεις μόνος/η σου τα λάθη των προγραμμάτων σου.*** [15]

Μόνο έτσι θα μάθεις προγραμματισμό! (Ε, στην αρχή μπορεί να χρειαστείς και λίγη βοήθεια...)

Αλλά, ας τα πάρουμε από την αρχή:

- Το ";" είναι για τη C++ **τερματιστής εντολής** (statement terminator). Με αυτό τελειώνουν οι εντολές και οι δηλώσεις.
- Όλες οι μεταβλητές και οι σταθερές (με όνομα) πρέπει να δηλώνονται υποχρεωτικώς πριν χρησιμοποιηθούν· καλύτερα να τις δηλώνεις στην αρχή.
- Μη χρησιμοποιείς σε παραστάσεις μεταβλητές που δεν τους έχεις δώσει τιμή πιο πριν είτε με εντολή εκχώρησης, είτε με εντολή `cin >>` ....

Συνηθισμένα λάθη εκτέλεσης είναι αυτά της ανάγνωσης στοιχείων, όταν μάλιστα πρόκειται για αριθμούς:

- Αν πληκτρολογείς τιμή μεταβλητής ακέραιου τύπου, δώσε σταθερά του τύπου αυτού: πρόσημο, αν χρειάζεται, και στη συνέχεια ψηφία. Το 17.0 *δεν είναι* σταθερά ακέραιου τύπου, ούτε το 17,0 ούτε το 1.7e1. Το 17 είναι! Και, για όνομα του Θεού, μην πληκτρολογήσεις *INT_MAX*.
- Αν πληκτρολογείς τιμή μεταβλητής πραγματικού τύπου, δώσε μια σταθερά του τύπου αυτού. Η υποδιαστολή είναι "." όχι ",".

Ξανακοίταξε τώρα τα προγράμματα με τα ανεξήγητα λάθη, διόρθωσέ τα και συνέχισε την προσπάθεια. Καλή τύχη!

2.14 Τι (Πρέπει να) Έμαθες Μέχρι Τώρα

Η βασική έννοια αυτού του κεφαλαίου είναι η *μεταβλητή*, το βασικό εργαλείο για τη διαχείριση της κύριας μνήμης του ΗΥ. Πρέπει να έμαθες

- Να *δηλώνεις* μεταβλητές και –αν θέλεις– να *ορίζεις* την αρχική τιμή τους.
- Να ορίζεις/αλλάζεις την τιμή τους με την *εντολή εκχώρησης*.
- Να ορίζεις/αλλάζεις την τιμή τους με *εντολή εισόδου* από το πληκτρολόγιο.

Ακόμη, πρέπει να έμαθες μερικά πράγματα για τους βασικούς (αριθμητικούς) *τύπους* της C++ και πώς να «μετατρέπεις τον τύπο» μιας τιμής.

Τι προγράμματα μπορείς να γράψεις; Αυτά που λύνουν τις ασκήσεις της Β Ομάδας είναι χαρακτηριστικά.

---

[15] Ξέρεις πως λέγεται αυτή η δουλειά στα αγγλικά; «debugging», που θα μπορούσε να μεταφραστεί «απεντόμωση», ενώ σημαίνει «διόρθωση λαθών». Ρώτησε να μάθεις γιατί λέγεται έτσι και γιατί τα λάθη προγραμματισμού λέγονται «bugs» (ζωϋφια, κοριοί). Έχει ιστορική και χιουμοριστική αξία.

Ασκήσεις

Α Ομάδα

2-1 Δίνεται το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int i, j, k;

   i = 5;  j = 7;  k = 10;
   cout << i << ' ' << (i+1) << ' ' << (i+j)
        << ' ' << (i + j*k) << endl;
   k = i + j;         cout << k << endl;
   j = j + 1;         cout << j << endl;
   i = j + 2*i + j;   cout << i << endl;
}
```

Εκτέλεσε το πρόγραμμα (*εσύ*, όχι ο υπολογιστής), όπως κάναμε στο παράδειγμα της αντιμετάθεσης.

2-2 Το ίδιο για το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{  double first, second, realResult;
   int    third, intResult;

   cin >> first >> second >> third;
   realResult = first + second + third;
   intResult = first + second - third;
   realResult = realResult - intResult;
   intResult = intResult - realResult;
   cout << realResult << intResult << endl;
}
```

Δίνεται μια γραμμή με στοιχεία εισόδου:

1.5e1 8.1 5 7.41e-5 33.0

2-3 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο μια τιμή –ας πούμε x– και θα υπολογίζει και θα τυπώνει τις τιμές των παραστάσεων για την τιμή που διαβάστηκε:

$$\frac{1}{1+\sqrt{x}} \qquad 1+e^{-x/2} \qquad x^x + x^{x/2}$$

$$\frac{\sigma\upsilon\nu x - \eta\mu x}{1+\sqrt{2x}} \qquad \frac{e^{-x/2}+e^{x/2}}{2} \qquad x^{x/3} + x^{3x}$$

όπου e η βάση των φυσικών λογαρίθμων.

2-4 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή μιας γωνίας σε μοίρες –ας πούμε x– και θα υπολογίζει και θα τυπώνει τα:

$$\eta\mu x,\ \sigma\upsilon\nu x,\ \varepsilon\varphi x,\ \sigma\varphi x,\ \tau\varepsilon\mu x,\ \sigma\tau\varepsilon\mu x$$

Β Ομάδα

2-5 Γράψε πρόγραμμα που διαβάζει από το πληκτρολόγιο δυο αριθμούς: β (θετικό πραγματικό, $\neq 1$) και x (θετικό πραγματικό) και θα υπολογίζει και θα τυπώνει τα $\log_\beta x$ και β^x.
Υπόδ.: Χρησιμοποίησε την ιδιότητα: $\log_\beta x = \ln x / \ln \beta$.

2-6 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή κάποιου αντικειμένου, χωρίς ΦΠΑ και τον συντελεστή ΦΠΑ (%). Θα υπολογίζει και θα γράφει την τιμή με ΦΠΑ.

2-7 Γράψε πρόγραμμα που θα διαβάζει τις καρτεσιανές συντεταγμένες, x, y, ενός σημείου στο επίπεδο και θα υπολογίζει και θα τυπώνει τις πολικές r, ϕ. Υπόθεσε ότι $x > 0$. Υπολόγισε τη ϕ: α) με την *atan* β) με την *atan2*.

Υπόδ.: $r = \sqrt{x^2 + y^2}$, $\phi = τοξεφ(y/x)$.

2-8 Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 2%, επί του βασικού μισθού, για κάθε χρόνο υπηρεσίας. Γράψε πρόγραμμα που θα διαβάζει τον βασικό μισθό και τα χρόνια υπηρεσίας ενός υπαλλήλου και θα υπολογίζει το χρονοεπίδομα και το συνολικό μισθό.

2-9 Αν συνδέσουμε δυο αντιστάσεις R_1 και R_2 σε σειρά, η συνολική αντίσταση είναι $R = R_1 + R_2$. Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 και θα υπολογίζει και θα τυπώνει την R.

2-10 Αν συνδέσουμε δυο αντιστάσεις R_1 και R_2 παραλλήλως, η συνολική αντίσταση είναι:

$$R = \frac{R_1 R_2}{R_1 + R_2}$$

Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 και θα υπολογίζει και θα τυπώνει την R.

2-11 Ένα κύκλωμα αποτελείται από πυκνωτή χωρητικότητας C και αντιστάτη αντίστασης R συνδεμένα παράλληλα. Στα κοινά τους άκρα συνδέουμε πηγή εναλλασσόμενης τάσης $U = U_0 ημ(2πνt)$.

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο τις τιμές των U_0 (V), $ν$ (Hz), C (F), R ($Ω$) και θα υπολογίζει και θα γράφει στην οθόνη:

- την ενεργή τιμή της τάσης $U_{εν} = U_0 / \sqrt{2}$,
- την κυκλική συχνότητά της $ω = 2πν$,
- την εμπέδηση $Z = \sqrt{\frac{1}{R^2} + (ωC)^2}$ του κυκλώματος,
- το πλάτος i_0 και την ενεργή τιμή $i_{εν}$ του ρεύματος,
- τη διαφορά φάσης μεταξύ U και i, $\phi = τοξεφ(ωCR)$.

2-12 Στην §2.5 είδαμε ένα πρόγραμμα που μας δίνει τη μέγιστη και την ελάχιστη τιμή του τύπου `int`. Αν στις εντολές του προγράμματος αντικαταστήσεις το `"INT"` με `"LONG"`, `"SHRT"`, `"CHAR"`, `"UINT"`, `"ULONG"`, `"USHRT"`, `"UCHAR"`, θα πάρεις τα στοιχεία του Πίν. 2-1 για τους τύπους `long int`, `short int`, `char`, `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char`, αντιστοίχως. Γράψε πρόγραμμα που θα βγάζει αυτά τα στοιχεία.

Στην ίδια παράγραφο είδαμε και ένα πρόγραμμα που μας δίνει τις χαρακτηριστικές τιμές του τύπου `double`. Αν αλλάξεις το `"DBL"` σε `"FLT"` και `"LDBL"` θα πάρεις τα στοιχεία των τύπων `float` και `long double` αντιστοίχως. Γράψε πρόγραμμα που θα βγάζει τα στοιχεία του Πιν. 2-2.

Γ Ομάδα

2-13 Γράψε πρόγραμμα που θα λύνει το πρόβλημα της ελεύθερης πτώσης, στην περίπτωση που η αρχική ταχύτητα δεν είναι μηδέν και έχει συνιστώσες στον κατακόρυφο και τον οριζόντιο άξονα.

κεφάλαιο

3

* Το Σωστό Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσουμε την έννοια της επαλήθευσης προγράμματος και τις βασικές σχετικές διαδικασίες.
Προσδοκώμενα αποτελέσματα:
Να μπορείς να αποδείξεις την ορθότητα απλών προγραμμάτων.
Έννοιες κλειδιά:

- *προδιαγραφές προγράμματος*
- *απόδειξη ορθότητας ή επαλήθευση προγράμματος*
- *συνθήκες επαλήθευσης*
- *αξίωμα της εκχώρησης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Γράφαμε στην §0.4:

- ***το πρόγραμμα είναι ένα προϊόν για το οποίο πρέπει να υπάρχουν προδιαγραφές,***
- ***το πρόγραμμα γράφεται έτσι ώστε να συμμορφώνεται με τις προδιαγραφές,***
- ***πρέπει να αποδεικνύεται η ορθότητα του προγράμματος (δηλαδή η συμμόρφωση με τις προδιαγραφές).***

Αν αυτά που λέγαμε στις §0.3 και 0.4 σου φάνηκαν αφηρημένα και «ακαταλαβίστικα» ήρθε η ώρα να τα δεις σε εφαρμογή σε συγκεκριμένα παραδείγματα, να δεις πώς γίνεται η **επαλήθευση** (verification) ή **απόδειξη ορθότητας** (correctness proof) προγράμματος.

Ακόμη, στην §2.2 γράφαμε:

- ***Με ένα παράδειγμα ή με στιγμιότυπα εκτέλεσης μπορείς να ανακαλύψεις λάθη στο πρόγραμμά σου αλλά δεν μπορείς να αποδείξεις ότι το πρόγραμμα είναι σωστό.***

Πολλοί συγχέουν τις **δοκιμές προγράμματος** (program testing) με την απόδειξη ορθότητας· καμία σχέση! Μπορείς να κάνεις δοκιμές στο πρόγραμμά σου για να βρεις και να διορθώσεις λάθη που ενδεχομένως έχει. Το να περάσει το πρόγραμμα επιτυχώς όλες τις δοκιμές δεν σημαίνει ότι δεν έχει άλλα λάθη. Θα πεις «Και αν το δοκιμάσω επιτυχώς για όλες τις τιμές που μπορεί να πάρουν οι μεταβλητές του;» Ε, τότε εντάξει, αλλά τι πρόγραμμα είναι αυτό;

3.1 Ξεκινώντας με τη Δήλωση

Οι συνθήκες που ισχύουν μετά από τις δηλώσεις είναι αυτές που συνάγονται από τους Πιν. 2-1 και 2-2. Ας πούμε ότι κάνουμε τις δηλώσεις:

```
int          number;
unsigned int uNumber;
double       rNumber;
```

Όπως είπαμε «*έχουμε τη σιγουριά ότι η number έχει πάντοτε ακέραιη τιμή ανάμεσα στη μεγαλύτερη και τη μικρότερη που μπορεί να παρασταθεί στον τύπο* `int`» που μπορεί να γραφεί ώς εξής:

$$(number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

Αυτή είναι η **αναλλοίωτη** (invariant) του τύπου `int`.

Παρομοίως, για τον *uNumber* θα έχουμε:

$$(uNumber \in \mathbb{N}) \;\&\&\; (0 \le uNumber \le UINT\_MAX)$$[1]

και για τον *rNumber*:

$$(rNumber \in \mathbb{Q}) \;\&\&\; (-DBL\_MAX \le rNumber \le DBL\_MAX)$$

Οι συνθήκες αυτές ισχύουν «όσο ζουν» οι μεταβλητές.

Αν δώσουμε και αρχική τιμή; Π.χ.:

```
int number( 37 );
```

τότε θα έχουμε:[2]

$$(number == 37) \;\&\&\; (number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

Εδώ έχουμε πλεονασμό: αφού *number* `==` 37 προφανώς θα έχουμε και $number \in \mathbb{Z}$ και $INT\_MIN \le number \le INT\_MAX$. Δεν πειράζει· όπως είπαμε, αυτές που έχουν σχέση με τη δήλωση ισχύουν «όσο ζουν» οι μεταβλητές ενώ η "*number* `==` 37" μετά από λίγο μπορεί να μην ισχύει.

Πρόσεξε όμως και το εξής: αν δώσεις

```
int number( 3.0/2 );
```

μπορεί να πάρεις από τον μεταγλωττιστή μια *ειδοποίηση* (warning) της μορφής «`converting to 'int' from 'double'`» αλλά η *number* θα πάρει αρχική τιμή "`1`". Μπορούμε λοιπόν να πούμε ότι μετά την

```
int number( Π );
```

θα έχουμε:

---

[1] *UINT_MAX;;;* Έλυσες την άσκ. 2-12;

[2] Τι είναι πάλι αυτό το "`==`"; Επειδή το "`=`" στη C++ έχει συγκεκριμένο νόημα, αυτό που καθορίζεται στην εκχώρηση, χρησιμοποιούμε το "`==`" για τη σύγκριση «είναι ίσο με». Αυτό θα το ξανασυζητήσουμε.

(*number* == static_cast<int>(*Π*)) &&
(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Γενικώς, μπορούμε να πούμε ότι μετά την

```
T x( Π );
```

και αν ορίζεται η static_cast<*T*>(*Π*), θα έχουμε:

(*x* == static_cast<*T*>(*Π*)) && I_T(*x*)

όπου I_T είναι η *αναλλοίωτη* του τύπου *T*, δηλαδή η συνθήκη που ισχύει για όλες τις τιμές τύπου *T*.

3.2 Εντολές Εισόδου και Εξόδου

Εδώ τα πράγματα είναι απλά:

- Μια εντολή εξόδου δεν έχει επίδραση στις τιμές των μεταβλητών. Έτσι, αν ισχύει η *P* πριν από την

```
cout << number;
```

θα ισχύει και μετά την εκτέλεσή της.

- Μετά από μια εντολή εισόδου –αν η τιμή που εισάγεται μπορεί να παρασταθεί– δεν είναι δυνατόν να ξέρουμε κάτι περισσότερο από αυτό που ξέρουμε από τις δηλώσεις. Για παράδειγμα, με τις δηλώσεις της προηγούμενης παραγράφου, μετά την:

```
cin >> number;
```

θα έχουμε:

(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Οτιδήποτε άλλο ίσχυε για τη *number* (π.χ. κάτι σαν *number* == 37) παύει να ισχύει.

3.3 Οι Σταθερές στις Αποδείξεις

Πώς χειριστήκαμε τη σταθερά *g* στο παράδειγμα του προηγούμενου κεφαλαίου. *Είχαμε τη συνθήκη g* == 9.81 *αναλλοίωτη σε ολόκληρο το πρόγραμμα*.

Αυτό ακριβώς θα κάνουμε και με όλες τις σταθερές: αν έχουμε δηλώσει:

```
const T cn( c );
```

σε ολόκληρο το πρόγραμμά[3] μας ισχύει η *cn* == *c* και η αναλλοίωτη I_T(*cn*) του τύπου *T*:

(*cn* == *c*) && I_T(*cn*)

3.4 Το Αξίωμα της Εκχώρησης

Να δούμε τώρα πώς μπορούμε να διατυπώσουμε με ακρίβεια το νόημα της εντολής εκχώρησης. Ας πούμε ότι έχουμε δηλώσει

```
double x;
int y;
```

και έχουμε στο πρόγραμμά μας τις εντολές:

```
x = 7;      // x == 7.0
y = x + 4;  // y == 11
```

Σε σχόλια έχουμε βάλει το τι ξέρουμε ότι ισχύει σε εκείνο το σημείο της εκτέλεσης σύμφωνα με αυτά που είπαμε: «υπολογίζεται η τιμή της παράστασης, η τιμή μετατρέπεται στον τύπο της μεταβλητής, η τιμή φυλάγεται ως τιμή της μεταβλητής».

[3] Για την ακρίβεια: σε ολόκληρη την *εμβέλεια* της δήλωσης, όπως θα μάθουμε αργότερα.

Πάντως, στα προγράμματα που θα δούμε στη συνέχεια, το συνηθισμένο θα είναι να μην ξέρεις την τιμή της παράστασης· θα ξέρεις όμως μερικές ιδιότητές της, π.χ.: ας υποθέσουμε ότι σε κάποιο πρόγραμμα έχουμε δηλώσει

```
double x, a, b;
```

και έχουμε την εντολή:

```
x = pow( a + b, 2 ) + 1;
```

Πριν από την εκτέλεση της εντολής έχουμε a == a_0, b == b_0, αλλά τα a_0, b_0 μας είναι άγνωστα όταν γράφουμε το πρόγραμμα· ξέρουμε όμως ότι:

$$|a_0 + b_0| \le ½$$

Τι ξέρουμε για την τιμή της x μετά την εκτέλεση της εντολής; Η τιμή Π του δεξιού μέρους όταν εκτελείται η εκχώρηση είναι:

$$(a_0 + b_0)^2 + 1 = (|a_0 + b_0|)^2 + 1$$

και αφού ξέρουμε ότι: $|a_0 + b_0| \le ½$, θα έχουμε:

$$\Pi = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Μπορούμε λοιπόν να πούμε ότι μετά την εκτέλεση της εντολής εκχώρησης θα έχουμε:

$$x = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Να λοιπόν τι μπορούμε να πούμε γενικά:

- Αν
  - η `v` είναι τύπου T και
  - πριν από την εκτέλεση της `v` = Π ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$ και
- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά

τότε

- Μετά την εκτέλεση της `v` = Π ισχύει η $P(v)$, δηλαδή η συνθήκη που προκύπτει από στην $P($ `static_cast<`T`>(`Π`)` $)$ αν αντικαταστήσουμε με τη v τη `static _cast<`T`>(` Π `)`.

Αυτό είναι το **αξίωμα της εκχώρησης**.

Παρατηρήσεις: ▶

1. Η $P($ `static_cast<`T`>(`Π`)` $)$ μπορεί να περιλαμβάνει και συνθήκες που δεν περιλαμβάνουν τη v. Αυτές δεν επηρεάζονται από την εντολή εκχώρησης, αλλά παραμένουν αναλλοίωτες από αυτήν. Στο παράδειγμά μας, συμφώνως με όσα είπαμε, θα έχουμε πριν από την εντολή:

$$(a == a_0) \;\&\&\; (b == b_0) \;\&\&\; (|a_0 + b_0| \le ½)$$

Όλα αυτά δεν επηρεάζονται από την εκτέλεση της

```
x = pow( a + b, 2 ) + 1;
```

και ισχύουν και μετά από αυτήν.

2. Ό,τι ίσχυε για τη v πριν από την εκτέλεση της "`v` = Π" δεν ισχύει μετά από αυτήν. Γυρνώντας στο παράδειγμά μας, ας πούμε ότι πριν από την εκτέλεση της `x = pow(a + b, 2) + 1` έχουμε x == 0. Μετά την εκτέλεσή της, το μόνο από τα «παλιά» που μένει να ισχύει είναι η $-DBL\_MAX \le x \le DBL\_MAX$ που απορρέει από τη δήλωση `double x`. Η x == 0 παύει να ισχύει και αντικαθίσταται από την $x = (a_0 + b_0)^2 + 1 \le {}^5/_4$. ◀

Μπορούμε να δούμε το αξίωμα της εκχώρησης και με έναν άλλο τρόπο:

- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά,
- Για να ισχύει μετά την εκτέλεση της `v` = Π η $P(v)$

θα πρέπει

- πριν από την `v` = Π να ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$, δηλαδή η συνθήκη που προκύπτει αν στην $P(v)$ βάλουμε όπου v τη `static_cast<`T`>(` Π `)`.

Όπως θα δεις στη συνέχεια, αυτόν τον τρόπο θα χρησιμοποιούμε συνήθως.

3.5 Συνθήκες "true" και "false"

Πολύ συχνά θέλουμε να γράψουμε τη συνθήκη: «για οποιαδήποτε τιμή των μεταβλητών μας». Αυτή γράφεται ως εξής: `true`. Να ένα παράδειγμα τέτοιας συνθήκης:

$(x \geq 0)$ `||` $(x < 0)$

Όπως υπάρχει συνθήκη `true`, υπάρχει και συνθήκη `false` και σημαίνει: «δεν ισχύει όποιες και αν είναι οι τιμές των μεταβλητών μας.» Π.χ.

$(x \geq 0)$ `&&` $(x < 0)$

3.6 Το Πρόγραμμα Μαζί με την Απόδειξη

Και τώρα να δούμε, με παραδείγματα, πώς χρησιμοποιούμε το αξίωμα της εκχώρησης για να αποδείξουμε την ορθότητα ενός προγράμματος. Στην πραγματικότητα θα δούμε τρεις φορές, από τρεις διαφορετικές σκοπιές, το παράδειγμα της αντιμετάθεσης. Όπως είπαμε όμως, η ορθότητα του προγράμματος αναφέρεται σε συγκεκριμένες **προδιαγραφές** (program specifications). Το πρώτο πράγμα που πρέπει να κάνουμε είναι να διατυπώσουμε αυτές τις προδιαγραφές για το πρόβλημά μας. Αυτό είναι απλό:

Προϋπόθεση: $(a == a_0)$ `&&` $(b == b_0)$

Απαίτηση: $(a == b_0)$ `&&` $(b == a_0)$

που σημαίνουν: Αν πριν από την πρώτη εντολή ισχύει η συνθήκη $(a == a_0)$ `&&` $(b == b_0)$ τότε μετά την εκτέλεση όλων των εντολών θα ισχύει: $(a == b_0)$ `&&` $(b == a_0)$.

Πώς γίνεται *πρακτικά* να πάρουμε την απόδειξη ορθότητας ενός προγράμματος; Θα δούμε πώς γίνεται αυτό από τρεις διαφορετικές «απόψεις»:

- Γράφουμε το πρόγραμμα και –*παράλληλα*– καταγράφουμε τις συνθήκες που προκύπτουν από την εκτέλεση της κάθε εντολής.
- Μας δίνεται το πρόγραμμα (ή το γράφουμε εμείς) και μετά αποδεικνύουμε ότι είναι σωστό.
- Γράφουμε το πρόγραμμα προσπαθώντας να βάζουμε τις εντολές που θα μας δώσουν αυτά που ζητούν οι προδιαγραφές.

Θα δοκιμάσουμε αυτές τις τρεις «απόψεις» στο πρόγραμμα της *αντιμετάθεσης*. Και στις τρεις περιπτώσεις, θα παρεμβάλουμε, μέσα σε σχόλια, τις συνθήκες που μας ενδιαφέρουν.

3.6.1 Απόδειξη – Πρόγραμμα Παραλλήλως

Ας δούμε αρχικά την απόδειξη που γίνεται παραλλήλως με το γράψιμο. Στην περίπτωση αυτή θα πρέπει να βάζουμε μετά από κάθε εντολή τη συνθήκη που προκύπτει από την εκτέλεσή της. Θα τη βρίσκουμε με την εξής συνταγή που βγάζουμε από το αξίωμα της εκχώρησης:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(v) ως εξής:***
  1. ***Θα αντιγράψεις την P(`static_cast<T>(Π)`).***
  2. ***Θα σβήσεις όποιαδήποτε συνθήκη περιλαμβάνει τη v. Η τιμή της v άλλαξε και ό,τι ίσχυε γι' αυτήν παύει να ισχύει.***
  3. ***Οτιδήποτε υπάρχει για την Π θα γραφεί άλλη μια φορά –με `&&`– με αντικατάσταση της Π από την v.***

Ξεκινούμε λοιπόν με:

```
// (a == a0) && (b == b0)
```

Το πρώτο που κάναμε ήταν να φυλάξουμε κάπου (στην *s*) την αρχική τιμή της *a*. Αυτό γίνεται με την εντολή εκχώρησης `s = a`. Τι θα ισχύει μετά την εκτέλεση αυτής της εντολής;

Ας βρούμε τη συνθήκη που θα ισχύει μετά την εντολή. Στην περίπτωσή μας μεταβλητή είναι η *s* και παράσταση είναι η *a*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0).
2. Αφού δεν υπάρχει τίποτε για την *s*, δεν κάνουμε διαγραφές.
3. Για την *a* υπάρχει η a == a_0· την αντιγράφουμε αντικαθιστώντας την *a* με *s*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
```

Ύστερα από αυτό, μπορούμε να βάλουμε στην *a* την τιμή της *b* (`a = b`). Πώς παίρνουμε τη συνθήκη που θα ισχύει μετά την εντολή. Εδώ, μεταβλητή είναι η *a* και παράσταση είναι η *b*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0) && (s == a_0).
2. Διαγράφουμε από αυτήν ό,τι ισχύει για την *a*: (b == b_0) && (s == a_0).
3. Για τη *b* υπάρχει η b == b_0· την αντιγράφουμε αντικαθιστώντας τη *b* με *a*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
```

Τέλος, δίνουμε στη *b* την τιμή της *a*, που έχουμε «φυλάξει» στην *s*. Ό,τι ίσχυε για τη *b* (b == b_0) παύει να ισχύει· από εδώ και πέρα για την *b* ισχύει ό,τι ίσχυε μέχρι τώρα την *s* (b == a_0):

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
   b = s
// (a == b0) && (b == a0) && (s == a0)
```

Αποδείξαμε ότι το πρόγραμμα είναι σωστό; Η συνθήκη που πήραμε τελικώς δεν είναι η απαίτηση. Αλλά... για να δούμε: Αποδείξαμε ότι αν το πρόγραμμά μας εκτελεσθεί με προϋπόθεση (a == a_0) && (b == b_0), θα καταλήξουμε στη συνθήκη: (a == b_0) && (b == a_0) && (s == a_0). Στο Παρ. Α, §Α.4, βλέπουμε ότι (P && Q) ⇒ P· στην περίπτωσή μας, αν πάρουμε ως P την (a == b_0) && (b == a_0) και ως Q την (s == a_0) έχουμε:

$$((a == b_0)\ \&\&\ (b == a_0))\ \&\&\ (s == a_0) \Rightarrow ((a == b_0)\ \&\&\ (b == a_0))$$

Αλλά ο συμπερασματικός κανόνας (*E*1), που είδαμε στην §0.4.1, λέει ότι αποδείξαμε την ορθότητα του προγράμματός μας.

3.6.2 Απόδειξη εκ των Υστέρων (από το Τέλος προς την Αρχή)

Τώρα, ας έρθουμε στη δεύτερη περίπτωση: Γράψαμε (ή μας δίνεται) το πρόγραμμα και πρέπει να αποδείξουμε ότι είναι σωστό σε σχέση με τις προδιαγραφές του. Μπορούμε να κάνουμε την απόδειξη είτε *από την αρχή προς το τέλος* είτε *από το τέλος προς την αρχή*.

Για το παράδειγμα της αντιμετάθεσης, αν κάνουμε την απόδειξη από την αρχή προς το τέλος, θα πάρουμε την απόδειξη που είδαμε παραπάνω.

Για να κάνουμε μια απόδειξη *από το τέλος προς την αρχή* θα πρέπει να μπορούμε να απαντούμε στο εξής ερώτημα: Αν ξέρω τη συνθήκη Q που ισχύει μετά την εκτέλεση της εντολής E, πώς μπορώ να βρω τη συνθήκη που πρέπει να ισχύει πριν από αυτήν; Χρησιμοποιώντας τη δεύτερη διατύπωση του αξιώματος της εκχώρησης, που είδαμε στο τέλος της §3.4, βγάζουμε την εξής συνταγή:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(`static_cast<T>(Π)`) ως εξής:***

1. *Θα αντιγράψεις την P(v).*
2. *Οπου υπάρχει η v θα την αντικαταστήσεις με τη* `static_cast<T>(Π)`.

🡒🡒🡒

Στο παράδειγμά μας θα πρέπει να αποδείξουμε:

```
// (a == a0) && (b == b0)
   s = a;  a = b;  b = s;
// (a == b0) && (b == a0)
```

Ξεκινούμε από την:

```
   b = s;
// (a == b0) && (b == a0)
```

Πώς παίρνουμε τη συνθήκη που θα ισχύει πριν από την εντολή; Εδώ, μεταβλητή είναι η *b* και παράσταση είναι η *s*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (b == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *b* την *s*: (a == b_0) && (s == a_0).

Φθάνουμε λοιπόν στο εξής:

```
   s = a;  a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Με τον ίδιο τρόπο βρίσκουμε τη συνθήκη που πρέπει να ισχύει πριν από την `a = b`. Μεταβλητή είναι η *a* και παράσταση είναι η *b*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (s == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *a* τη *b*: (b == b_0) && (s == a_0).

Έχουμε λοιπόν:

```
   s = a;
// (b == b0) && (s == a0)
   a = B;
// (a == b0) && (s == a0)
   B = s;
// (a == b0) && (b == a0)
```

Τέλος, ας βρούμε τη συνθήκη που θα πρέπει να ισχύει πριν από την `s = a` ώστε μετά από αυτήν να έχουμε (b == b_0) && (s == a_0). Μεταβλητή μας είναι η *s* και παράσταση η *a*. Άρα:

1. παίρνουμε την (b == b_0) && (s == a_0) και
2. αντικαθιστούμε το *s* με το *a* και παίρνουμε την (b == b_0) && (a == a_0).

Φθάνουμε λοιπόν στο εξής:

```
// (b == b0) && (a == a0)
   s = a;
// (b == b0) && (s == a0)
   a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Για να είναι το πρόγραμμά μας σωστό θα πρέπει: η συνθήκη (b == b_0) && (a == a_0) –που θέλουμε να ισχύει πριν από την πρώτη εντολή– να συνάγεται από την προϋπόθεση. Αλλά αυτό συμβαίνει: Επειδή η πράξη && είναι αντιμεταθετική, από την προϋπόθεση (a == a_0) && (b == b_0) παίρνουμε τη (b == b_0) && (a == a_0). Άρα το πρόγραμμά μας είναι σωστό.

🡐🡐🡐

3.6.3 Το Πρόγραμμα από τις Προδιαγραφές

Τέλος, ας έρθουμε στην τρίτη «άποψη»: Να γράψουμε το πρόγραμμα με οδηγό τις προδιαγραφές; Θα ρωτήσεις: Δηλαδή μπορούμε να το γράψουμε και αλλιώς; Όχι βέβαια·

αυτό που εννοούμε είναι να δουλέψουμε πιο συστηματικά και από τις απαιτήσεις να βρίσκουμε τις εντολές.

Θα δουλέψουμε και πάλι από το τέλος προς την αρχή.

✎✎✎

Στο παράδειγμά μας ξεκινούμε με την ερώτηση: με ποια εντολή μπορούμε να καταλήξουμε στη συνθήκη:

```
// (a == b0) && (b == a0)
```

Σύμφωνα με το αξίωμα της εκχώρησης μπορούμε να πάρουμε την b `==` a_0, αν εκτελεσθεί η εντολή `b = s` και πριν από αυτήν ισχύει η s `==` a_0. Παρόμοια, μπορούμε να φτάσουμε στην a = b_0, αν εκτελεσθεί η εντολή `a = t` και πριν από αυτήν ισχύει η t `==` b_0:

```
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Πώς μπορούμε να φτάσουμε στην (t `==` b_0) `&&` (s `==` a_0); Σύμφωνα με το αξίωμα της εκχώρησης και πάλι, μπορούμε να φτάσουμε στη συνθήκη αυτή αν εκτελεσθούν οι εντολές `s = a; t = b` και πριν από αυτές ισχύει η (a `==` a_0) `&&` (b `==` b_0), που είναι η προϋπόθεσή μας. Άρα, το παρακάτω πρόγραμμα είναι σωστό:

```
// (a == a0) && (b == b0)
   s = a;  t = b;
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Εδώ χρησιμοποιήσαμε δύο βοηθητικές μεταβλητές, αλλά αυτό δεν είναι κάτι τραγικό.

✎✎✎

3.7 Διά Ταύτα ...

Πριν κάνουμε μια σύγκριση των τριών απόψεων να προλάβουμε κάποιους που μπορεί να κάνουν ήδη απαισιόδοξες σκέψεις του εξής τύπου: «Και αν έχω ένα πρόγραμμα των εκατό γραμμών τι γίνεται; Θα έχω συνθήκες που πιάνουν σελίδες! Για να μη πούμε τι θα γίνει όταν, όπως γίνεται στην πραγματικότητα, έχουμε χιλιάδες γραμμές προγράμματος!» Η απάντηση είναι: τα προγράμματα γράφονται με τη διαδικασία της **βήμα-προς-βήμα ανάλυσης** (§0.5). Έτσι τα κομμάτια προγράμματος που πρέπει να γραφούν και να αποδειχτούν έχουν μέγεθος και «δυσκολία» που μπορούμε να διαχειριστούμε[4]. Αυτά ισχύουν και για τον αντικειμενοστραφή προγραμματισμό· εκεί όμως έχουμε διαφορετικό τρόπο καταμερισμού της «δυσκολίας».

Ποιος από τους τρεις τρόπους εργασίας είναι η πιο συνηθισμένος; Ο δεύτερος και μάλιστα με απόδειξη από το τέλος προς την αρχή! Δηλαδή, ο προγραμματιστής –με τη βήμα-προς-βήμα ανάλυση ή με άλλον τρόπο– έχει φτάσει σε ένα πρόβλημα που από τις προδιαγραφές του «φαίνεται» η λύση. Γράφει λοιπόν τη λύση και μετά αποδεικνύει ότι είναι σωστή.

Ο πρώτος, όπως και ο δεύτερος, αλλά με απόδειξη από την αρχή προς το τέλος, είδες ότι έχουν πιο πολύπλοκη συνταγή για την εύρεση των συνθηκών που μας χρειάζονται. Έχει όμως και το εξής πρόβλημα: Οι συνθήκες «φουσκώνουν» πολύ γρήγορα –με κομμάτια που είναι άχρηστα στη συνέχεια– και γίνονται δύσχρηστες. Στο παράδειγμά μας είδες ότι καταλήξαμε με την (άχρηστη) s `==` a_0 μετά το τελευταίο βήμα.

Ο τρίτος έχει ενδιαφέρον όταν προσπαθούμε να λύσουμε το εξής πρόβλημα: Να γραφεί ένα πρόγραμμα που θα τροφοδοτείται με τις προδιαγραφές ενός προγράμματος και θα το γράφει, ας πούμε σε C++ (αυτόματη σύνθεση προγράμματος). Φυσικά, αν μπορέσουμε να

[4]Πάντως, συνθήκες που πιάνουν αρκετές γραμμές είναι συνηθισμένες.

γράψουμε αλγόριθμο που να κάνει κάτι τέτοιο, θα μπορούμε, πολύ πιο εύκολα να διατυπώσουμε κανόνες που να τους μάθει ένας άνθρωπος.

Εμείς θα χρησιμοποιούμε κατά κύριο λόγο τον δεύτερο τρόπο, με απόδειξη από το τέλος προς την αρχή και σπανιότερα τους άλλους.

Κάποιοι θα αναρωτηθούν «όταν τελειώσει η απόδειξη, θα σβήσουμε τα σχόλια με τις συνθήκες;» Καλύτερα να τις κρατάμε, τουλάχιστον ορισμένες από αυτές. Οι **συνθήκες επαλήθευσης** (verification conditions) είναι απαραίτητες σε περίπτωση που θα θελήσουμε να αλλάξουμε κάτι αργότερα.

3.8 Τα Προβλήματα των Τύπων Κινητής Υποδιαστολής

Πριν προσπαθήσουμε να κάνουμε μια απόδειξη προγράμματος με πραγματικούς αριθμούς (αυτού για την ελεύθερη πτώση της §2.7) ας δούμε μερικά (ακόμη) προβλήματα των τύπων κινητής υποδιαστολής.

Όπως λέγαμε στην §2.5 «*Η παράσταση των τιμών τύπου* `double` *(και* `float` *και* `long double`*) γίνεται προσεγγιστικώς.*» και «*οι πράξεις ... στους πραγματικούς τύπους (όπως ο* `double`*) δεν είναι* [ακριβείς]». Ήδη, στην §1.7.1 είδαμε ότι η:

```
cout.precision(20);
cout << 12.34567890123456789 << endl;
```

μας δίνει:

```
12.345678901234567884
```

Μια ματιά στον Πίν. 2-2 μας λέει ότι η ακρίβεια που μπορούμε να ζητήσουμε από τον `double` δεν υπερβαίνει τα 16 ψηφία. Φυσικά και στην περίπτωση αυτή έχουμε:

```
12.34567890123457
```

Ας προσπαθήσουμε να υπολογίσουμε δυνάμεις με τη χρήση της pow:

```
v = pow(0.3,4.0);   cout << " 0.3^4 = " << v << endl;
v = pow(0.7,2.0);   cout << " 0.7^2 = " << v << endl;
```

Αποτελέσματα:

```
0.3^4 = 0.0081
0.7^2 = 0.4899999999999999
```

Το δεύτερο είναι εντυπωσιακό.

Αν προσπαθήσουμε να κάνουμε το ίδιο πράγμα με τιμές τύπου `float` (εδώ βέβαια η ακρίβεια δεν μπορεί να είναι μεγαλύτερη από 8 ψηφία):

```
cout.precision(8);
v = pow(0.3f,4.0);  cout << " 0.3^4 = " << v << endl;
v = pow(0.7f,2.0);  cout << " 0.7^2 = " << v << endl;
```

θα πάρουμε:

```
0.3^4 = 0.0081000013
0.7^2 = 0.48999998
```

Τι συμπέρασμα βγαίνει από τα παραπάνω; Στις προδιαγραφές, όπου έχουμε τιμές πραγματικού τύπου (π.χ. `double`), δεν μπορούμε να βάζουμε ισότητα.

Στο παράδειγμα της ελεύθερης πτώσης γράψαμε:

Απαίτηση: $(tP == \sqrt{2h/g}\,)$ && $(vP == -\sqrt{2gh}\,)$

αλλά πιο πολύ νόημα θα είχε αν γράφαμε:

Απαίτηση: $(tP \approx \sqrt{2h/g}\,)$ && $(vP \approx -\sqrt{2gh}\,)$

Αν θέλουμε να γράψουμε κάτι πιο ποσοτικό θα μπορούσαμε να βάλουμε φράγματα στο απόλυτο σφάλμα:

Απαίτηση: $(|tP - \sqrt{2h/g}\,| < \varepsilon_{TA})$ && $(|vP + \sqrt{2gh}\,| < \varepsilon_{VA})$

όπου ε_{TA} και ε_{VA} θετικοί αριθμοί, *φράγματα στο απόλυτο σφάλμα*. Τι σημαίνει; Η τιμή της *tP* πρέπει να διαφέρει από την ακριβή τιμή της $\sqrt{2h/g}$ λιγότερο από ε_{TA} και η τιμή της *vP* να διαφέρει από την ακριβή τιμή της, $-\sqrt{2gh}$, λιγότερο από ε_{VA}.

Παρομοίως, μπορούμε να βάλουμε φράγμα στο σχετικό σφάλμα:

Απαίτηση: $(|tP - \sqrt{2h/g}| < \varepsilon_{TR}\sqrt{2h/g})$ **&&** $(|vP + \sqrt{2gh}| < \varepsilon_{VR}\sqrt{2gh})$

Τα ε_{TA}, ε_{VA}, ε_{TR} και ε_{VR} εξαρτώνται όχι μόνο από τον τύπο κινητής υποδιαστολής που χρησιμοποιούμε αλλά και (κυρίως) από το πρόβλημα που λύνουμε. Στο παράδειγμά μας δεν έχει νόημα ακρίβεια μεγαλύτερη από τρία ψηφία. Αυτό μεταφράζεται σε $\varepsilon_{TR} = \varepsilon_{VR} = 5\times10^{-4}$. Φυσικά, ακόμη και ο `float` μας εγγυάται πολύ καλύτερη ακρίβεια.

Και τι θα κάνουμε εδώ; Θα συνεχίσουμε να δίνουμε παραδείγματα στον τύπο `double`, αλλά στις προδιαγραφές θα βάζουμε είτε φράγματα στο σφάλμα, όπως παραπάνω, είτε το "≈" αντί για "`==`". Εσύ θα πρέπει να θυμάσαι ότι *οι πραγματικοί τύποι θέλουν ιδιαίτερη προσοχή*.

3.9 Μια Απόδειξη με Πραγματικούς

Και τώρα να δούμε: Είναι σωστό το πρόγραμμά για την ελεύθερη πτώση; Θα πάρουμε ως προδιαγραφές:

Προϋπόθεση: $(g == 9.81)$ **&&** $(h \ge 0)$

Απαίτηση: $(tP \approx \sqrt{2h/g})$ **&&** $(vP \approx -\sqrt{2gh})$

Θα πρέπει δηλαδή να αποδείξουμε:

```
// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   tP = sqrt(2*h/g);
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Θα κάνουμε την απόδειξη από το τέλος προς την αρχή και ξεκινάμε από την:

```
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Ποια συνθήκη θα πρέπει να ισχύει πριν από την εκτέλεση της εντολής; Σύμφωνα με τη συνταγή μας, καταλήγουμε στην:

$$(tP \approx \sqrt{2h/g}) \;\&\&\; (-g\cdot tP \approx -\sqrt{2hg})$$

Φτάσαμε λοιπόν στο:

```
   tP = sqrt(2*h/g);
// (tP ≈ √(2h/g)) && (gTp ≈ √(2hg))
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Τι θα πρέπει να έχουμε πριν από την `tP = sqrt(2*h/g)`; Και πάλι σύμφωνα με τη συνταγή μας βρίσκουμε:

$$(sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

Εδώ όμως πρέπει να προσέξουμε και κάτι άλλο:

- έχουμε διαίρεση δια *g* και θα πρέπει να έχουμε $g \ne 0$,
- έχουμε τετραγωνικές ρίζες και για να υπολογίζονται θα πρέπει να έχουμε: $2h/g \ge 0$ και $2hg \ge 0$ (η μια από τις δύο φτάνει).

  Θα πρέπει λοιπόν να έχουμε:

$$(g \ne 0) \;\&\&\; (hg \ge 0) \;\&\&\; (sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

  Από την προϋπόθεση, $(g == 9.81)$ **&&** $(h \ge 0)$

- έπεται η $hg \ge 0$ και
- από τη $g == 9.81$ έπεται η $g \ne 0$

Αν πούμε ότι η *sqrt* υπολογίζει μια, όσο γίνεται καλύτερη, *προσέγγιση* της τετραγωνικής ρίζας, ισχύει και η $(sqrt(2h/g) \approx \sqrt{2h/g})$ && $(g \cdot sqrt(2h/g) \approx \sqrt{2hg})$.

Πρόσεξε ακόμη ότι αν δεν έχουμε τη g == 9.81 οι υπολογισμοί μας χάνουν το φυσικό τους νόημα, αφού τα αποτελέσματα που έχουμε από τη Φυσική θέλουν αυτήν την τιμή της g.

Προσοχή! Το αν θα ισχύει η $h \geq 0$ εξαρτάται από την καλή πρόθεση του χρήστη. Αργότερα θα μάθουμε πώς μπορούμε να κάνουμε *έλεγχο* της προϋπόθεσης και να ενεργήσουμε ανάλογα.

Αλλά τώρα μας έρχονται και άλλες ιδέες: ας πούμε ότι το h δεν είναι αρνητικό· μήπως υπάρχει περίπτωση να είναι πολύ μεγάλο και να έχουμε υπερχείλιση; Ας το σκεφτούμε. Για το h θα έχουμε: $0 \leq h \leq DBL\_MAX$. Όταν πάει να εκτελέσει την εντολή:

```
tP = sqrt(2*h/g);
```

για το πολλαπλασιασμό θα έχουμε:

$$0 \leq 2h \leq 2DBL\_MAX$$

Να μια περίπτωση υπερχείλισης[5] και μάλιστα χωρίς λόγο, διότι αν κάνουμε πρώτα τη διαίρεση δια g όλα πάνε καλά:

$$(2/g)h \leq (2/g)DBL\_MAX < DBL\_MAX, \text{ αφού } 2/g < 1$$

Θα πρέπει λοιπόν να υποχρεώσουμε, π.χ. με χρήση παρενθέσεων, τον υπολογιστή να κάνει τις πράξεις όπως εμείς θέλουμε:

```
tP = sqrt((2/g)*h);
```

Ύστερα από τα παραπάνω, η τιμή της *tP* θα είναι:

$$0 \leq tP == sqrt((2/g)h) \leq \sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Ερχόμαστε τώρα στην:

```
vP = -g*tP;
```

Από την σχέση για το *tP*, πολλαπλασιάζοντας επί g, παίρνουμε:

$$0 \leq g \cdot sqrt((2/g)h) \leq g\sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Αλλά, η $g\sqrt{2/g}\sqrt{DBL\_MAX} == \sqrt{2g}\sqrt{DBL\_MAX}$ είναι μικρότερη από *DBL_MAX* όταν έχουμε: $2g < DBL\_MAX$. Αυτό ισχύει για οποιονδήποτε τύπο `double`, αφού οι συνηθισμένες τιμές της *DBL_MAX* είναι της τάξης του 10^{308}. Άρα και η $-DBL\_MAX \leq g \cdot sqrt(2h/g) \leq DBL\_MAX$ ισχύει.

Βλέπουμε κοιπόν ότι η μελέτη της ορθότητας του προγράμματος μας οδηγεί σε διορθώσεις όπως αυτή της εντολής: `tP = sqrt(2*h/g)` που την αλλάξαμε σε: `tP = sqrt((2/g)/h)`. Και πρόσεξε και κάτι ακόμη: αν δεν γράφαμε τη `vP = -g*tP` για λόγους «οικονομίας», αλλά τη βάζαμε `vP = -sqrt(2*g*h)`, θα είχαμε και εδώ προβλήματα «υπερχείλισης χωρίς λόγο».

3.10 Τι Να Κάνουμε

Σε γενικές γραμμές έχουμε τρεις τρόπους για να αποδείξουμε την ορθότητα ενός προγράμματος. Συνήθως την αποδεικνύουμε πηγαίνοντας από το τέλος προς την αρχή. Αργότερα θα δούμε ότι και οι άλλοι δύο τρόποι είναι χρήσιμοι σε πολλές περιπτώσεις.

Τα εργαλεία που χρησιμοποιούμε στην απόδειξη είναι

- *αξιώματα* (όπως αυτό της εκχώρησης) και

[5] «Σιγά που θα έχουμε υπερχείλιση!» σκέφτεσαι «Για να γίνει κάτι τέτοιο θα πρέπει να έχουμε $h > DBL\_MAX/2$ και για τέτοιες τιμές δεν ισχύουν αυτοί οι τύποι.» Σωστό! Αλλά αυτά είναι δουλειά της Φυσικής· εδώ κοιτάμε μόνο τις προδιαγραφές μας.

- *συμπερασματικοί κανόνες* όπως αυτοί που είδαμε στο Κεφ. 0.

Ιδιαίτερη προσοχή θα πρέπει να δίνουμε σε «κακοτοπιές» όπως η υπερχείλιση, η κλήση συνάρτησης με όρισμα εκτός πεδίου ορισμού (π.χ. αρνητικό υπόριζο) και άλλα παρόμοια.

Θα πρέπει να αποδεικνύουμε την ορθότητα όλων των προγραμμάτων που γράφουμε; Κατ' αρχήν «Ναι!» αλλά αυτό δεν είναι και τόσο εύκολο. Αυτό που κάνουμε συνήθως είναι να επαληθεύουμε ορισμένα κρίσιμα κομμάτια του κάθε προγράμματος.

Ασκήσεις

Α Ομάδα

3-1 Απόδειξε ότι: Αν

```
   int k, sum;
```

τότε:

```
// sum == (k - 1)*k/2
   sum = sum + k;
   k = k + 1;
// sum == (k - 1)*k/2
```

3-2 Απόδειξε ότι: Αν

```
   double x, y, z;
```

τότε:

```
// z == y/2 - 1
   x = z + 1;   x = x + y/2;   z = z + 1;
// (x == y) && (z == y/2) }
```

Β Ομάδα

3-3 Αν, στο πρόβλημα της αντιμετάθεσης, ο τύπος των μεταβλητών είναι αριθμητικός, υπάρχει και άλλη λύση χωρίς βοηθητική μεταβλητή:

```
   a = a - b;  b = b + a;  a = b - a;
```

Απόδειξε ότι είναι σωστός. Αγνόησε την περίπτωση υπερχείλισης.

3-4 Απόδειξε ότι: Αν

```
   int d1, d2, q, r;
```

τότε:

```
// (r ≥ d2) && (r ≥ 0) && (d1 = q*d2 + r)
   q = q + 1;
   r = r - d2;
// (r ≥ 0) && (d1 == q*d2 + r)
```

3-5 Απόδειξε ότι: Αν

```
   int k, x, y;
```

τότε:

```
// true
   k = 0;  x = 1;  y = 1;
// (y = 2k+1) && (x == (k+1)²)
```

και

```
// (y == 2k+1) && (x == (k+1)²)
   k = k + 1;  y = y + 2;  x = x + y;
// (y == 2k+1) && (x == (k+1)²)
```

Γ Ομάδα

3-6 Τι σχέση υπάρχει μεταξύ (`double`) x και `static_cast<long>` (x); Διατύπωσε τις προδιαγραφές της συνάρτησης `static_cast<long>` (ή, τουλάχιστον, προσπάθησε).
Υπόδ.: Ξεχώρισε τις περιπτώσεις $x \geq 0$ και $x < 0$.

3-7 Δίνεται το πρόβλημα: Γράψε εντολές που να μεταθέτουν κυκλικώς τις τιμές τριών μεταβλητών a, b, c ($a \leftarrow b \leftarrow c \leftarrow a$). Διατύπωσε τις προδιαγραφές του και γράψε τις εντολές που το λύνουν μαζί με την απόδειξη ορθότητας.

κεφάλαιο

4

bool, char και Άλλοι Παρόμοιοι Τύποι

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσεις τις ιδιαιτερότητες των «ειδικών» ακέραιων τύπων `bool` και `char`. Μαζί με αυτούς θα δούμε και τους παρόμοιους *απαριθμητούς* τύπους.

Προσδοκώμενα αποτελέσματα:
Θα είσαι έτοιμος για αυτά που θα μάθεις στα επόμενα δύο κεφάλαια που στηρίζονται στον τύπο `bool`. Θα μπορείς να χειριστείς στα προγράμματά σου −εκτός από αριθμούς− και χαρακτήρες (και −αργότερα− κείμενα).

Έννοιες κλειδιά:

- *τύπος* `bool`
- *συνθήκες*
- *assert*
- *λογικές παραστάσεις*
- *τύποι* `char`, `signed char`, `unsigned char`
- *απαριθμητοί τύποι*
- *μετονομασία τύπου*
- `typedef`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Οι «Άλλοι» Ακέραιοι Τύποι:
Μέχρι τώρα γράφουμε τις συνθήκες επαλήθευσης σε σχόλια, για να παρακολουθούμε τη λογική του προγράμματός μας. Τώρα θα δούμε ότι μπορούμε:

- να τυπώνουμε την τιμή (1 για `true` ή 0 για `false`) μιας συνθήκης, με μια `"cout << …"`,
- να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή,

και (στο επόμενο κεφάλαιο):

- να ρυθμίζουμε την εκτέλεση του προγράμματός μας με βάση τις τιμές συνθηκών.

Αυτά έχουν να κάνουν με τον τύπο `bool`.
Θα δούμε ακόμη τους τύπους `char` και `wchar_t`.

Στο Κεφ. 2 είπαμε ότι όλοι αυτοί είναι ακέραιοι τύποι. Τι θα πει αυτό; Ας πούμε ότι έχουμε:

```
bool b( true );
char c1( ' ' ), c2( '!' ), c3;
int j;
```

και δίνουμε:

```
j = 32*b;
c3 = c1 + c2;
cout << j << "  " << c3 << endl;
```

Αποτέλεσμα:

```
32  A
```

Στη συνέχεια θα καταλάβεις πώς βγαίνουν αυτά τα αποτελέσματα αλλά και γιατί δεν πρέπει να γράφεις τέτοιες εντολές!

Πριν αρχίσεις τη μελέτη της §4.1 ρίξε μια ματιά στο Παρ. Α και μη διστάζεις να το συμβουλεύεσαι όσο θα μελετάς τις τρεις πρώτες παραγράφους.

4.1 Οι Συνθήκες στο Πρόγραμμα

Σε όσα είπαμε μέχρι τώρα είδαμε και μερικές συνθήκες, π.χ.:

$$(number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

$$(a == a_0) \;\&\&\; (b == b_0) \;\&\&\; (|a_0 + b_0| \le ½)$$

$$(|tP - \sqrt{2h/g}\,| < \varepsilon_{TR}\sqrt{2h/g}\,) \;\&\&\; (|vP + \sqrt{2gh}\,| < \varepsilon_{VR}\sqrt{2gh}\,)$$

Σε όλες τις περιπτώσεις (εκτός από την πρώτη) οι συνθήκες που είδαμε είναι συγκρίσεις που συνδέονται μεταξύ τους με λογικές πράξεις. Όποτε βάλαμε τις συνθήκες μέσα στο πρόγραμμά μας ήταν σε σχόλια. Η C++ μπορεί να «κατανοήσει» και να χρησιμοποιήσει μια συνθήκη αν γραφεί σύμφωνα με τους κανόνες που βάζει η γλώσσα:

- Στις συγκρίσεις, αντί των συμβόλων που χρησιμοποιούν τα μαθηματικά, χρησιμοποίησε αυτά που θέλει η C++. Δες το Πλ. 4.1.
- Στις λογικές πράξεις γράψε τους τελεστές όπως δίνονται στο Πλ. 4.2 αλλά πρόσεξε μερικές «επιφυλάξεις» που παραθέτουμε στη συνέχεια. Μπορείς να χρησιμοποιείς τα "`&&`", "`||`", "`!`", αντί των "$\wedge$", "$\vee$", "$\neg$" αντιστοίχως, αλλά αντί του "$\Leftrightarrow$" θα χρησιμοποιείς το "`==`", αντί του "$\Rightarrow$" το "`<=`" και αντί του "`xor`" το "`!=`". Ο Πίν. 4-1 είναι στην πραγ-

Τιμές Ορισμάτων		Αποτελέσματα Πράξεων					
P	Q	!P (not P)	P && Q (P and Q)	P || Q (P or Q)	P <= Q (P ⇒ Q)	P == Q (P ⇔ Q)	P != Q (P xor Q)
false	false	true	false	false	true	true	false
false	true	true	false	true	true	false	true
true	false	false	false	true	false	false	true
true	true	false	true	true	true	true	false

Πίν. 4-1 Οι πίνακες αλήθειας όλων των λογικών πράξεων που μας δίνει η C++.

Πλαίσιο 4.1

Τελεστές Συσχέτισης (Σύγκρισης)

Μαθηματικά	C++	Όνομα και σημασία του τελεστή συσχέτισης
=	`==`	ίσο με (π.χ. `x == 5`)
≠	`!=`	διάφορο
<	`<`	μικρότερο (π.χ. `x < 12`)
>	`>`	μεγαλύτερο
≤	`<=`	μικρότερο ή ίσο
≥	`>=`	μεγαλύτερο ή ίσο

ματικότητα σύντμηση των πινάκων του Παρ. Α.

- Αντικατάστησε τις διπλές (ή πολλαπλές) συγκρίσεις με απλές,

  π.χ. *μην γράφεις* *INT_MIN ≤ number ≤ INT_MAX*

  αλλά `INT_MIN <= number && number <= INT_MAX`

Η C++ σου επιτρέπει αντί των "`&&`", "`||`", "`!`" να χρησιμοποιείς τα "`and`", "`or`", "`not`" αντιστοίχως. Εμείς θα χρησιμοποιούμε εδώ τα "`&&`", "`||`", "`!`" μια και είναι αυτά που χρησιμοποιούνται συνήθως (από τον καιρό της C) και αυτά θα συναντάς όποτε διαβάζεις άλλα βιβλία. Άλλωστε, είναι πολύ πιθανό, η C++ που χρησιμοποιείς (π.χ. η Borland C++) να μη δέχεται τα "`not_eq`" (αντί για το "`!=`"), "`and`", "`or`", "`not`".

Τα "`==`", "`<`", "`>`", "`<=`", "`>=`", "`!=`", "`not_eq`", "`&&`", "`and`", "`||`", "`or`", "`!`", "`not`" είναι λεξικά σύμβολα (tokens) της C++ και δεν θα πρέπει να διαχωρίζεις τους χαρακτήρες τους, με οποιονδήποτε τρόπο, όταν τα γράφεις ενώ από την άλλη θα πρέπει να τα διαχωρίζεις από τα προηγούμενα και τα επόμενά τους.

Οι **λογικές παραστάσεις** (logical expressions), που η τιμή τους είναι "`true`" ή "`false`", κατασκευάζονται με κανόνες παρόμοιους με αυτούς που είδαμε για τις αριθμητικές παραστάσεις. Μπορούμε να πούμε, ότι μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές ("`true`" ή "`false`") και από συσχετίσεις (π.χ. συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους λογικούς τελεστές.

Στο Πλ.4.2 βλέπεις ένα συντακτικό κανόνα *για τις παρενθέσεις*. Μας λέει ότι μπορούμε να απλουστεύουμε τη γραφή λογικών παραστάσεων παραλείποντας παρενθέσεις.

Δηλαδή: *Μπορείς να γράψεις*

```
!(a > 0) && b <= 0 || c == 1
```

που είναι το ίδιο με το

```
((!(a > 0)) && (b <= 0)) || (c == 1)
```

Πάντως η δεύτερη μορφή είναι προτιμότερη.

Αν όμως θέλεις να γράψεις

```
(!(a > 0) && b <= 0) xor (c == 1)
```

θα *πρέπει να κρατήσεις τις παρενθέσεις*. Διότι αν γράψεις:

```
!(a > 0) && b <= 0 != c == 1
```

θα υπολογιστεί η `!(a > 0) && (((b <= 0) != c) == 1)` (δες τον Πίν. 4-2).

Ο ίδιος κανόνας υπάρχει, με άλλον τρόπο, και στον πίνακα του Παρ. Ε, και μας λέει ότι:

- πρώτα υπολογίζονται οι αρνήσεις "`!`" ("`not`") (προτ. 3),
- υπολογίζονται οι αριθμητικές παραστάσεις,
- στη συνέχεια γίνονται οι συγκρίσεις, πρώτα οι "`<`", "`<=`", "`>`", "`>=`" (προτ. 5) και μετά οι "`==`", "`!=`" (προτ. 6),

Πλαίσιο 4.2

Λογικοί Τελεστές

Λογική	C++
∧, **and**, **και**	`&&` `and`
∨, **or**, **ή**	`||` `or`
¬, ~, **όχι**	`!` `not`
⊻, **xor**	`!=`
⇒, ⊃	`<=`
⇔	`==`

Συντακτικός κανόνας:

- ***Στις λογικές πράξεις επιτρέπεται να μη βάλεις παρενθέσεις, αλλά να ξέρεις ότι οι πράξεις γίνονται με την εξής σειρά: πρώτα οι not (!), μετά οι and (&&), και τέλος οι or (||). Για τις "!=" (xor), "<=" (⇒) και "==" (⇔) χρειάζονται παρενθέσεις.***

- μετά γίνονται οι λογικές πράξεις με τη σειρά: "`&&`" ("`and`") (προτ. 10) και "`||`" ("`or`")[1] (προτ. 11),
- τέλος γίνονται οι εκχωρήσεις.

Μερικά παραδείγματα λογικών παραστάσεων:

```
(x > 0) || (y > 0)
(x > 0) == (y > 0)
(x < 0) == (y < 0)
(k <= l) && (l < m + 5)
(a – 2 > b) && (b < c) || (c == 0)
(a – 2 > b) && ((b < c) || (c == 0))
```

Βέβαια η τιμή όλων αυτών των παραστάσεων, `true` ή `false`, εξαρτάται από τη συγκεκριμένη τιμή των λογικών ορισμάτων. Αν π.χ. υποθέσουμε ότι τα αριθμητικά ορίσματα (τύπου `int`) έχουν τιμές: $x = 5$, $y = 4$, $a = -1$, $b = c = 0$, $k = -2$, $l = 1$ και $m = -4$, τότε οι προηγούμενες λογικές παραστάσεις θα πάρουν τις εξής τιμές:

`(x > 0) || (y > 0)`: `true` (επειδή $x > 0$ και $y > 0$),
`(x > 0) == (y > 0)`: `true` (επειδή $(x > 0)$ `true` και $(y > 0)$ `true`),
`(x < 0) == (y < 0)`: `true` (επειδή $(x < 0)$ `false` και $(y > 0)$ `false`),
`(k <= l) && (l < m + 5)`: `false` (επειδή: $(k <= l)$ `true` ενώ $(l < m + 5)$ `false`),
`(a – 2 > b) && (b < c) || (c == 0)`: `true` (επειδή c `==` 0),
`(a – 2 > b) && ((b < c) || (c == 0))`: `false` (επειδή $(a - 2 > b)$ `false`).

Πρόσεξε τη 2η και την 3η: το `==` υλοποιεί την αμοιβαία συνεπαγωγή (⇔). Και οι δύο παραστάσεις έχουν τιμή `true` διότι και στις δύο περιπτώσεις οι ανισότητες έχουν την ίδια τιμή: στη 2η και οι δύο είναι `true`, στην 3η και οι δύο είναι `false`.

Πρόσεξε ακόμη την 5η και την 6η:

- Αφού η `&&` υπολογίζεται πριν από την `||`, η 5η ισοδυναμεί με: `((a – 2 > b) && (b < c)) || (c == 0)` και επειδή `true ||` οτιδήποτε μας κάνει `true` και η `c == 0` είναι `true`, όλη η παράσταση είναι `true`.
- Στην 6η, με τις παρενθέσεις ζητούμε να υπολογιστεί πρώτα η `||` και μετά η `&&`. Επειδή `false &&` οτιδήποτε μας κάνει `false` και η `a – 2 > b` είναι `false`, όλη η παράσταση είναι `false`.

[1] Εδώ υπάρχει ένα λεπτό σημείο που θα το δούμε στο επόμενο κεφάλαιο.

Παίρνοντας υπόψη την προτεραιότητα των πράξεων μπορούμε να βγάλουμε μερικές παρενθέσεις και τα παραδείγματά μας να γραφούν:

```
x > 0 || y > 0
x > 0 == y > 0
x < 0 == y < 0
k <= l && l < m + 5
a - 2 > b && b < c || c == 0
a - 2 > b && (b < c || c == 0)
```

Και τώρα πρόσεξε:

- ***Σε μια εντολή εξόδου (`cout << …`) μπορείς να βάζεις ως όρισμα μια συνθήκη, μέσα σε παρενθέσεις. Αυτό που θα γίνει είναι το εξής: θα υπολογισθεί η τιμή της και αν είναι "`true`" (αν ισχύει) θα τυπωθεί η τιμή "1" (ένα), ενώ αν είναι "`false`" (αν δεν ισχύει) θα τυπωθεί η τιμή "0" (μηδέν).***

Με άλλα λόγια, αν *Π* μια λογική παράσταση τότε η εντολή:

```
cout << (Π) …
```

ισοδυναμεί με:

```
cout << 1 …
```
αν η *Π* έχει τιμή `true`, και με
```
cout << 0 …
```
αν η *Π* έχει τιμή `false`.

Παράδειγμα

Αν έχουμε δηλώσει:

```
int  n1, n2;
```

τότε οι εντολές:

```
n1 = 12;  n2 = 13;
cout << n1 << " < " << n2 << " : " << (n1 < n2) << endl;
cout << n1 << " > " << n2 << " : " << (n1 > n2) << endl;
```

θα δώσουν:

```
12 < 13 : 1
12 > 13 : 0
```

Στην επόμενη παράγραφο βλέπεις πώς χρησιμοποιούμε αυτή τη δυνατότητα για να ελέγχουμε την ορθότητα ενός προγράμματος.

Προς το παρόν όμως δες κάτι που μπορεί να σου συμβεί: Είπαμε πιο πάνω ότι για να «κατανοήσει» η C++ μια λογική παράσταση πρέπει να αντικαταστήσεις «*τις διπλές (ή πολλαπλές) συγκρίσεις με απλές*», π.χ. αντί για `0 <= x < 10` γράψε `(0 <= x) && (x < 10)`. Αν δεν το κάνεις, ο μεταγλωττιστής θα δεχθεί τη διπλή σύγκριση αλλά θα την «κατανοήσει» ως `(0 <= x) < 10` που έχει πάντοτε τιμή `true` διότι το `(0 <= x)` είναι μικρότερο από 10 είτε έχει τιμή `true` (1) είτε έχει τιμή `false` (0).

4.1.1 Το Λάθος που θα Κάνεις Συχνά!

Ο τίτλος της παραγράφου είναι απαισιόδοξος, αλλά είναι παρατηρημένο: οι αρχάριοι στη C++ (και στη C) κάνουν το ίδιο *σοβαρότατο* σφάλμα: προσπαθούν (μάταια) να συγκρίνουν για ισότητα με το "`=`" αντί για το "`==`". Κοίταξε όμως τι μπορεί να συμβεί:

Ας πούμε ότι έχεις: *a* `==` –1, *b* `==` *c* `==` 0 και θέλεις να τυπώσεις την τιμή της παράστασης:

```
(a – 2 > b) && (b < c) || (c == 0)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 0)) << endl;
```

Αποτέλεσμα: **0** (**false**)! Πώς έγινε αυτό; Η C++ υπολογίζει την τιμή της *εκχώρησης* [2] **c = 0** που είναι μηδέν και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 0 ως **false**.

Ας πούμε τώρα ότι θέλεις την τιμή της παράστασης

```
(a – 2 > b) && (b < c) || (c == 44)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 44)) << endl;
```

Αποτέλεσμα: **1** (**true**)! Εδώ τι έγινε; Η C++ υπολογίζει την τιμή της *εκχώρησης* **c = 44** που είναι 44 και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 44 ≠ 0 ως **true**. Εκτός από αυτό όμως, η *c*, για το υπόλοιπο πρόγραμμά σου, έχει τιμή 44 αντί για 0 που θα ήθελες να έχει!

Και στις δύο περίπτώσεις η Borland C++ θα σου δώσει *προειδοποίηση*: «**Possibly incorrect assignment**» (πιθανόν εσφαλμένη εκχώρηση). Καλό είναι λοιπόν *να προσέχεις*, όχι μόνον τα *λάθη*, *αλλά* και *τις προειδοποιήσεις*.

4.2 Οι Τιμές των Συνθηκών Επαλήθευσης

Αν ένα πρόγραμμά μας είναι σωστό τότε, κάθε φορά που εκτελείται, όλες οι συνθήκες επαλήθευσης θα (πρέπει να) έχουν τιμή **true**. Αν αυτό δεν συμβαίνει τότε κάτι δεν πάει καλά.

Όπως καταλαβαίνεις, σύμφωνα με όσα είπαμε στην προηγούμενη παράγραφο μπορούμε να ζητήσουμε από τον υπολογιστή να υπολογίσει και να μας τυπώσει τις τιμές των συνθηκών επαλήθευσης.

Δες ένα παράδειγμα με το πρόγραμμα της αντιμετάθεσης:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   S = a;
// (b == b0) && (S == a0)
   cout << "   1: " << ((b == b0) && (S == a0)) << endl;
   a = b;
// (a == b0) && (S == a0)
   cout << "   2: " << ((a == b0) && (S == a0)) << endl;
   b = S;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Πρόσεξε τα εξής:

- Κάτω από κάθε σχόλιο με συνθήκη επαλήθευσης υπάρχει μια "**cout << …**" που τυπώνει την τιμή της συνθήκης. Οι εκτυπώσεις αρχίζουν με ένα χαρακτηριστικό: « **Προ:**» για την προϋπόθεση, « **Απτ:**» για την απαίτηση και αριθμούς 1, 2, 3,... για τις ενδιάμεσες συνθήκες.

[2] «τιμή της *εκχώρησης*»!; Ναι, υπάρχει τέτοιο πράγμα και θα το δούμε αργότερα· προς το παρόν: είναι η τιμή που αποθηκεύεται στη μεταβλητή.

- Πρόσεξε πώς έγινε η μετάφραση των συνθηκών σε C++, σύμφωνα με τους κανόνες που είπαμε.

  Να ένα παράδειγμα εκτέλεσης:

```
Προ: 1
  1: 1
  2: 1
Απτ: 1
a = 20   b = 10
```

  Ας δούμε τώρα το πρώτο, εσφαλμένο, πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   a = b;  b = a;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Παράδειγμα εκτέλεσης:

```
Προ: 1
Απτ: 0
a = 20   b = 20
```

Προσοχή! Να υπενθυμίσουμε ότι με τύπους κινητής υποδιαστολής μπορεί να έχεις και μερικές εκπλήξεις όταν οι συνθήκες σου έχουν ισότητες[3].

Βέβαια, θα πρέπει να (ξανα)τονίσουμε ότι το να πάρεις μερικά παραδείγματα εκτέλεσης με όλες τις συνθήκες `true` δεν σημαίνει ότι το πρόγραμμά σου είναι σωστό.

4.3 Έλεγχος Συνθηκών Επαλήθευσης: *assert()*

Στο επόμενο κεφάλαιο θα μάθουμε πώς να ελέγχουμε τη ροή της εκτέλεσης του προγράμματος. Προς το παρόν όμως θα δούμε έναν τρόπο για να αποφασίζουμε αν θα *διακόψουμε* την εκτέλεση του προγράμματος σε σχέση με τις τιμές των συνθηκών επαλήθευσης.

Δες έναν άλλον τρόπο για να γράψουμε το δεύτερο πρόγραμμα της προηγούμενης παραγράφου:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   a = b;  b = a;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

[3] Για δοκίμασε να κάνεις τα ίδια με το πρόγραμμα της ελεύθερης πτώσης. Χα!

Η εκτέλεση του προγράμματος δίνει:

```
Assertion failed: a == b0 && b == a0, file test02a.cpp, line 13
```

Τι είναι αυτή η «παράξενη» εντολή "`assert(`*συνθήκη*`)`"; Να τη σκέφτεσαι ως κλήση συνάρτησης, όπως αυτές που μάθαμε στο Κεφ. 1, που όμως:

- παίρνει όρισμα μια *συνθήκη* (ή μια τιμή τύπου `bool`, όπως θα μάθουμε στην επόμενη παράγραφο) και
- *δεν επιστρέφει* κάποια *τιμή*.

Για τέτοιες συναρτήσεις θα μιλήσουμε αργότερα.

Για να χρησιμοποιήσεις την *assert* θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <cassert>`".

Ας δούμε τη λειτουργία της:

- Την πρώτη φορά που καλείται, στη γραμμή 11, η συνθήκη-όρισμα ισχύει (`true`). Ως προς τα αποτελέσματα: είναι σαν να μην υπάρχει.
- Τη δεύτερη φορά καλείται, στη γραμμή 13, η συνθήκη-όρισμα δεν ισχύει (`false`). Ως αποτέλεσμα διακόπτεται η εκτέλεση του προγράμματος και παίρνουμε αυτά που είδες.

Ξαναγράφουμε και το πρώτο πρόγραμμα ως εξής:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   S = a;
   assert( (b == b0) && (S == a0) );
   a = b;
   assert( (a == b0) && (S == a0) );
   b = S;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

Αποτέλεσμα:

```
 a = 20   b = 10
```

Καμιά ένδειξη για την ύπαρξη της *assert*![4]

Τέλος, ας δούμε πώς μπορούμε να χρησιμοποιήσουμε την *assert* για τη διασφάλιση της προϋπόθεσης του προγράμματος της ελεύθερης πτώσης (§2.7). Στο αρχείο `eleptw.cpp` έχουμε το πρόγραμμά μας που τώρα, με την εισαγωγή της "`assert( h >= 0 )`" στη 14η γραμμή, έχει την εξής μορφή:

```
#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
```

[4] Παρ' όλα αυτά η εκτέλεση του προγράμματος γίνεται πιο αργή. Θα δούμε αργότερα ότι αυτό διορθώνεται.

```
// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
   tP = sqrt( 2*h/g );
   vP = -sqrt(2*h*g);
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
// Τύπωσε τα tP, vP
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << vP << " m/sec" << endl;
}
```

Δες ένα παράδειγμα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: -80
Assertion failed: h >= 0, file eleptw.cpp, line 14
```

Φυσικά, αν δίναμε αρχικό ύψος **80** το αποτέλεσμα θα ήταν ακριβώς αυτό που ξέραμε από το αρχικό πρόγραμμα.

4.4 Ο Τύπος bool

Ας δούμε τώρα πώς μπορούμε «*να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή*», όπως λέγαμε στην §4.1.

Στα προηγούμενα κεφάλαια γνωρίσαμε τον τρόπο γραφής και δήλωσης δύο τύπων στοιχείων: **int** (ακέραιοι) και **double** (πραγματικοί). Στην παράγραφο αυτή θα γνωρίσουμε έναν τρίτο τύπο στοιχείων, που λέγεται τύπος **bool**. Οι τιμές του τύπου **bool** είναι οι **false** και **true** και είναι διαταγμένες:

```
false < true
```

Μεταξύ τιμών τύπου **bool** μπορεί να γίνονται οι λογικές πράξεις που μάθαμε: η "**&&**", η "**||**", η "**!**". Μα είπαμε ότι μπορεί να γίνονται και οι άλλες: ⇒, ⇔, **xor**! Ναι, αλλά να καταλάβουμε τι συμβαίνει: η υλοποίησή τους στηρίζεται στη διάταξη που δώσαμε παραπάνω. Πήγαινε στο Παρ. Α και

- Δες τον αληθοπίνακα της "⇒" (Πίν. Α-3)· στις γραμμές 1, 2 και 4 για τις οποίες η $P \Rightarrow Q$ έχει τιμή **true**, με βάση τη διάταξη **false** < **true**, ισχύει η P **<=** Q. Στη γραμμή 3, όπου η $P \Rightarrow Q$ έχει τιμή **false**, έχουμε $P > Q$ ή αλλιώς **!**(P **<=** Q).
- Δες τον αληθοπίνακα της "**xor**" (Πίν. Α-4)· στις γραμμές 2 και 3 για τις οποίες η P **xor** Q έχει τιμή **true**, έχουμε P **!=** Q, ενώ στις 1 και 4 όπου η P **xor** Q έχει τιμή **false**, P **==** Q ή **!**(P **!=** Q).
- Τέλος, δες τον αληθοπίνακα της "⇔" (Πίν. Α-5)· στις γραμμές 1 και 4 για τις οποίες η $P \Leftrightarrow Q$ έχει τιμή **true**, έχουμε P **==** Q, ενώ στις 2 και 3 όπου η $P \Leftrightarrow Q$ έχει τιμή **false**, P **!=** Q ή **!**(P **==** Q).

Μπορούμε λοιπόν να συμπληρώσουμε αυτό που είπαμε πιο πάνω: μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές (**true** ή **false**), μεταβλητές τύπου **bool** και από συσχετίσεις (ή συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους τελεστές "**&&**", "**||**", "**!**", "**<=**", "**==**" και "**!=**". Ακόμη, σε μια μεταβλητή τύπου **bool** μπορείς να αποθηκεύεις την τιμή μιας συνθήκης (λογικής παράστασης).

Οι μεταβλητές τύπου **bool** δηλώνονται στο πρόγραμμα, όπως και οι μεταβλητές των άλλων τύπων που γνωρίσαμε, στο μέρος δήλωσης μεταβλητών. Π.χ.:

```
   bool   x, y, logical, signal, ok, lgc;
   int    i, k;
```

```
   double a;
```

Η παραπάνω δήλωση σημειώνει ότι κατά την εκτέλεση του προγράμματος οι μεταβλητές *x*, *y*, *logical*, *signal*, *ok* και *lgc* θα παίρνουν τις τιμές **true** ή **false**. Έτσι π.χ., μπορούμε να γράψουμε μέσα στο πρόγραμμα τις παρακάτω εντολές εκχώρησης, όπου η δεξιά πλευρά είναι μια λογική παράσταση (ή μια λογική σταθερά), ενώ η αριστερή πλευρά είναι μια μεταβλητή τύπου **bool**.

```
    x = true;                       y = !signal || x;
    logical = (x || y) && signal;   ok = (a <= 18) && (k != 2);
    lgc = false;                    signal = k < -1;
```

Παράδειγμα

Το παρακάτω πρόγραμμα τυπώνει το περιεχόμενο του Πίν. 4-1, μόνο που αντί για **false** και **true** έχει 0 και 1 αντίστοιχα.

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q" << endl;
   P = false;  Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = false;  Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
}
```

Αποτέλεσμα:

```
P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q
0   0   1   0    0    1    1    0
0   1   1   0    1    1    0    1
1   0   0   0    1    0    0    1
1   1   0   1    1    1    1    0
```

Αφού όταν ζητάμε να τυπωθεί **true** ή **false** παίρνουμε 1 ή 0 αντιστοίχως καταλαβαίνεις και γιατί (στο πρόγραμμα της εισαγωγής) το **32*true** μας δίνει 32! Ο **bool** είναι ένας ακέραιος τύπος με δύο τιμές 0 και 1. Αυτό θα *πρέπει να προσπαθείς να το ξεχάσεις* αν και θα υπάρχουν πολλά για σου το θυμίζουν.

- Να θυμάσαι μόνον ότι **false < true** (και όχι τα 0 και 1).
- Να θυμάσαι ότι **!false == true** και **!true == false** (και όχι ότι **1-false == true** και **1-true == false**).

Να δούμε τώρα τις κληρονομιές από τη C: Στη C δεν υπάρχει τύπος **bool** και *οποιαδήποτε τιμή δεν είναι μηδέν θεωρείται ως "***true***", ενώ το "***0***" θεωρείται ως "***false***"*. Η C++ είναι συμβατή με αυτά. Αν έχεις δηλώσει:

```
   bool b;
```

τότε οι εντολές:

```
   b = static_cast<bool>( 37.5 );  cout << b << "  ";
```

```
   b = 37.5;                        cout << b << "  ";
   b = static_cast<bool>( 0 );      cout << b << "  ";
   b = 0;                           cout << b << endl;
```

δίνουν:

```
1  1  0  0
```

Δηλαδή, η μη μηδενική τιμή (τύπου **double**) γίνεται **true** με ανοικτή ή εννοούμενη τυποθεώρηση ενώ το 0 γίνεται **false**.

Καλώς ή κακώς, εργαλεία από τη C θα χρησιμοποιούμε συχνά και δεν γίνεται να ξεφύγουμε από χρήσεις μη λογικών τιμών σε θέσεις όπου περιμένουμε τιμή τύπου **bool**. Εμείς πάντως θα αποφεύγουμε παρόμοιες χρήσεις. Όσο είναι δυνατό.

Σημείωση: ▶

Για τις μεταβλητές τύπου **bool** θα δεις πολύ συχνά τον όρο **σημαία** (flag) που μπορεί να είναι *ανεβασμένη* (**true**) ή *κατεβασμένη* (**false**). ◀

4.4.1 Για να Γράφουμε "false" και "true"

Αν τα "**0**" και "**1**" σε ενοχλούν πολύ και θέλεις να γράφεις "**false**" και "**true**" μπορείς να στείλεις προς το *cout* το μήνυμα "**boolalpha**" και όλα θα αλλάξουν με μιας: Το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
  cout << false << "  " << true << endl;
  cout << boolalpha;
  cout << false << "  " << true << endl;
}
```

θα δώσει:

```
0  1
false  true
```

4.5 Οι Τύποι char

Στην §2.3 μεταξύ των άλλων ακέραιων τύπων είδαμε και τον τύπο **char**. Στην πραγματικότητα η C++ έχει τρεις διαφορετικούς τύπους:

- **char**,
- **signed char**,
- **unsigned char**.

Ας δούμε ένα μικρό αλλά χαρακτηριστικό

Παράδειγμα 1 ➴

```
#include <iostream>
using namespace std;
int main()
{
   char c;     unsigned char u;     signed char s;

   c = 195;  u = 195;  s = 195;
   cout << c << "  " << u << "  " << s << endl;
   cout << static_cast<int>(c) << "  " << static_cast<int>(u)
        << "  " << static_cast<int>(s) << endl;
}
```

Αποτέλεσμα:

```
Γ  Γ  Γ
-61  195  -61
```

Ας τα πάρουμε από την αρχή:

- Με τις εντολές: `c = 195;  u = 195;  s = 195` δίνουμε και στις τρεις μεταβλητές την ίδια ακέραιη τιμή: 195. Να τονίσουμε βέβαια ότι αυτές οι εντολές εκτελούνται ως:

```
c = static_cast<char>(195);
u = static_cast<unsigned char>(195);
s = static_cast<signed char>(195) .
```

- Το αποτέλεσμα της `cout << c << "  " << u << "  " << s << endl` είναι λίγο αναπάντεχο: `Γ  Γ  Γ`. Πώς εξηγείται; Στο Παρ. Δ, Πίν. Δ-4 (Δ-3) μπορείς να δεις ότι στο σύνολο χαρακτήρων για τα Windows, στη θέση 195, υπάρχει το ελληνικό γράμμα Γ.
- Η δεύτερη γραμμή είναι εντελώς ανεξήγητη για όποιον δεν ξέρει αριθμητική modulo (mod 256 στην περίπτωσή μας). Απλώς παρατήρησε ότι 61 = 256 – 195.

&

Τώρα, ξεχνούμε για λίγο το ότι βάλαμε τον `char` στους ακέραιους τύπους και ξεκινούμε από το εξής: *στον τύπο* `char` *παριστάνουμε χαρακτήρες*· όλους τους χαρακτήρες που περιλαμβάνει το σύνολο χαρακτήρων του υπολογιστή.

Παράδειγμα 2

Μέσα στο πρόγραμμά μας μπορεί να δηλώσουμε:

```
const char plus = '+', space = ' ', aLower = 'a', aUpper = 'A';
char letter, digit, spcChar;
```

και στη συνέχεια να δώσουμε:

```
letter = aLower;
digit = '7';
spcChar = plus;
cout << ' ' << letter << space << digit << spcChar << endl;
```

Αποτέλεσμα:

```
 a 7+
```

ή, για να το δεις καλύτερα:

```
␣a␣7+
```

&

Οι σταθερές τύπου `char` έχουν τη μορφή που βλέπεις στα παραδείγματα: ένας χαρακτήρας μεταξύ δύο αποστρόφων:

```
'+'   ' '   'a'   'A'   '7'
```

Όπως είναι φανερό, έχεις πρόβλημα να παραστήσεις την απόστροφο· η λύση είναι αυτή που λέγαμε και για τους ορμαθούς χαρακτήρων: αν θέλεις να γράψεις σταθερές τύπου `char` που έχουν ως τιμή κάποιον από τους χαρακτήρες: `\`, `?`, `'`, `"` θα πρέπει να γράψεις: `'\\'`, `'\?'`, `'\''`, `'\"'`. Με παρόμοιο τρόπο (Πίν. 4-2) μπορείς να παραστήσεις και *μη εκτυπώσιμους* χαρακτήρες.

Στο παράδειγμά μας είδαμε ότι σε μια μεταβλητή τύπου `char` μπορούμε να δίνουμε τιμές με εντολές εκχώρησης. Μπορούμε να διαβάζουμε και από το πληκτρολόγιο; Βεβαίως.

LF	`'\n'`
HT	`'\t'`
VT	`'\v'`
BS	`'\b'`
CR	`'\r'`
FF	`'\f'`
BEL	`'\a'`
\	`'\\'`
?	`'\?'`
'	`'\''`
"	`'\"'`

Πίν. 4-2 Πώς γράφουμε ως σταθερές τύπου `char` μερικούς χαρακτήρες εκτυπώσιμους και μη. Για τους μη εκτυπώσιμους δες το Παρ. D.

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3;
   :
cin >> c1 >> c2 >> c3;
cout << c1 << c2 << c3 << endl;
```

Πληκτρολογούμε:

a b c<Enter>

και παίρνουμε:

```
abc
```

Θαυμάσια! Αλλά… Δεν είπαμε ότι το κενό (διάστημα) είναι χαρακτήρας; Γιατί δεν έγινε τιμή της *c2*; Γι' αυτό φταίει ο "`>>`" του *cin* που «τρώει» τα διαστήματα, τις αλλαγές γραμμής και άλλα παρόμοια. Αν θέλεις να διαβάζεις τα πάντα χρησιμοποίησε τη

```
cin.get(a);
```

που διαβάζει από το πληκτρολόγιο έναν χαρακτήρα και τον αποθηκεύει ως τιμή της *a*, τύπου `char`. Αλλά εδώ προσοχή: *πρέπει να διαβάζεις τα πάντα*!

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3, c4;

cin.get(c1);  cin.get(c2);
cin.get(c3);  cin.get(c4);
cout << " c1: " << c1 << endl;
cout << " c2: " << c2 << endl;
cout << " c3: " << c3 << endl;
cout << static_cast<int>(c4) << endl;
```

Πληκτρολογούμε:

```
a b<enter>
```

και παίρνουμε:

```
 c1: a
 c2:
 c3: b
10
```

Στη δεύτερη γραμμή έχουμε στην πραγματικότητα: "`c2: ␣`". Να το διάστημα που λέγαμε. Το 10 στην τέταρτη γραμμή είναι αποτέλεσμα της: `cout << static_cast<int>(c4)`. Στη *c4* αποθηκεύτηκε το <Enter> (LF, '`\n`') που δώσαμε στο τέλος της γραμμής!

Από το τελευταίο παράδειγμα θα πρέπει να καταλαβαίνεις ότι αν στο ίδιο πρόγραμμα ανακατέψεις "`cin >> …`" και "`cin.get(v)`" θα έχεις προβλήματα.

- ***Σε ένα πρόγραμμα θα δουλεύεις είτε με "`cin >> …`" είτε με "`cin.get(v)`". Στη δεύτερη περίπτωση θα παίρνεις ό,τι στέλνει το πληκτρολόγιο προς το πρόγραμμά σου. Για να τις ανακατέψεις θα πρέπει να ξέρεις πώς ακριβώς δουλεύει η "`cin >> …`".***

4.5.1 Το Σύνολο Χαρακτήρων και οι Τύποι `char`

Η θέση του χαρακτήρα μέσα στο σύνολο χαρακτήρων τον καθορίζει απόλυτα. Π.χ. στη θέση 65 του ASCII υπάρχει το κεφαλαίο γράμμα '`A`' του λατινικού αλφαβήτου.

Οι γλώσσες προγραμματισμού βάζουν μερικές, πολύ χαλαρές, προδιαγραφές στα σύνολα χαρακτήρων που χρησιμοποιούν:

- Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη.
- Αν τα πεζά γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.
- Αν τα κεφαλαία γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.

Αν έχουμε μια τιμή c τύπου `unsigned char`, με τυποθεώρηση με τη `static_cast<int>` μπορούμε να βρούμε τη θέση (order) της στο σύνολο χαρακτήρων. Π.χ. οι:

```
unsigned char c;

c = 'D';  cout << static_cast<int>(c);
```

```
c = 'Γ';  cout << "   " << int(c);
cout << "   "
 << static_cast<int>(static_cast<unsigned char>('ώ')) << endl;
```

θα δώσουν:

```
68   195    254
```

Αντιστρόφως, τυποθεώρηση τιμής τύπου `int` από τον `unsigned char` μας δίνει το χαρακτήρα που υπάρχει στη θέση που καθορίζει το όρισμα. Π.χ. οι:

```
cout << static_cast<unsigned char>(68) << "   ";
cout << static_cast<unsigned char>(195) << "   ";
cout << static_cast<unsigned char>(254) << endl;
```

θα δώσουν:

```
D   Γ   ώ
```

Μπορούμε λοιπόν να πούμε ότι: για οποιαδήποτε τιμή *c*, τύπου `unsigned char` έχουμε:

`static_cast<unsigned char>(static_cast<int>(c)) == c`

και αντιστρόφως (`int k`, $0 \le$ `k` ≤ 255):

`static_cast<int>(static_cast<unsigned char>(k)) == k`

Αλλά, πώς αποθηκεύεται στη μνήμη μια τιμή τύπου `unsigned char`; Αποθηκεύεται η «ζωγραφιά»; Όχι βέβαια. Μετά την εκτέλεση της εντολής: `c = 'D'` αυτό που θα αποθηκευτεί στη μνήμη, στη θέση *c*, είναι ο *ακέραιος* που δίνεται από την `static_cast<int>('D')`, σε δυαδική παράσταση.

Το να εμφανιστεί η ζωγραφιά στην οθόνη σου ή στον εκτυπωτή σου είναι ένα άλλο πρόβλημα που έχει να κάνει με το μέσο και την τεχνολογία του. Παλιότερα, στον τύπο `unsigned char` ήταν δυνατή η παράσταση όλων των χαρακτήρων που μπορούσε να εμφανίσει ένας ΗΥ. Σήμερα τα πράγματα είναι πιο πολύπλοκα. Οι ΗΥ έχουν δυνατότητα παράστασης πάρα πολλών χαρακτήρων –ο κατάλληλος τύπος είναι πια ο `wchar_t` που θα δούμε στη συνέχεια– ενώ ο `unsigned char` παριστάνει ένα επιλεγμένο υποσύνολο που μπορεί να αλλάζει.

Οι τύποι `signed char` και `char` έχουν και αυτοί τη δυνατότητα να παραστήσουν όλους τους χαρακτήρες που μπορεί να παραστήσει ο `unsigned char`, αλλά έχουμε μια διαφορά με το αποτέλεσμα της `int` για τέτοιες τιμές: αν το 8ο (ή το 1ο αν προτιμάς) δυαδικό ψηφίο είναι 1 ερμηνεύεται ως πρόσημο (–). Π.χ. οι:

```
unsigned char uc;
char c;

c = 'ώ';  uc = 'ώ';
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
c = c - 1;  uc = c;
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
```

θα δώσουν:

```
ώ   ώ
-2   254
ύ   ύ
-3   253
```

Ας δούμε τι ενδιαφέροντα υπάρχουν εδώ:

1. Και στις δύο μεταβλητές δώσαμε την ίδια τιμή, `'ώ'`, επιτυχώς, όπως φαίνεται και από την εκτύπωση των τιμών τους.
2. Οι `static_cast<int>(c)` και `static_cast<int>(uc)` προκάλεσαν την εκτύπωση των –2 και 254. Τι συνέβη; Πρόκειται για δύο διαφορετικές ερμηνείες της ίδιας δυαδικής παράστασης: 11111110_2. Στην πρώτη περίπτωση, που περιμένουμε προσημασμένο

αριθμό, το 1ο "1" θεωρείται ότι δείχνει αρνητική τιμή. Αυτά θα τα μάθουμε αργότερα. Πάντως προς το παρόν μπορείς να θυμάσαι τον εξής κανόνα:

Έστω ότι οι *c* (`int`) και *uc* (`unsigned int`) έχουν ίδια τιμή (στην περίπτωση αυτή: τον ίδιο χαρακτήρα)

αν `static_cast<int>(c)` ≥ 0 τότε
`static_cast<int>(c) == static_cast<int>(uc)`
αλλιώς (αν `static_cast<int>(c)` < 0)
`static_cast<int>(c) == (static_cast<int>(uc) - 256)`

3. Μειώσαμε την τιμή της *c* με τη `c = c - 1`. Έχουμε αυτό το δικαίωμα, αφού οι `char` θεωρούνται ακέραιοι τύποι.
4. Εκχωρήσαμε την τιμή της *c* στη *uc*· μην ξεχνάς ότι το νόημα της εκχώρησης είναι `uc = static_cast<unsigned char>(c)`. Μετά την εκχώρηση οι δύο μεταβλητές έχουν το ίδιο περιεχόμενο (`'ϋ'`).

Στα Windows, στους τύπους `char` και `signed char` όλοι οι χαρακτήρες παριστάνονται με ακέραιες τιμές από −128 μέχρι 127.

4.6 Ο Τύπος char στο Πρόγραμμα

Ας δούμε τώρα το εξής πρόβλημα: Σε κάποιο πρόγραμμά σου έχεις δηλώσει:

```
int k;
```

και στη συνέχεια, την εντολή:

```
cin >> k;
```

Όταν η εντολή αυτή εκτελείται, ας πούμε ότι θέλεις να δώσεις την τιμή 375. Πιέζεις λοιπόν τα πλήκτρα <3>, το <7> και το <5>. Όπως ήδη ξέρεις, το *k* θα πάρει την τιμή 375, αλλά ας δούμε πώς.

Αυτό που «φεύγει» από το πληκτρολόγιο προς τον υπολογιστή είναι η πληροφορία ότι πιέσθηκε το πλήκτρο <3> και στη συνέχεια το <7> και τέλος το <5>. Μέχρι εδώ λοιπόν το πρόγραμμά σου έχει «παραλάβει» τρεις *χαρακτήρες*, τρεις τιμές τύπου `char`. Πώς θα βγει τώρα το 375; Αυτό θα γίνει, με την εκτέλεση μερικών εντολών που παίρνουν ως στοιχεία εισόδου τους τρεις χαρακτήρες και βγάζουν ως αποτέλεσμα την τιμή 375 στη θέση μνήμης *k* του προγράμματός μας. Στη συνέχεια με ένα απλοϊκό προγραμματάκι θα προσπαθήσουμε να σου δείξουμε πώς γίνεται αυτή η δουλειά.

Κατ' αρχάς να επαναλάβουμε τον περιορισμό που είπαμε για τα σύνολα χαρακτήρων: «Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη». Π.χ. στον ASCII οι θέσεις αυτές είναι από το 48 (`'0'`) μέχρι το 57 (`'9'`). Έτσι, αν από τη σειρά ενός ψηφίου στον πίνακα αφαιρέσουμε την σειρά που έχει ο χαρακτήρας '0' παίρνουμε την αριθμητική τιμή του ψηφίου αυτού. Π.χ. (στο ASCII):

`static_cast<int>('7')-static_cast<int>('0')` = 55 − 48 = 7

Με βάση τις παραπάνω παρατηρήσεις, μπορούμε να δούμε πώς περίπου διαβάζει τα αριθμητικά στοιχεία η C++. Ας πούμε ότι θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει τριψήφιους ακέραιους χωρίς πρόσημο· ή καλύτερα, θα διαβάζει τρεις χαρακτήρες - ψηφία και θα τους μεταφράζει σε ακέραιο αριθμό. Από τα τρία ψηφία: το πρώτο θα είναι το ψηφίο των εκατοντάδων, το δεύτερο των δεκάδων και το τρίτο των μονάδων. Αφού υπολογίσουμε την τιμή που αντιστοιχεί σε κάθε ψηφίο, τις προσθέτουμε πολλαπλασιασμένες με την αντίστοιχη δύναμη του 10. Να το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int ord0 = static_cast<int>('0');
// η θέση του '0' στο σύνολο χαρακτήρων
```

```
   unsigned char  digit1, digit2, digit3;
// τρία ψηφία (χαρακτήρες) που διαβάζονται από το πληκτρολόγιο
   unsigned char  ceol;  // εδώ θα πάει το <enter>
   int value1, value2, value3;
// οι ακέραιοι που αντιστοιχούν στα τρία ψηφία
   int integer;
// Ο ακέραιος που φιλοδοξούμε να διαβάσουμε

   cin.get(digit1);  cin.get(digit2);  cin.get(digit3);
   cin.get(ceol);
   value1 = static_cast<int>(digit1) - ord0;
   value2 = static_cast<int>(digit2) - ord0;
   value3 = static_cast<int>(digit3) - ord0;
   integer = value1*100 + value2*10 + value3;
   cout << " Ακέραιος = " << integer << endl;
}
```

Αν δώσουμε σ' αυτό το πρόγραμμα:

375

θα πάρουμε απάντηση:

```
 Ακέραιος = 375
```

Αν δώσουμε:

007

θα πάρουμε:

```
 Ακέραιος = 7
```

Το πρόγραμμα περιμένει **τρία δεκαδικά ψηφία.** Με διαφορετικά στοιχεία εισόδου δεν θα πάρεις την απάντηση που περιμένεις. Αν δώσουμε:

␣␣*7*

θα πάρουμε λάθος απάντηση:

```
 Ακέραιος = -1753
```

Και η εκτύπωση πώς γίνεται; Η οθόνη σου ή ο εκτυπωτής σου περιμένουν χαρακτήρες. Αν θέλουμε να γράψουμε τον αριθμό 375 θα πρέπει πρώτα να γράψουμε το ψηφίο των εκατοντάδων (`'3'`), μετά το ψηφίο των δεκάδων (`'7'`) και τέλος το ψηφίο των μονάδων (`'5'`).

Οι εκατοντάδες υπολογίζονται ως πηλίκο της *ακέραιης* διαίρεσης 375 `/` 100. Αν πάρουμε το υπόλοιπο αυτής της διαίρεσης (375 `%` 100 `==` 75) και το διαιρέσουμε δια 10, το πηλίκο (75 `/` 10) μας δίνει τις δεκάδες, ενώ το υπόλοιπο (75 `%` 10) μας δίνει τις μονάδες.

Οι παρακάτω εντολές κάνουν αυτή τη δουλειά για τριψήφιους μη-αρνητικούς ακέραιους.

```
   cin >> integer;
   value1 = integer / 100;
   digit1 = static_cast<unsigned char>(ord0 + value1);
   integer = integer % 100;  value2 = integer / 10;
   digit2 = static_cast<unsigned char>(ord0 + value2);
   integer = integer % 10;  value3 = integer;
   digit3 = static_cast<unsigned char>(ord0 + value3);
   cout << digit1 << digit2 << digit3 << endl;
```

Η C++ σου δίνει συναρτήσεις για επεξεργασία τιμών τύπου `char`. Στον Πίν. 4-3 βλέπεις μερικές από αυτές, που σου επιτρέπουν να δεις τι είδους είναι κάποιος χαρακτήρας.

Οι συναρτήσεις επιστρέφουν τιμή τύπου `int` για συμβατότητα με τη C, από την οποίαν και κληρονομήθηκαν. Θα τις χειριζόμαστε ως συναρτήσεις τύπου `bool` (κατηγορήματα). Όταν τις χρησιμοποιείς θα πρέπει να βάζεις στο πρόγραμμά σου την οδηγία: "`#include <cctype>`". Ας δούμε ένα παράδειγμα. Το πρόγραμμα:

```
#include <iostream>
#include <cctype>
```

Όνομα	Δίνει αποτέλεσμα `true` (≠ 0) αν το όρισμα είναι:
`isalpha`	Λατινικό γράμμα (`'a'..'z'`, `'A'..'Z'`)
`isupper`	Κεφαλαίο λατινικό γράμμα (`'A'..'Z'`)
`islower`	Πεζό λατινικό γράμμα (`'a'..'z'`)
`isdigit`	Δεκαδικό ψηφίο (`'0'..'9'`)
`isxdigit`	Δεκαεξαδικό ψηφίο (`'0'..'9'`, `'a'..'f'`, `'A'..'F'`)
`isspace`	Κάποιος από τους `' '`, `'\t'`, `'\r'`, `'\n'`, `'\f'`
`iscntrl`	Χαρακτήρας ελέγχου: `static_cast<char>(0) .. static_cast<char>(31)`, `static_cast<char>(127)`
`ispunct`	Τίποτε από τα παραπάνω
`isalnum`	Λατινικό γράμμα ή δεκαδικό ψηφίο
`isprint`	Εκτυπώσιμος: `static_cast<char>(32)` (= `' '`) .. `static_cast<char>(126)` (= `'~'`)
`isgraph`	`isalpha || isdigit || ispunct`
`isascii`	Χαρακτήρας ASCII (ISO 646) `static_cast<char>(0) .. static_cast<char>(127)`

Πίν. 4-3 Συναρτήσεις της C++ για επεξεργασία χαρακτήρων. Παίρνουν ένα όρισμα τύπου `char`. Επιστρέφουν τιμή `true` αν το όρισμα έχει την ιδιότητα που περιγράφεται στη δεύτερη στήλη. Αλλιώς `false`. Για να τις χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου: `#include <ctype>`.

```
using namespace std;
int main()
{
   char c1 = 'f', c2 = 'ϕ', c3 = '3';

   cout << isalpha(c1) << "  " << isalpha(c2) << endl;
   cout << (isxdigit(c1) && !isdigit(c1)) << endl;
   cout << (isxdigit(c3) && !isdigit(c3)) << endl;
}
```

μας δίνει:

```
8  0
1
0
```

Το `'f'` είναι γράμμα του λατινικού αλφαβήτου και η `isalpha(c1)` μας δίνει τιμή `8` ≠ 0 δηλαδή `true`. Το `'ϕ'` δεν είναι γράμμα του λατινικού αλφαβήτου· η `isalpha(c2)` μας δίνει τιμή `0` δηλαδή `false`. Στη συνέχεια ζητάμε την τιμή της πρότασης «το `'f'` είναι δεκαεξαδικό ψηφίο και το `'f'` δεν είναι δεκαδικό ψηφίο». Η απάντηση είναι `true` (1). Αλλά, στη συνέχεια μας λέει ότι η τιμή της πρότασης «το `'3'` είναι δεκαεξαδικό ψηφίο και το `'3'` δεν είναι δεκαδικό ψηφίο» είναι `false` (0).

Α, και κάτι ακόμη: αν σε ενοχλεί το 8 άλλαξε την πρώτη εντολή σε:

```
   cout << static_cast<bool>(isalpha(c1)) << "  "
        << static_cast<bool>(isalpha(c2)) << endl;
```

Πάντως, οι συνθήκες με τις λογικές πράξεις υπολογίζονται σωστά· δεν υπάρχει πρόβλημα.

Στο `cctype` δηλώνεται ακόμη η συνάρτηση *tolower* που αν τροφοδοτηθεί με κεφαλαίο γράμμα του λατινικού αλφαβήτου μας δίνει το αντίστοιχο πεζό, π.χ. η `tolower('Q')` θα δώσει `'q'`. Η *toupper*, που δηλώνεται επίσης στο `cctype`, αν τροφοδοτηθεί με πεζό θα μας δώσει το αντίστοιχο κεφαλαίο, π.χ. η `toupper('q')` θα δώσει `'Q'`.

4.7 Ο Τύπος wchar_t

Ο τύπος **wchar_t** είναι ένα εργαλείο της C++[5] που χρησιμοποιείται για να αντιμετωπισθεί το πρόβλημα της παράστασης στον υπολογιστή πολλών χαρακτήρων. Έχει 65536 διαφορετικές τιμές, αντί για τις 256 του **char** και χρησιμοποιείται για την υλοποίηση του προτύπου Unicode.

Μια σταθερά τύπου **wchar_t** είναι μια σταθερά τύπου **char** με το πρόθεμα **L**, π.χ.:

```
L'a'   L'%'  L'ξ'
```

Όπως λέγαμε και στον Πίν. 2-1, υλοποιείται σε 16 δυαδικά ψηφία (2 ψηφιολέξεις). Μπορείς να δοκιμάσεις την:

```
cout << sizeof('A') << "  " << sizeof(L'A') << endl;
```

που θα σου δώσει:

```
1  2
```

Δεν μπορείς να διαβάσεις τιμές τύπου **wchar_t** από το *cin* ενώ μπορείς να στείλεις τιμές στο ρεύμα *cout*, αλλά θα δεις να τυπώνονται οι αντίστοιχοι αριθμοί. Π.χ. αν

```
wchar_t a, b;
```

τότε οι

```
a = L'a';  b = L'\n';
cout << a << "  " << b << endl;
```

θα δώσουν:

```
97  10
```

Αν περιλάβεις στο πρόγραμμά σου το **cwctype**[6] μπορείς να χρησιμοποιήσεις τις συναρτήσεις επεξεργασίας τιμών τύπου **wchar_t**, αντίστοιχες αυτών που έχουμε στον Πίν. 4-3. Τα ονόματά τους διαφέρουν κατά το ότι αντί για **is** έχουν πρόθεμα **isw**. Π.χ., για τις παραπάνω τιμές των *a*, *b*, η

```
cout << iswalnum(a) << "  " << iswprint(b)  << endl;
```

θα δώσει:

```
256  0
```

(ο **L'a'** είναι αλαφαριθμητικός ενώ ο **L'\n'** δεν είναι εκτυπώσιμος.)

Παρατήρηση: ▶

Σύμφωνα με αυτά που είπαμε, στην *a* μπορούμε να αποθηκεύσουμε το ελληνικό γράμμα "πεζό άλφα" στην κωδικοποίηση Unicode, που είναι η τιμή **0x03b1** (945) γράφοντας "**a = 0x03b1**" ή "**a = 945**". Όχι όμως γράφοντας "**a = L'α'**"· στην περίπτωση αυτή αποθηκεύεται η τιμή 225 (Windows).◀

4.8 Τακτικοί Τύποι

Ξεκινούμε με την εξής παρατήρηση:

- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **bool** στο υποσύνολο { 0, 1 } του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **unsigned char** στο υποσύνολο 0 .. 255 του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές των τύπων **signed char** και **char** στο υποσύνολο −128 .. 127 του **int**.

---

[5] Στη C ορίζεται στο stddef.h ως (Borland C):
```
typedef unsigned short wchar_t;
```

[6] Αν δεν το έχεις δοκίμασε το **cctype**.

Η απεικόνιση –και στις τρεις περιπτώσεις– γίνεται με την `int`. Οι τύποι αυτοί λέγονται **τακτικοί τύποι** (ordinal types)[7]. Φυσικά και ο `int`, που μπορεί να απεικονισθεί στον εαυτό του, είναι επίσης τακτικός τύπος. Και στην περίπτωση αυτή η απεικόνιση γίνεται με την `int`, αλλά, φυσικά για τον `int`, η `int` απεικονίζει κάθε αριθμό στον εαυτό του.

Οι τακτικοί τύποι είναι **διαταγμένοι** (ordered). Για κάθε τιμή ενός τακτικού τύπου υπάρχει μια **προηγούμενη** (predecessor) και μια **επόμενη** (successor). Π.χ. προηγούμενος του 0 είναι ο –1 και επόμενος ο 1, προηγούμενος του `'C'` είναι ο `'B'` και επόμενος ο `'D'`. Φυσικά, δεν υπάρχει επόμενος του *INT_MAX* και προηγούμενος του *INT_MIN*. Στον τύπο `char` δεν υπάρχει προηγούμενος του `char(-128)` και επόμενος του `char(127)`. Στον `bool` δεν υπάρχει προηγούμενη της `false` ούτε επόμενη της `true`.

4.9 Απαριθμητοί Τύποι (που Ορίζονται από τον Χρήστη)

Η C++ δίνει τη δυνατότητα στον προγραμματιστή να ορίζει δικούς του τύπους. Μια τέτοια κατηγορία τύπων είναι και τακτικοί ή **απαριθμητοί τύποι** (enumerated types). Μπορείς, για παράδειγμα, στο πρόγραμμά σου να ορίσεις:

```
   enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
                  friday, saturday };
   enum EurUn { Austria, Belgium, Bulgaria, Cyprus, CzechRepublic,
                Denmark, Ellas, Estonia, Finland, France, Germany,
                Hungary, Ireland, Italy, Latvia, Lithuania,
                Luxembourg, Malta, Netherlands, Poland, Portugal,
                Romania, Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom };
```

στη συνέχεια να δηλώσεις:

```
   WeekDay day1, day2;
   EurUn country;
```

και να δώσεις τιμές:

```
   day1 = sunday;  day2 = wednesday;
// . . .
   country = Ellas;
```

Ο ορισμός ενός απαριθμητού τύπου αρχίζει με το λεξικό σύμβολο "`enum`". Στη συνέχεια γράφουμε το όνομα του τύπου και μετά, μέσα σε άγκιστρα, τα *ονόματα των τιμών* που περιλαμβάνονται στον τύπο.

Φυσικά, ο ορισμός τύπου μπαίνει, στο πρόγραμμά μας, *πριν* από τις δηλώσεις μεταβλητών.

Ο ορισμός του τύπου είναι συγχρόνως και ορισμός των τιμών που περιέχει. Για παράδειγμα, οι τιμές που μπορεί να πάρει κάθε μεταβλητή τύπου *WeekDay*, όπως η *day*1, είναι οι: *sunday*, *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday*.

Οι τιμές του κάθε τύπου είναι διαταγμένες σύμφωνα με τον ορισμό του τύπου. Π.χ.:

sunday < *monday* < *tuesday* < *wednesday* < *thursday* < *friday* < *saturday*

Austria < *Belgium* < *Bulgaria* < . . . < *Sweden* < *UnitedKingdom*

Η `static_cast<int>` δέχεται ως όρισμα τιμή τέτοιων τακτικών τύπων και μας δίνει ως τιμή τη θέση (τάξη) του ορίσματος στον τύπο:

`static_cast<int>(sunday) == 0`,
`static_cast<int>(monday) == 1`,
`static_cast<int>(tuesday) == 2` κ.ο.κ.
`static_cast<int>(Austria) == 0`,
`static_cast<int>(Belgium) == 1`,
`static_cast<int>(Denmark) == 2` κ.ο.κ.

[7] Ο όρος από την Pascal.

Μαζί με κάθε τέτοιο τύπο ορίζεται και η αντίστοιχη αντίστροφη συνάρτηση που μας δίνει τιμές του τύπου αυτού:

`static_cast<WeekDay>(0) == sunday`,
`static_cast<WeekDay>(1) == monday`,
`static_cast<WeekDay>(2) == tuesday` κ.ο.κ.
`static_cast<EurUn>(0) == Austria`,
`static_cast<EurUn>(1) == Belgium`,
`static_cast<EurUn>(2) == Bulgaria` κ.ο.κ.

Μπορείς να χρησιμοποιείς τέτοιες τιμές σε εντολές εξόδου· η "`cout << v …`" θα λειτουργήσει σαν την: "`cout << static_cast<int>(v)...`"

Δεν μπορείς να χρησιμοποιείς μεταβλητές τέτοιων τύπων σε εντολές εισόδου (`cin >> v`). Το πρόβλημα παρακάμπτεται με τη χρήση μιας μεταβλητής "`int k`" και με τις εξής εντολές:

```
cin >> k;
v = static_cast<T>( k );
```

όπου *T* ο τύπος της *v*.

Μπορείς αν θέλεις να αλλάζεις τις τιμές του `int` στους οποίους απεικονίζονται οι τιμές του τύπου σου. Π.χ. με τον ορισμό:

```
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };
```

θα έχεις:

```
static_cast<int>( zero) == 48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αν θέλεις μπορείς να δίνεις περισσότερα από ένα ονόματα στην ίδια τιμή. Με τους παρακάτω ορισμούς:

```
enum Digit { miden = 48, zero = 48, one, two, three, four,
             five, six, seven, eight, nine };
enum Digit { zero = 48, one, two, three, four,
             five, six, seven, eight, nine, miden = 48 };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) ==48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αλλά προσοχή· αν δώσεις:

```
enum Digit { zero = 48, one, two, three, four,
             miden = 48, five, six, seven, eight, nine };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) == 48,
static_cast<int>(one) == static_cast<int>(five) == 49,
static_cast<int>(two) == static_cast<int>(six) == 50,
. . .
```

4.10 Μετονομασία Τύπου

Θα μπορούσαμε να ορίσουμε τους τύπους *WeekDay* και *EurUn* και ως εξής:

```
typedef enum { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday }                        WeekDay;
typedef enum { Austria, Belgium, Bulgaria, Cyprus,
               CzechRepublic, Denmark, Ellas, Estonia, Finland,
               France, Germany, Hungary, Ireland, Italy,
               Latvia, Lithuania, Luxembourg, Malta,
```

```
                Netherlands, Poland, Portugal, Romania,
                Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom }                          EurUn;
```

Το **typedef** (*type def*inition) είναι λεξικό σύμβολο. Αν σκεφτούμε ότι ο τύπος είναι στην παργαματικότητα το **enum { sunday, monday, tuesday, wednesday, thursday, friday, saturday }**, η **typedef** είναι στην πραγματικότητα εντολή *μετονομασίας* τύπου. Με αυτήν μπορείς να μετονομάσεις και άλλους τύπους που είναι ήδη ορισμένοι. Π.χ.

```
typedef unsigned int       Natural;
typedef unsigned long int  LongNatural;
typedef unsigned char      byte;
typedef bool               Logical;
```

Οι «νέοι» τύποι, *Natural*, *LongNatural*, *Logical* και *byte*, είναι στην πραγματικότητα οι αρχικοί τύποι και, επομένως, έχουν όλες τις ιδιότητές τους. Μετά τους παραπάνω ορισμούς μπορείς να δηλώσεις:

```
Natural     n, eN;
LongNatural athr, aAth;
Logical     yparxei;
byte        c1, c2;
```

Ασκήσεις

Α Ομάδα

4-1 Εξήγησε πώς βγήκε το αποτέλεσμα –1753 όταν δώσαμε στο πρόγραμμα της §4.5, στοιχείο εισόδου: ␣␣**7**.

Β Ομάδα

4-2 Γράψε πρόγραμμα που θα αποδεικνύει την ισοδυναμία: ((*A* **||** *B*) **&&** *B*) ≡ *B*.

4-3 Το ίδιο για τις ταυτολογίες: (*A* **&&** *B*) ⇒ *A* και (*A* **&&** *B*) ⇒ *B*.

Γ Ομάδα

4-4 Με βάση το πρόγραμμα της §4.5, γράψε πρόγραμμα που θα διαβάζει *δυαδικούς ακέραιους*.

κεφάλαιο

5

Επιλογές

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κάνεις το πρώτο βήμα στη χρήση συνθηκών για τον έλεγχο εκτέλεσης ενός προγράμματος: Να μπορείς να επιλέγεις ομάδες εντολών που θα εκτελεσθούν ή δεν θα εκτελεσθούν.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις προγράμματά πιο ευέλικτα, που η πορεία της εκτέλεσής τους θα εξαρτάται από τιμές συνθηκών και των μεταβλητών που περιλαμβάνονται σε αυτές. Θα μπορείς να ελέγχεις αν ισχύει η προϋπόθεση όταν αρχίζει η εκτέλεση του προγράμματος.

Έννοιες κλειδιά:

- *εντολές επιλογής*
- *εντολή* `if`
- *εντολή* `ifelse`
- *σύνθετη εντολή*
- *πολλαπλές επιλογές*
- *λογική εντολών* `if` *και* `ifelse`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Οι εντολές που γνωρίσαμε στα προηγούμενα κεφάλαια, δηλ. οι εντολές εισόδου/εξόδου, οι εντολές δήλωσης και η εντολή εκχώρησης μας δίνουν τη δυνατότητα να γράφουμε απλά προγράμματα υπολογισμών, που όμως δεν έχουν δυνατότητες για επιλογή «διαδρομών», στη διάρκεια της εκτέλεσης. Αυτό σημαίνει ότι, όταν εκτελείται το πρόγραμμα, δεν υπάρχει δυνατότητα «παράκαμψης» κάποιων εντολών του προγράμματος και επομένως *όλες οι*

εντολές εκτελούνται υποχρεωτικά με την ίδια σειρά που είναι γραμμένες στο πρόγραμμα, η μια μετά την άλλη.

Φυσικά, αυτός ο περιορισμός δεν είναι βολικός στην πράξη και στις γλώσσες προγραμματισμού υπάρχουν ειδικές εντολές, που μας δίνουν τη δυνατότητα να ζητούμε την εκτέλεση ή την παράκαμψη εντολών που υπάρχουν στο πρόγραμμά μας.

Στη C++ τέτοιες εντολές είναι οι **εντολές επιλογής** (selection statements). Στο κεφάλαιο αυτό, θα ασχοληθούμε ειδικά με τέτοιες εντολές.

5.1 Επιλογή – Οι Εντολές if

Στο προηγούμενο κεφάλαιο λέγαμε ότι «*μπορούμε ... να ρυθμίζουμε την εκτέλεση του προγράμματός μας με βάση τις τιμές συνθηκών.*» Εδώ θα μάθουμε έναν τρόπο για να κάνουμε κάτι τέτοιο. Ξεκινούμε με ένα παράδειγμα.

Θέλουμε ένα πρόγραμμα που θα διαβάζει δύο πραγματικούς αριθμούς, από το πληκτρολόγιο, και θα γράφει τον μεγαλύτερο από αυτούς μαζί με το μήνυμα `" Μεγαλύτερος είναι ο αριθμός: "`. Ας πούμε ότι δηλώνουμε:

```
   double x, y, maxxy;
```

και διαβάζουμε:

```
   cin >> x >> y;
```

Στο τέλος θα γράψουμε:

```
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
```

Το πρόβλημά μας είναι να δώσουμε στη *maxxy* τη σωστή τιμή. Ένας απλός τρόπος, που θα σκέφτονταν πολλοί, είναι ο εξής:

αν $x > y$ **τότε** { **βάλε** *maxxy* ← *x*
αλλιώς { **βάλε** *maxxy* ← *y*

Τα «**βάλε** *maxxy* ← *x*» και «**βάλε** *maxxy* ← *y*» ξέρουμε να τα γράψουμε σε C++. Τα υπόλοιπα; Σχεδόν μεταφράζουμε στα αγγλικά:

```
   if ( x > y )  maxxy = x;
          else  maxxy = y;
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   if (x > y)  maxxy = x;
         else  maxxy = y;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

Η εντολή **ifelse** της C++ έχει γενικώς τη μορφή:

```
if ( S )  Et  else  Ef
```

όπου *S* μια συνθήκη (λογική παράσταση) και E_t, E_f εντολές. Η εντολή εκτελείται ως εξής:

- υπολογίζεται η τιμή της λογικής παράστασης *S*,
- αν η τιμή είναι `true` (αν η συνθήκη ισχύει) τότε α) εκτελείται η εντολή E_t και στη συνέχεια β) αγνοείται η εντολή E_f και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**,
- αν η τιμή είναι `false` (αν η συνθήκη δεν ισχύει) τότε α) αγνοείται η εντολή E_t, β) εκτελείται η εντολή E_f και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**.

Αυτά φαίνονται και στο λογικό διάγραμμα της **ifelse** στο Σχ. 5-1. Ο ρόμβος δείχνει το σχήμα «απόφασης» –επιλογής διαδρομής– που λαμβάνεται ανάλογα με την τιμή (`true` ή `false`) της παράστασης *S*. Από το σχήμα μπορείς να καταλάβεις ακόμη γιατί οι εντολές **ifelse** (και οι **if** που θα δούμε στη συνέχεια) λέγονται και εντολές **διακλάδωσης** (branching).

Ερχόμαστε στο παράδειγμά μας και ας πούμε ότι όταν εκτελείται η "`cin >> x >> y`" απαντούμε με:

```
1.6   0.6
```

Το "`1.6`" γίνεται τιμή της *x* και το "`0.6`" γίνεται τιμή της *y*. Στη συνέχεια εκτελείται η **ifelse** και κατ' αρχάς υπολογίζεται η συνθήκη "`x > y`" ή "1.6 > 0.6", που είναι `true`. Ύστερα από αυτό εκτελείται η εντολή "`maxxy = x`" και στη συνέχεια η εντολή που ακολουθεί την **ifelse**, δηλαδή η "`cout << ...`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 1.6
```

Η "`maxxy = y`" είναι σαν να μην υπάρχει.

Αν στη "`cin >> ...`" απαντήσουμε με:

```
1.6   2.6
```

η "`x > y`" (1.6 > 2.6) θα πάρει τιμή `false`. Στην περίπτωση αυτή θα εκτελεσθεί η εντολή "`maxxy = y`" και στη συνέχεια η "`cout << ...`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 2.6
```

Το ίδιο θα συμβεί και στην περίπτωση που απαντούμε στη "`cin >> ...`" με:

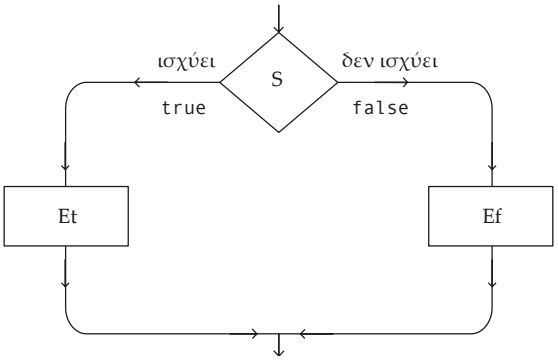
```
2.6   2.6
```

αφού και πάλι η "`x > y`" (2.6 > 2.6) θα πάρει τιμή `false`.

Ας ξαναδούμε τώρα το παράδειγμα 3 της §2.3. Κάναμε ένα πρόγραμμα που έβρισκε τις ρίζες της εξίσωσης $ax^2 + bx + c = 0$. Αλλά όταν η διακρίνουσα ήταν αρνητική είχαμε πρόβλημα. Τώρα μπορούμε να κάνουμε ένα καλύτερο πρόγραμμα: οι εντολές

```
x1 = (-b + sqrt(delta))/(2*a);
x2 = (-b - sqrt(delta))/(2*a);
cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
```

θα πρέπει να εκτελούνται μόνον αν *delta* ≥ 0 αλλιώς θα πρέπει να βγαίνει ένα μήνυμα που



Σχ. 5-1 Λογικό διάγραμμα για την εντολή **ifelse**.

θα λέει ότι η εξίσωση δεν έχει πραγματικές λύσεις. Θα βάλουμε λοιπόν μια **ifelse**, αλλά… Μάθαμε παραπάνω ότι μετά τη συνθήκη μπορούμε να γράψουμε μια εντολή και εδώ έχουμε τρεις· τι κάνουμε τώρα;

Η C++ μας προσφέρει την εξής δυνατότητα: Να δημιουργήσουμε μια **σύνθετη εντολή** (compound statement) που θα περιλαμβάνει τις τρεις εντολές που μας ενδιαφέρουν. Αυτό γίνεται ως εξής: «συσκευάζουμε» τις εντολές μας με ένα άγκιστρο (**{**) στην αρχή και ένα άγκιστρο (**}**) στο τέλος. Στην περίπτωσή μας:

```
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
```

Να πώς γίνεται το πρόγραμμά μας:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;    // οι συντελεστές της εξίσωσης
   double  x1, x2;     // οι ρίζες της εξίσωσης
   double  delta;      // η διακρίνουσα

   cout << "Δώσε τρεις πραγματικούς αριθμούς" << endl;
   cin >> a >> b >> c;
   cout << " a = " << a << "   b = " << b
        << "   c = " << c << endl;
   delta = b*b - 4*a*c;
   if ( delta >= 0 )
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
   else
      cout << " Δεν υπάρχουν Πραγματικές Λύσεις" << endl;
   cout << " ΤΕΛΟΣ" << endl;
}
```

Αν τροφοδοτήσουμε το πρόγραμμα με τους παρακάτω τρείς αριθμούς (συντελεστές της εξίσωσης):

```
            1   3   -10
```

θα πάρουμε το εξής αποτέλεσμα:

```
 a = 1   b = 3   c = -10
 Λύση1 = 2   Λύση2 = -5
 ΤΕΛΟΣ
```

Αν όμως πληκτρολογήσουμε τους αριθμούς:

```
            2   3   4
```

θα πάρουμε τα εξής:

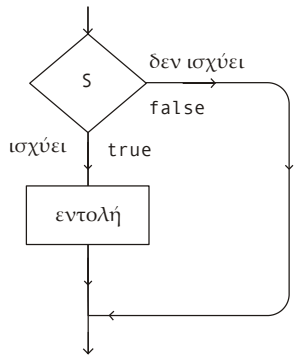
```
 a = 2   b = 3   c = 4
 Δεν υπάρχουν Πραγματικές Λύσεις
 ΤΕΛΟΣ
```

Ζητάμε άλλη μια εκτέλεση, δίνοντας για στοιχεία εισόδου:

```
    0   1   2
```

Αποτέλεσμα: διακοπή εκτέλεσης του προγράμματος και κάποιο «κακό» μήνυμα. Γιατί; Το "**0**" έγινε τιμή της *a* και στον υπολογισμό των *x1* και *x2* διαιρούμε δια 2*a*. Χρειαζόμαστε λοιπόν άλλον έναν έλεγχο, άλλη μια **if**. Θα επανέλθουμε στη συνέχεια.

5.2 Η Εντολή if



Σχ. 5-2 Λογικό διάγραμμα για την εντολή **if**.

Ξαναγυρνάμε στο πρόβλημα του μεγίστου· πολλοί θα προτιμούσαν να δώσουν μια κάπως διαφορετική λύση:

αν *x* > *y* **τότε** { **βάλε** *maxxy* ← *x*

αν *x* ≤ *y* **τότε** { **βάλε** *maxxy* ← *y*

Αυτό πώς το γράφουμε σε C++; Μια πρώτη σκέψη είναι η εξής:

```
if (x > y)  maxxy = x; else;
if (x <= y)  maxxy = y; else;
```

που στην πραγματικότητα μεταφράζει το:

αν *x* > *y* **τότε** { **βάλε** *maxxy* ← *x*

αλλιώς { μη κάνεις τίποτε

αν *x* ≤ *y* **τότε** { **βάλε** *maxxy* ← *y*

αλλιώς { μη κάνεις τίποτε

Μετά το `else` θεωρείται ότι υπάρχει η **κενή** (empty) ή **μηδενική** (null) εντολή που αντιπροσωπεύει στη C++ αυτό το «*μη κάνεις τίποτε*» του ψευδοκώδικα.

Πάντως μας δίνεται και άλλη δυνατότητα, χωρίς εκείνο το "`else;`":

```
if (x > y)  maxxy = x;
if (x <= y)  maxxy = y;
```

Εδώ χρησιμοποιούμε την απλή **if** που έχει γενικώς τη μορφή:

`if ( S )  Et`

όπου *S* μια συνθήκη (λογική παράσταση) και E_t εντολή. Η εντολή εκτελείται ως εξής:

- υπολογίζεται η τιμή της λογικής παράστασης *S*,
- αν η τιμή είναι `true` (αν η συνθήκη ισχύει) τότε α) εκτελείται η εντολή E_t και στη συνέχεια β) η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **if**,
- αν η τιμή είναι `false` (αν η συνθήκη δεν ισχύει) τότε αγνοείται η εντολή E_t και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**.

Στο Σχ. 5-2 βλέπεις και το λογικό διάγραμμα της **if**.

Να πώς γράφεται το πρόγραμμά μας:

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   if (x > y)  maxxy = x;
   if (x <= y)  maxxy = y;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

και να πώς εκτελείται. Στη `cin >> x >> y` απαντούμε με:

```
1.6   0.6
```

Το 1.6 γίνεται τιμή της *x* και το 0.6 γίνεται τιμή της *y*. Στη συνέχεια εκτελείται η "`if (x > y) maxxy = x`" και πρώτα απ' όλα υπολογίζεται η συνθήκη "`x > y`" ή "1.6 > 0.6", που είναι `true`. Ύστερα από αυτό εκτελείται η εντολή "`maxxy = x`" και στη συνέχεια η εντολή που ακολουθεί την **if**, δηλαδή η "`if (x <= y) maxxy = y`". Και εδώ υπολογίζεται η συνθήκη "`x <= y`" ή 1.6 <= 0.6 που είναι `false`· έτσι η "`maxxy = y`" *δεν* εκτελείται και πηγαίνουμε στην "`cout << ...`" που δίνει:

```
 Μεγαλύτερος είναι ο αριθμός: 1.6
```

Η "`maxxy = y`" είναι σαν να μην υπάρχει.

Αν στη "`cin >> …`" απαντήσουμε με:

Πλαίσιο 5.1

Η Εντολή if

"`if`", "`(`", *συνθήκη*, "`)`", *εντολή*

Συστατικά:

- το `if` είναι λεξικό σύμβολο,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή* είναι μια οποιαδήποτε εντολή της C++.

Εκτέλεση της εντολής **if**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη

Αν η τιμή είναι `false` η εντολή που ακολουθεί τη συνθήκη αγνοείται.

Η Εντολή ifelse

"`if`", "`(`", *συνθήκη*, "`)`", *εντολή*-1, "`else`", *εντολή*-2

Συστατικά:

- τα `if` και `else` είναι λεξικά σύμβολα,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή*-1 (περιοχή του if) είναι μια οποιαδήποτε εντολή της C++.
- η *εντολή*-2 (περιοχή του else) είναι μια οποιαδήποτε εντολή της C++.

Εκτέλεση της εντολής **ifelse**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η *εντολή*-1 που βρίσκεται μεταξύ της συνθήκης και του **else**.

Η *εντολή*-2 αγνοείται.

αλλιώς (αν η τιμή είναι `false`)

εκτελείται η *εντολή*-2 που βρίσκεται μετά το **else**.

Η *εντολή*-1 αγνοείται.

1.6 2.6

η "`x > y`" (1.6 > 2.6) θα πάρει τιμή `false` και η "`maxxy = x`" δεν θα εκτελεσθεί. Στην επόμενη **if** η συνθήκη "`x <= y`" ή 1.6 <= 2.6 είναι `true`· στην περίπτωση αυτή θα εκτελεσθεί η εντολή "`maxxy = y`" και στη συνέχεια η "`cout << …`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 2.6
```

Πριν προχωρήσουμε ας δούμε έναν ακόμη τρόπο για να λύσουμε το πρόβλημα του μεγίστου· ο τρόπος αυτός θα μας είναι πολύ χρήσιμος στη συνέχεια.

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   maxxy = y;
   if (x > maxxy)  maxxy = x;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

Εδώ *θεωρούμε αυθαιρέτως* ότι ο μεγαλύτερος αριθμός είναι ο δεύτερος (δηλ. ο *y*). Στην επόμενη εντολή όμως ελέγχουμε μήπως έχουμε κάνει λάθος (`x > maxxy`)· στην περίπτωση αυτή κάνουμε τη διόρθωση: `maxxy = x`.

Από τους τρεις τρόπους, ο πρώτος είναι ο πιο οικονομικός. Ας δούμε γιατί.

Στο πρώτο πρόγραμμα θα γίνει μια σύγκριση –η `x > y`– και αναλόγως θα εκτελεσθεί μια μόνο από τις εντολές: "`maxxy = x`" ή "`maxxy = y`". Δηλαδή θα έχουμε 1 σύγκριση και 1 εκχώρηση.

Στο δεύτερο θα εκτελεστούν και οι δυο συγκρίσεις (`x > y`, `x <= y`), μια και θα εκτελεσθούν και οι δυο εντολές **if**. Επειδή όμως οι δυο συνθήκες είναι αντίθετες, η μια θα είναι `true` και η άλλη `false`. Έτσι, από τις εντολές "`maxxy = x`" και "`maxxy = y`" θα εκτελεστεί μόνο η μια. Έχουμε λοιπόν: 2 συγκρίσεις και 1 εκχώρηση.

Στην τρίτη περίπτωση η εκχώρηση "`maxxy = y`" θα γίνει οπωσδήποτε. Το ίδιο και η σύγκριση "`x > maxxy`". Η εκχώρηση "`maxxy = x`" θα εκτελεσθεί μόνο στην περίπτωση που $x > y$. Αν δεχτούμε ότι η πιθανότητα για κάτι τέτοιο είναι 50%, το πρόγραμμα αυτό θα εκτελεί 1 σύγκριση και 1.5 εκχώρηση κατά μέσο όρο.

5.3 Φωλιασμένες if – Πολλαπλές Επιλογές

Στις θέσεις των εσωτερικών εντολών της **ifelse** (ή της **if**) μπορεί να δοθεί, όπως ειπώθηκε, οποιαδήποτε εντολή της C++. Επομένως μπορεί να δοθεί μια άλλη εντολή **if**, όπως δείχνει το παρακάτω παράδειγμα:

```
if ( S1 )
   if ( S2 ) εντολή-1
        else εντολή-2
```

Εδώ μπορεί να αναρωτηθείς: που κολλάει το `else`; Στη C++ ισχύει η πρόσθετη σύμβαση που λέει ότι το παραπάνω γενικό σχήμα της **if** (μέσα στην **if**) είναι ισοδύναμο με το σχήμα:

```
if ( S1 )
{
   if ( S2 )  εντολή-1
        else  εντολή-2
}
```

Δηλαδή, η δεύτερη **if** είναι η περιοχή του **if** για την `if (S1)...`. Το `else` είναι της `if (S2)...` . Για να εκτελεσθεί η *εντολή*-1 θα πρέπει να είναι αληθείς και η *S*1 και η *S*2 συγχρόνως. Για να εκτελεσθεί η *εντολή*-2 θα πρέπει να είναι αληθής η *S*1 και να είναι ψευδής η *S*2.

Μπορούμε δηλαδή να πούμε γενικότερα ότι το λεξικό σύμβολο `else` (και η εντολή που το ακολουθεί) αναφέρεται πάντα στο αμέσως προηγούμενο `if`. Αν όμως θέλουμε η *εντολή*-2 να εκτελεστεί σύμφωνα με την τιμή λογικής παράστασης *S*1 μόνον, θα πρέπει να δώσουμε υποχρεωτικώς την παρακάτω μορφή:

```
if ( S1 )  {  if ( S2 )  εντολή-1  }
     else  εντολή-2
```

Στην περίπτωση που μια εντολή **if** εμφανίζεται μέσα σε μιά άλλη **if**, λέγεται **φωλιασμένη** (nested) εντολή **if**.

Με φωλιασμένες **if** λύνουμε προβλήματα στα οποία έχουμε να επιλέξουμε μεταξύ εντολών που είναι περισσότερες από δύο. Στην περίπτωση αυτή βάζουμε δύο ή περίσσότερες **ifelse** τη μία μέσα στην άλλη· συνήθως βάζουμε την επόμενη **ifelse** στην περιοχή **else** της προηγούμενης. Ας ξεκινήσουμε με το

Παράδειγμα 1

Η ΔΕΗ χρεώνει την κατανάλωση ημερήσιου ρεύματος ώς εξής[1]:

- Για τις πρώτες 800 *kWh* (0 .. 800]: 0.088 €/*kWh*
- για τις επόμενες 400 *kWh* (800 .. 1200]: 0.112 €/*kWh*

[1] Η χρέωση γίνεται έτσι, με κλιμακούμενη αύξηση. Για τις τιμές... μην τα συζητάς. Ας πούμε ότι είναι έτσι.

- για τις επόμενες 500 *kWh* (1200 .. 1700]: 0.136 €/*kWh*
- για τις υπόλοιπες *kWh* (1700 .. +∞): 0.28 €/*kWh*

Να γραφεί πρόγραμμα που θα διαβάζει την κατανάλωση σε *kWh* και θα βγάζει το κόστος της ενέργειας που καταναλώθηκε[2].

Για να δούμε τι μας λέει ο παραπάνω πίνακας:

```
αν κατανάλωση <= 800 τότε
  κόστος = κατανάλωση*0.088
αλλιώς (είναι πάνω από 800)
  αν 800 < κατανάλωση <= 1200 τότε
    κόστος = 800*0.088 + (κατανάλωση - 800)*0.112
  αλλιώς (είναι πάνω από 1200)
    αν 1200 < κατανάλωση <= 1700 τότε
      κόστος = 800*0.088 + 400*0.112 + (κατανάλωση - 1200)*0.136
    αλλιώς (είναι πάνω από 1700)
      κόστος = 800*0.088 + 400*0.112 + 500*0.136 +
                    (κατανάλωση - 1700)* 0.28
```

Ας μεταφράσουμε σε **ifelse** την πρώτη σύγκριση:

```
if ( consumption <= 800 )
  cost = consumption*0.088
else
   ...
```

Η περιοχή του **else** θα εκτελεσθεί αν και μόνον αν η "`consumption <= 800`" έχει τιμή `false` ή αλλιώς "`consumption > 800`". Αν λοιπόν γράψουμε την επόμενη **ifelse** ως εξής:

```
if ( 800 < consumption && consumption <= 1200 ) ...
```

η πρώτη σύγκριση (και η λογική πράξη "`&&`") είναι άχρηστη, αφού στη θέση που το βάλαμε έχει σίγουρα τιμή `true`. Στο παρακάτω πρόγραμμα μπορείς να δεις πώς θα γραφούν οι πολλαπλές επιλογές μας.

```
#include <iostream>
using namespace std;
int main()
{
   double  consumption;    // Κατανάλωση σε kWh
   double  cost;

   cout << " Δώσε την κατανάλωση σε kWh: ";  cin >> consumption;
   if (consumption <= 800)
      cost = consumption * 0.088;
   else if (consumption <= (800 + 400))
      cost = 800*0.088 + (consumption - 800)*0.112;
   else if (consumption <= (800 + 400 + 500))
      cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
   else
      cost = 800*0.088 + 400*0.112 + 500*0.136
             + (consumption - 1700)*0.28;
   cout << cost << endl;
}
```

Γενικώς μπορούμε να γράψουμε:

```
if (S1)  E1
else if (S2)  E2
else if (S3)  E3
   :
else if (Sn)  En
else En+1
```

που θα εκτελεσθεί ως εξής:

[2] Ούτε νυκτερινό, ούτε πάγιο, ούτε ΦΠΑ, ούτε ΕΡΤ, ούτε τίποτε άλλο. Μόνον ημερήσιο!

αν η S_1 έχει τιμή `true` τότε

θα εκτελεσθεί η E_1 και θα αγνοηθούν οι E_2, E_3,... E_n,

αλλιώς (S_1 έχει τιμή `false`) αν η S_2 έχει τιμή `true` τότε

θα εκτελεσθεί η E_2 και θα αγνοηθούν οι E_1, E_3,... E_n,

αλλιώς (S_1 και S_2 έχουν τιμή `false`) αν η S_3 έχει τιμή `true` τότε

θα εκτελεσθεί η E_3 και θα αγνοηθούν οι E_1, E_2,... E_n,

. . .

αλλιώς (S_1, S_2,... S_{n-1} έχουν τιμή `false`) αν η S_n έχει τιμή `true` τότε

θα εκτελεσθεί η E_n και θα αγνοηθούν οι E_1, E_2,... $E_{n\text{-}1}$,

αλλιώς (S_1, S_2,... S_n έχουν τιμή `false`)

θα εκτελεσθεί η E_{n+1} και θα αγνοηθούν οι E_1, E_2,... E_n.

Αλλά προσοχή! Η σειρά που θα βάλεις τις συνθήκες των **if** είναι ουσιαστική. Οι εντολές της k-οστής **if** θα εκτελεστούν αν ισχύει η

$$(\texttt{!}\ S_1)\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ (\texttt{!}\ S_{k-1})\ \texttt{\&\&}\ S_k$$

ή αλλιώς:

$$(\texttt{!}\ (S_1\ \texttt{||}\ \ldots\ \texttt{||}\ S_{k-1}))\ \texttt{\&\&}\ S_k$$

Καμιά φορά, αν γράφεις απρόσεκτα, αυτή μπορεί να είναι ισοδύναμη με `false`! Για να δεις ένα (χονδροειδές) παράδειγμα, στο τελευταίο πρόγραμμα βάλε:

```
if ( consumption <= (800 + 400 + 500) )
   cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
else if ( consumption <= (800 + 400) )
   cost = 800*0.088 + (consumption - 800)*0.112;
else if ( consumption <= 800 )
   cost = consumption * 0.088;
else
  :
```

Για να εκτελεσθεί η

```
   cost = 800*0.088 + (consumption - 800)*0.112;
```

θα πρέπει να έχουμε: "`!(consumption <= 1700) && consumption <= 1200`". Η τιμή αυτής της παράστασης είναι `false`, όποια τιμή και αν έχει η *consumption*. Παρομοίως, `false` είναι πάντοτε και η "`!(consumption <= 1700) && !(consumption <= 1200) && consumption <= 100`" και έτσι αποκλείεται να εκτελεσθεί η: "`cost = consumption*0.088`".

Μερικοί προτιμούν να λύνουν αυτό το πρόβλημα με πολλές ανεξάρτητες **if**. Π.χ. για το παράδειγμά μας θα γράψουν:

```
if ( consumption <= 800 )
   cost = consumption * 0.088;
if ( 800 < consumption && consumption <= 1200 )
   cost = 800*0.088 + (consumption - 800)*0.112;
if ( 1200 < consumption && consumption <= 1700 )
   cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
if ( consumption < 1700 )
   cost = 800*0.088 + 400*0.112 + 500*0.136
          + (consumption - 1700)*0.28;
```

Φυσικά, το πρόγραμμα είναι μια χαρά. Ποιο είναι το πρόβλημά της; Κάνει πολλές πράξεις χωρίς λόγο.[3] Π.χ. αν του δώσεις κατανάλωση 50 τότε θα υπολογίσει τον λογαριασμό από την πρώτη **if**, αλλά συνέχεια θα υπολογίσει τις συνθήκες όλων των υπολοίπων **if** για να τις βγάλει `false` φυσικά.

[3] Χωρίς λόγο; Πρόσεξε ότι είναι σαφώς πιο σίγουρη από τις άλλες μορφές.

Παράδειγμα 2

Θέλουμε να γράψουμε ένα πρόγραμμα που θα προσομοιώνει τη λειτουργία ενός απλού υπολογιστή τσέπης (pocket calculator). Θα διαβάζει μια γραμμή όπου στην *πρώτη* θέση θα δίνεται το σύμβολο της πράξης (`+`, `-`, `*`, `/`) και στη συνέχεια δυο πραγματικοί αριθμοί με τους οποίους θα γίνει η πράξη.

Θα αποθηκεύουμε το σύμβολο της πράξης σε μια μεταβλητή τύπου `unsigned char` με το όνομα *oper*. Τις δυο πραγματικές τιμές θα τις αποθηκεύουμε σε δυο μεταβλητές *x*1, *x*2, τύπου `double`.

Αφού διαβάσαμε μια γραμμή θα πρέπει να ελέγξουμε την *oper*:

```
αν oper == '+' τότε
  Δώσε το άθροισμα
αλλιώς, η oper μπορεί να είναι '-', '*', '/'
  αν oper == '-' τότε
    Δώσε τη διαφορά
  αλλιώς, η oper μπορεί να είναι '*', '/'
    αν oper == '*' τότε
      Δώσε το γινόμενο
    αλλιώς, η oper μπορεί να είναι '/'
      Δώσε το πηλίκο
```

Μεταφράζοντας τα παραπάνω σε C++ θα έχουμε δυο **ifelse** που η κάθε μια τους έχει σαν περιοχή του else μια άλλη **ifelse**. Αλλά μάλλον δεν τελειώσαμε: Θα πρέπει να προβλέψουμε την πιθανότητα λάθους στην πράξη. Δεν είναι καθόλου σίγουρο ότι αν η πράξη δεν είναι '`+`', '`-`', '`*`' θα είναι '`/`'. Μπορεί ο χρήστης να δώσει κατά λάθος '`@`', ας πούμε. Θα πρέπει λοιπόν να ελέγχουμε την πράξη και για τη διαίρεση. Τέλος, δεν θα πρέπει να προσπαθήσουμε να υπολογίσουμε το πηλίκο αν ο διαιρέτης είναι "`0`" (μηδέν). Να το πρόγραμμά μας τελικώς:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;          // Τα ορίσματα της πράξης
   unsigned char oper;     // Το σύμβολο της πράξης

   cin >> oper >> x1 >> x2;
   if (oper == '+')        cout << (x1 + x2) << endl;
   else if (oper == '-')   cout << (x1 - x2) << endl;
   else if (oper == '*')   cout << (x1 * x2) << endl;
   else if (oper == '/')
      if (x2 != 0)          cout << (x1 / x2) << endl;
      else
         cout << " Δεν γίνεται" << endl;
   else
      cout << " Η πράξη (+,-,*,/) στην πρώτη θέση" << endl;
}
```

5.4 * Η Λογική των Εντολών Επιλογής

Ας δούμε τώρα τα εργαλεία που μας χρειάζονται για να κάνουμε αποδείξεις ορθότητας προγραμμάτων με εντολές **ifelse** και **if**.

Υποθέτουμε ότι έχουμε την εξής περίπτωση:

```
// P
   if (S)  Et else Ef
```

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* `&&` *S* και έστω ότι καταλήγουμε στην συνθήκη *Qt*. Αν δεν ισχύει η *S* –δηλαδή ισχύει η `!`*S*– τότε εκτελείται η εντολή *Ef* με

προϋπόθεση *P* **&&** (**!***S*) και έστω ότι καταλήγουμε στην *Qf*. Άρα, η συνθήκη που θα καταλήξουμε μετά την εκτέλεση της **ifelse** θα είναι η: *Qt* **||** *Qf*. Βλέπουμε λοιπόν ότι:

- ***η ifelse είναι ένας τρόπος για να πάρουμε συνθήκες που να περιέχουν*** **||**.

Επειδή όμως, όπως ξέρουμε, δεν γράφουμε εντολές στην τύχη για να δούμε πού θα καταλήξουμε, αλλά τις γράφουμε για να καταλήξουμε σε κάποια συγκεκριμένη συνθήκη, ας δούμε το πρόβλημά μας κάπως διαφορετικά:

```
// P
   if (S)  Et else Ef                                    (1)
// Q
```

Δηλαδή: είμαστε στην κατάσταση *P* και γράφουμε την `if (S) Et else Ef` με στόχο να καταλήξουμε στην *Q*. Τι γίνεται τώρα;

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S*. Αν το πρόγραμμά μας είναι σωστό θα πρέπει να φτάνουμε στην *Q*. Αν πάλι δεν ισχύει η *S* –δηλαδή ισχύει η **!***S*– τότε εκτελείται η εντολή *Ef* με προϋπόθεση *P* **&&** (**!***S*). Αν το πρόγραμμά μας είναι σωστό θα πρέπει και πάλι να φτάνουμε στην *Q*. Δηλαδή:

- ***Για να αποδείξουμε την ορθότητα της* (1) *πρέπει και αρκεί να αποδείξουμε την ορθότητα των:***

```
      P && S { Et } Q
P && (!S) { Ef } Q
```

Αυτός είναι ο συμπερασματικός κανόνας της **ifelse**, που γράφεται συμβολικώς ως εξής:

$$\frac{P \,\&\&\, S\,\{Et\}\,Q,\; P \,\&\&\, (!S)\,\{Ef\}\,Q}{P\,\{\textbf{if}\ (S)\ Et\ \textbf{else}\ Ef\}\,Q}$$

Παρομοίως ισχύουν για την **if**. Αν έχουμε:

```
// P
   if (S)  Et
```

τότε: αν μεν ισχύει η *S* εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S* και έστω ότι καταλήγουμε στην συνθήκη *Qt*. Αν δεν ισχύει η *S* –δηλαδή ισχύει η **!***S*– τότε *δεν εκτελείται* η εντολή *Et* και παραμένει η *P* **&&** (**!***S*). Άρα, η συνθήκη που θα καταλήξουμε μετά την εκτέλεση της **if** θα είναι η: *Qt* **||** (*P* **&&** (**!***S*)).

Ας δούμε και τον συμπερασματικό κανόνα της **if**. Ας πούμε ότι έχουμε:

```
// P
   if (S)  Et                                            (2)
// Q
```

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S*. Αν το πρόγραμμά μας είναι σωστό θα πρέπει να φτάνουμε στην *Q*. Αν πάλι δεν ισχύει η *S* (ισχύει η **!***S*) τότε θα πρέπει να φτάνουμε στην *Q* χωρίς να εκτελεστεί οποιαδήποτε εντολή. Αφού λοιπόν ισχύει η *P* **&&** (**!***S*) θα πρέπει από αυτήν να συνάγεται η *Q*. Δηλαδή:

- ***Για να αποδείξουμε την ορθότητα της* (2) *πρέπει και αρκεί να αποδείξουμε την ορθότητα των:***

```
P && S { Et } Q
(P && (!S)) ⇒ Q
```

Αυτός είναι ο συμπερασματικός κανόνας της **if**. Συμβολικώς:

$$\frac{P \,\&\&\, S\,\{Et\}\,Q,\; (P \,\&\&\, (!S)) \Rightarrow Q}{P\,\{\textbf{if}\ (S)\ Et\}\,Q}$$

Παράδειγμα

Για παράδειγμα χρήσης του κανόνα της **ifelse** ας αποδείξουμε ότι στο πρώτο πρόγραμμα της §5.2 βρίσκουμε σωστά το μέγιστο. Φυσικά, θα πρέπει να ξεκινήσουμε με τις προδιαγραφές.

Ξεκινούμε με την απαίτηση· θέλουμε να έχουμε τη *maxxy* μεγαλύτερη (ή ίση) και από τη *x* και από τη *y*: (*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*). Αλλά η τιμή της *maxxy* θα είναι ίση είτε με *x* είτε με *y*: (*maxxy* **==** *x*) **||** (*maxxy* **==** *y*). Να λοιπόν η απαίτηση:

((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*))

Τι προϋπόθεση θα έχουμε; Τίποτε ιδιαίτερο. Θα πρέπει να μπορούμε να επεξεργαστούμε *οποιεσδήποτε* τιμές και αν πάρουμε –στις *x*, *y*– από τη "`cin >> x >> y`". Δηλαδή, όπως είπαμε στην §3.7, θα έχουμε για προϋπόθεση:

`true`

Να λοιπόν τι πρέπει να αποδείξουμε:

```
// true
   if (x > y)  maxxy = x;
         else  maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Σύμφωνα με τον κανόνα της **ifelse**, αρκεί να αποδείξουμε ότι:

```
// true && (x > y)
   maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

και

```
// true && (!(x > y))
   maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Αυτές όμως αποδεικνύονται πολύ εύκολα.

1: Για να ισχύει η ((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*)) μετά τη "`maxxy = x`" θα πρέπει πριν από αυτήν να ισχύει η: ((***x*** ≥ *x*) **&&** (***x*** ≥ *y*)) **&&** ((***x*** **==** *x*) **||** (***x*** **==** *y*)), που παίρνουμε από την απαίτηση αν βάλουμε όπου *maxxy* το *x*. Αν πάρουμε υπόψη μας ότι

- ότι οι ***x*** ≥ *x* και ***x*** **==** *x* έχουν τιμή `true` για οποιαδήποτε τιμή της *x*,
- ότι (`true` **&&** A) ≡ A,
- ότι (`true` **||** A) ≡ `true`,

αυτή απλουστεύεται σε: ***x*** ≥ *y*. Αυτή όμως συνάγεται από την προϋπόθεση *x* > *y*.

2: Για να ισχύει η ((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*)) μετά τη "`maxxy = y`" θα πρέπει πριν από αυτήν να ισχύει η: ((***y*** ≥ *x*) **&&** (***y*** ≥ *y*)) **&&** ((***y*** **==** *x*) **||** (***y*** **==** *y*)). Παρομοίως και εδώ, αν παρουμε υπόψη μας ότι οι ***y*** ≥ *y* και ***y*** **==** *y* έχουν τιμή `true` για οποιαδήποτε τιμή της *y*, καθώς και τις ταυτολογίες που είδαμε παραπάνω, αυτή απλουστεύεται σε: ***y*** ≥ *x*. Αυτή όμως συνάγεται από την προϋπόθεση **!**(*x* > *y*) που σημαίνει: *x* ≤ *y*.

5.4.1 * Από το Τέλος προς την Αρχή

Στις αποδείξεις που κάναμε με την εντολή εκχώρησης, χρησιμοποιούσαμε το αξίωμα της εκχώρησης για βρούμε τη συνθήκη που θα πρέπει να ισχύει *πριν* από μια εντολή ώστε μετά από αυτή να έχουμε μια συγκεκριμένη απαίτηση. Εδώ θα δούμε πώς θα κάνουμε τα ίδια με τις εντολές **if**.

Το πρόβλημα που έχουμε να λύσουμε είναι το εξής: Ποια θα πρέπει να είναι η *P* ώστε να έχουμε:

```
   // P
      if (S)  Et else Ef
   // Q
```

Ας ονομάσουμε *Pt* και *Pf* τις συνθήκες για τις οποίες έχουμε:

```
// Pt                  // Pf
   Et        και          Ef
// Q                   // Q
```

Αν βρούμε τις *Pt* και *Pf* τότε η *P* είναι (δες και την Άσκ. 5-3):

$$(Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))$$

Παράδειγμα 1

Έστω ότι έχουμε να αποδείξουμε:

```
// (-1 ≤ x < 0) || (1 ≤ x < 6)
   if (x < 0)  y = x;
         else  y = x - 2;
// -1 ≤ y ≤ 4
```

Για να αποδείξουμε αυτό που ζητείται θα πρέπει:

1. να βρούμε, όπως είπαμε παραπάνω, την *P* ώστε:

```
// P
   if (x < 0)  y = x;
         else  y = x - 2;
// -1 ≤ y ≤ 4
```

2. να αποδείξουμε ότι η *P* συνάγεται από την προϋπόθεση (σ.κ. (*E*2), §0.4), δηλ.:

$((-1 \le x \le 0)$ `||` $(1 \le x \le 6)) \Rightarrow P$

Ξεκινούμε από το:

1. Οι αντιστοιχίες με το μοντέλο που δώσαμε πιο πάνω είναι:

- *Et* είναι η: `y = x;`
- *Ef* είναι η: `y = x - 2;`
- *Q* είναι η: $-1 \le y \le 4$

Θέλουμε λοιπόν να βρούμε την *Pt* και την *Pf* ώστε:

```
// Pt                        // Pf
   y = x;                       y = x - 2;
// -1 ≤ y ≤ 4                // -1 ≤ y ≤ 4
```

Για να έχουμε $-1 \le y \le 4$ μετά την `y = x` θα πρέπει να έχουμε πριν από αυτήν $-1 \le x \le 4$. Αυτή είναι η *Pt*.

Παρομοίως, για να έχουμε $-1 \le y \le 4$ μετά την `y = x - 2` θα πρέπει, πριν από αυτήν, να έχουμε $-1 \le x-2 \le 4$ ή αλλιώς $1 \le x \le 6$. Αυτή είναι η *Pf*.

Άρα, πριν από την **ifelse** θα πρέπει να ισχύει η: $(Pt$ `&&` $S)$ `||` $(Pf$ `&&` $(!S))$ ή αλλιώς, αν πάρουμε υπόψη μας ότι *S* είναι η $x < 0$ και `!`*S* η $!(x < 0)$ ή $(x \ge 0)$:

$$((-1 \le x \le 4) \;\&\&\; (x < 0)) \;||\; ((1 \le x \le 6) \;\&\&\; (x \ge 0))$$

ή αλλιώς: $(-1 \le x < 0)$ `||` $(1 \le x \le 6)$

Τώρα ερχόμαστε στο

2. Θα πρέπει να απόδείξουμε ότι η $(-1 \le x < 0)$ `||` $(1 \le x \le 6)$ συνάγεται από την προϋπόθεση $(-1 \le x < 0)$ `||` $(1 \le x < 6)$. Για να ισχύει η προϋπόθεση θα πρέπει να ισχύει είτε η $-1 \le x < 0$ είτε η $1 \le x < 6$ (είτε και οι δύο, αλλά στην περίπτωσή μας αυτό είναι αδύνατο). Αν ισχύει η πρώτη τότε η *P* ισχύει. Αν ισχύει η δεύτερη $(1 \le x < 6)$, τότε ισχύει και η $1 \le x \le 6$, άρα ισχύει και η *P*.

Επομένως το πρόγραμμά μας είναι σωστό.

Στην περίπτωση της **if** έχουμε το εξής πρόβλημα: Ποια θα πρέπει να είναι η *P* ώστε να έχουμε:

```
   // P
      if (S)  Et
   // Q
```

Ας ονομάσουμε, όπως παραπάνω, *Pt* τη συνθήκη για την οποία έχουμε:

```
// Pt
   Et
// Q
```

Αν βρούμε την *Pt*, η *P* είναι (δες και την Άσκ. 5-3):

$$(S \texttt{ \&\& } Pt) \texttt{ || } ((\texttt{! } S) \texttt{ \&\& } Q)$$

Παράδειγμα 2

Θα αποδείξουμε, από το τέλος προς την αρχή, την ορθότητα του τρίτου προγράμματος για τον υπολογισμό του μεγίστου, δηλαδή:

```
// true
   maxxy = y;
   if (x > maxxy)  maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Θα πρέπει πρώτα να βρούμε τη συνθήκη που θα πρέπει να ισχύει πριν από την **if**, δηλ. την $(S \texttt{ \&\& } Pt) \texttt{ || } ((\texttt{! } S) \texttt{ \&\& } Q)$. Στην περίπτωσή μας:

- S είναι η $x > maxxy$ και $\texttt{!}S$ είναι η $\texttt{!}(x > maxxy)$ ή $x \le maxxy$,
- Q είναι η $((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y))$

Το πρώτο βήμα είναι να υπολογίσουμε την Pt ώστε:

```
// Pt
   maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Όπως είδαμε και στο παράδ. της §5.5 η Pt είναι: $\boldsymbol{x} \ge y$.

$$((x > maxxy) \texttt{ \&\& } (x \ge y)) \texttt{ ||}$$
$$((x \le maxxy) \texttt{ \&\& } ((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y)))$$

ή, απλούστερα:

$$((x > maxxy) \texttt{ \&\& } (x \ge y)) \texttt{ ||}$$
$$(((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y)))$$

Έχουμε λοιπόν:

```
   maxxy = y;
// ((x>maxxy) && (x≥y)) ||
// ((maxxy≥x) && (maxxy≥y) && ((maxxy==x)||(maxxy==y)))
   if (x > maxxy)  maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Αντικαθιστώντας στη συνθήκη που βρήκαμε την y στη θέση της $maxxy$ παίρνουμε τη συνθήκη που θα πρέπει να ισχύει πριν από τη `maxxy = y`:

$$((x > y) \texttt{ \&\& } (x \ge y)) \texttt{ || } (((y \ge x) \texttt{ \&\& } (y \ge y)) \texttt{ \&\& } ((y \texttt{ == } x) \texttt{ || } (y \texttt{ == } y)))$$

Αυτή όμως απλουστεύεται:

- η $(x > y) \texttt{ \&\& } (x \ge y)$ ισοδυναμεί με $(x > y)$,
- οι $y \ge y$ και $y \texttt{ == } y$ έχουν τιμή `true`

και η συνθήκη μας γίνεται:

$$(x > y) \texttt{ || } (y \ge x)$$

Αυτή όμως είναι η `true` που είναι και η προϋπόθεσή μας.

5.5 Εξασφάλιση Προϋποθέσεων

Τώρα έχουμε στα χέρια μας και άλλα εργαλεία –εκτός από την *assert*– για να ελέγχουμε τις προϋποθέσεις για την εκτέλεση των προγραμμάτων μας.

Παράδειγμα 1

Οι υπολογισμοί του προγράμματος για το πρόβλημα της ελεύθερης πτώσης απαιτούν να διασφαλίσθεί η συνθήκη $h \ge 0$. Αυτό μπορεί να γίνει πολύ εύκολα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
```

```
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   // Υπολόγισε τα tP, vP
      tP = sqrt( (2/g)*h );
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   // Τύπωσε τα tP, vP
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
}
```

Εκείνο το «**false**» στο σχόλιο, μετά το **else**, τι σημαίνει; Ότι δεν υπάρχουν τιμές των *tP* και *vP* που να πληρούν την απαίτηση όταν $h < 0$.

Σύγκρινέ αυτό το πρόγραμμα με αυτό της §4.4 και διάλεξε αυτό που προτιμάς: *assert* ή **if**. Θα επανέλθουμε...

Παράδειγμα 2

Να ξαναγυρίσουμε στο παράδειγμα του τριωνύμου, για το οποίο, την τελευταία φορά, ανακαλύψαμε ότι ξεχάσαμε τον έλεγχο της *a*.

Πριν κάνουμε οποιονδήποτε άλλον υπολογισμό θα πρέπει να ελέγξουμε αν: $a \neq 0$ και $\Delta \geq 0$. Αυτή είναι η προϋπόθεσή μας:

Προϋπόθεση: $(a \neq 0)$ **&&** $(b^2 - 4ac \geq 0)$

Απαίτηση: $(x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a})$ **&&** $(x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a})$

Το παρακάτω πρόγραμμα, που είναι μια τροποποιημένη μορφή του προγράμματος της §5.2, πριν από όλα ελέγχει αν ισχύει η προϋπόθεση.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;    // οι συντελεστές της εξίσωσης
   double  x1, x2;     // οι ρίζες της εξίσωσης
   double  delta;      // η διακρίνουσα

   cout << "Δώσε τρεις πραγματικούς αριθμούς" << endl;
   cin >> a >> b >> c;
   cout << " a = " << a << "   b = " << b
        << "   c = " << c << endl;
   delta = b*b - 4*a*c;
   if ( a != 0  &&  delta >= 0 )
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
   else // εκτός προδιαγραφών
   {
      if ( a == 0 )
```

```
         cout << " Η εξίσωση είναι 1ου βαθμού" << endl;
      else
         cout << " Δεν υπάρχουν Πραγματικές Λύσεις" << endl;
   }
   cout << " ΤΕΛΟΣ" << endl;
}
```

Τώρα, όπως βλέπεις, τα *x1* και *x2* θα υπολογιστούν μόνο στην περίπτωση που $a \neq 0$ **και** $delta \geq 0$.

Παρατηρήσεις ▶

1. Να υπενθυμίσουμε ότι αυτή η σύγκριση, "`a != 0`", δεν έχει και τόσο νόημα, αφού η *a* είναι τύπου `double`. Αν η τιμή της *a* δεν έρχεται από το πληκτρολόγιο, αλλά από ενδιάμεσους υπολογισμούς, μπορεί να έχει μια πολύ μικρή (απόλυτη) τιμή, που να μην είναι μηδέν, αλλά να δημιουργεί πρόβλημα.
2. Όπως ξέρεις, μπορούμε να συνεχίσουμε την αναζήτηση λύσης ακόμη και στην περίπτωση που *a* `==` 0. Δες την Άσκ. 5-7.
3. Το να γράφουμε τις εντολές εκχώρησης για τις *x1* και *x2* όπως ξέρουμε από τα μαθηματικά μπορεί να φαίνεται όμορφο αλλά δεν είναι και το καλύτερο για το πρόγραμμά μας. Όπως θα μάθουμε αργότερα, η αφαίρεση είναι μια «κακή» πράξη για τους τύπους κινητής υποδιαστολής διότι μπορεί να προκαλέσει *απώλεια σημαντικών ψηφίων*. Θα δούμε λοιπόν έναν άλλον τρόπο που αποφεύγει την αφαίρεση:

```
      if ( b >= 0 )
         x1 = (-b - sqrt(delta))/(2*a);
      else // b < 0
         x1 = (-b + sqrt(delta))/(2*a);
      x2 = c/(a*x1);
```

Αν, ας πούμε, τα *a*, *b*, *c* έχουν τιμές 1, -1, -30 αντιστοίχως, θα εκτελεσθεί η

```
         x1 = (-b + sqrt(delta))/(2*a);
```

όπου έχουμε τον υπολογισμό $-(-1)+\sqrt{(-1)^2 - 4\cdot1\cdot(-30)} = 1 + \sqrt{121} = 1 + 11 = 12$ (*x1* `==` 6). Η άλλη λύση υπολογίζεται (*x2* `==` -5) από τη γνωστή ιδιότητα: *x1*`*`*x2* `==` *c*`/`*a*.

Αν όμως τα *a*, *b*, *c* έχουν τιμές 1, 11, 30, θα εκτελεσθεί η

```
         x1 = (-b - sqrt(delta))/(2*a);
```

όπου έχουμε τον υπολογισμό $-11 - \sqrt{11^2 - 4\cdot1\cdot30} = -11 - \sqrt{1} = -11 - 1 = -(11+1) = -12$ (*x1* `==` -6). Η άλλη λύση υπολογίζεται ως: *x2* `==` (30`/`1)/(-6) `==` -5.

Και στις δύο περιπτώσεις, οι δύο όροι του αριθμητή έχουν το ίδιο πρόσημο και έτσι «αποφεύγουμε την αφαίρεση».◀

Παράδειγμα 3

Ας έρθουμε στο παράδ. 2 της §5.4 (υπολογιστής τσέπης). Όπως λέγαμε, έχουμε πρόβλημα όταν πάρουμε σύμβολο πράξης που δεν είναι '`+`', '`-`', '`*`', '`/`'. Ακόμη, έχουμε πρόβλημα όταν η πράξη είναι '`/`' και ο διαιρέτης είναι "`0`" (μηδέν). Να λοιπόν οι προδιαγραφές μας:

Προϋπόθεση:

(*oper* `==` '`+`') `||` (*oper* `==` '`-`') `||` (*oper* `==` '`*`') `||`
((*oper* `==` '`/`') `&&` (*x*2 `!=` 0))

Απαίτηση:

((*oper* `==` '`+`') `&&` (*res* `==` *x*1 + *x*2)) `||` ((*oper* `==` '`-`') `&&` (*res* `==` *x*1 - *x*2)) `||`
((*oper* `==` '`*`') `&&` (*res* `==` *x*1 * *x*2)) `||` ((*oper* `==` '`/`') `&&` (*x*2 `!=` 0) `&&` (*res* `==` *x*1 / *x*2))

και να πώς ελέγχουμε την προϋπόθεση στο πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;           // Τα ορίσματα της πράξης
   unsigned char oper;      // Το σύμβολο της πράξης
```

```
   double res;              // Το αποτέλεσμα της πράξης

   cin >> oper >> x1 >> x2;
   if (oper == '+' || oper == '-' ||
       oper == '*' || (oper == '/' && x2 != 0))
   {
      if (oper == '+')            res = x1 + x2;
      else if (oper == '-')       res = x1 - x2;
      else if (oper == '*')       res = x1 * x2;
      else /* if (oper == '/') */ res = x1 / x2;
      cout << res << endl;
   }
   else
      cout << " ΛΑΘΟΣ" << endl;
}
```

Φυσικά και εδώ, όπως και στο προηγούμενο παράδειγμα, η σύγκριση `x2 != 0` δεν λέει και πολλά πράγματα.

5.6 Σειρά Εκτέλεσης των Πράξεων

Στο προηγούμενο κεφάλαιο, §4.1, όταν εξετάζαμε τη σειρά εκτέλεσης των πράξεων, αφήσαμε για αργότερα ένα «λεπτό» σημείο· ας το δούμε. Η C++ επιταχύνει τον υπολογισμό λογικών παραστάσεων ως εξής

- Στη λογική πράξη "`&&`" αν το πρώτο όρισμα υπολογιστεί "`false`" τότε το αποτέλεσμα της πράξης υπολογίζεται "`false`" χωρίς να υπολογιστεί το δεύτερο όρισμα.
- Στη λογική πράξη "`||`" αν το πρώτο όρισμα υπολογιστεί "`true`" τότε το αποτέλεσμα της πράξης υπολογίζεται "`true`" χωρίς να υπολογιστεί το δεύτερο όρισμα.

Τι επιπτώσεις μπορεί να έχουν τα παραπάνω στον προγραμματισμό; Ας δούμε ένα

Παράδειγμα

Ας πούμε ότι σε κάποιο πρόγραμμα θέλουμε να εκτελεσθούν οι εντολές E με την προϋπόθεση ότι $\sqrt{x} > y + 2$ και, φυσικά, ότι $x \geq 0$.

Με αυτά που ξέραμε μέχρι τώρα θα γράφαμε:

```
   if ( x >= 0 )
      if ( sqrt(x) > y + 2 )  E
```

Έτσι έχουμε σιγουριά ότι δεν θα γίνει απόπειρα υπολογισμού της τετραγωνικής ρίζας αν δεν έχουμε εξασφαλίσει ότι η x έχει τιμή μη αρνητική.

Αν πάρουμε υπόψη τον γρήγορο υπολογισμό της "`&&`" γράφουμε:

```
   if ( x >= 0 && sqrt(x) > y + 2 )  E
```

Αν η x έχει τιμή αρνητική τότε η "`x >= 0`" υπολογίζεται σε "`false`" και την ίδια τιμή παίρνει και ολόκληρη η συνθήκη *χωρίς να γίνει απόπειρα υπολογισμού της* `sqrt(x)`, οπότε οι E δεν εκτελούνται.

Αυτό το χαρακτηριστικό της C++ ονομάζεται **υπολογισμός παράστασης bool με βραχυκύκλωμα** (short-circuit boolean evaluation)[4] και το αναφέρουμε για να καταλαβαίνεις προγράμματα γραμμένα από άλλους και όχι για να τη χρησιμοποιείς κατ' ανάγκη. Ο πρώτος τρόπος, με τις δύο **if**, είναι πιο σίγουρος και σου δίνει πρόγραμμα πιο ευανάγνωστο.[5]

[4] Η «συγγενείς» προς τη C++ γλώσσες Java και η C# δίνουν διαφορετικούς τελεστές για τις βραχυκυκλωμένες πράξεις από αυτούς που δίνουν για τις συνήθεις.

[5] Φυσικά ο δεύτερος είναι πιο γρήγορος, αλλά άφησέ για αργότερα την ανησυχία για την ταχύτητα.

5.7 Τα ";" και Άλλα Λάθη

Καινούριες εντολές, πιο πολλές δυνατότητες για προγραμματισμό, αλλά και πιο πολλές πιθανότητες για λάθη.

Πρώτα απ' όλα πρόσεξε «το λάθος που θα κάνεις πάντα» (§4.1.1):

♦ ***Η σύγκριση για ισότητα γίνεται με το "==" και όχι με το "=".***

Από το μεταγλωττιστή περνάει εύκολα (με μια προειδοποίηση) και μετά δεν είναι εύκολο να το ανακαλύψεις, αν μάλιστα το πρόγραμμα είναι μεγάλο.

Ας έρθουμε τώρα στο ';'. Μερικοί έχουν τη συνήθεια να βάζουν αυτόν το χαρακτήρα κάθε φορά που τελειώνουν μια γραμμή. Γράφει λοιπόν κάποιος:

```
if ( x > y );
   maxxy = x;
else;
   maxxy = y;
```

Ο μεταγλωττιστής της Borland C++ θα δώσει μια προειδοποίηση και ένα λάθος. Συγκεκριμένα:

- Για την `if (x > y)` θα δώσει προειδοποίηση: «code has no effect» (κώδικας που δεν κάνει κάτι). Όπως είναι γραμμένη η εντολή, ο μεταγλωττιστής καταλαβαίνει ότι πρόκειται για μια **if** (όχι **ifelse**) που ζητάει: αν έχουμε `x > y` να εκτελεσθεί η κενή εντολή! Επομένως, η **if** δεν έχει οποιοδήποτε αποτέλεσμα.
- Για το `else` μας δίνει λάθος «misplaced else» (else σε λάθος θέση). Αφού κατάλαβε ότι έχουμε **if** και όχι **ifelse**, μας λέει ότι το **else** κακώς τοποθετήθηκε εκεί.

Μέχρι εδώ τα πράγματα δεν είναι και τόσο άσχημα: αφού μας βγάζει λάθος το βρίσκουμε, το διορθώνουμε και το ξαναδίνουμε:

```
if ( x > y )
   maxxy = x;
else;
   maxxy = y;
```

Ο μεταγλωττιστής σου το βρήκε εντάξει. Το δοκιμάζεις κιόλας: δίνεις στο *x* την τιμή 4, στο *y* την τιμή 5, σου βγάζει μεγαλύτερο το 5, όλα πάνε καλά. Κάνεις άλλη μια δοκιμή: 5 στο *x*, 4 στο *y*, σου βγάζει μεγαλύτερο το 4. Μα τι γίνεται; Τη μια δουλεύει, την άλλη δεν δουλεύει! Για να δούμε: Την πρώτη φορά η συνθήκη "`x > y`" ή "4 > 5" είναι `false`, άρα θα πρέπει να εκτελεσθεί η περιοχή του **else** που είναι... ποιά; Η *κενή εντολή!* Μόνο αυτή χωράει ανάμεσα στο **else** και το ';' που το ακολουθεί. Στη συνέχεια εκτελείται η εντολή που ακολουθεί την **ifelse**, η "`maxxy = y`" και παίρνεις σωστό αποτέλεσμα, αλλά τελείως συμπτωματικά. Τη δεύτερη φορά, η "`x > y`" παίρνει τιμή `true` και εκτελείται η περιοχή του **if**, "`maxxy = x`". Η *maxxy* παίρνει τη σωστή τιμή (5) και στη συνέχεια εκτελείται η εντολή που ακολουθεί την **ifelse**, "`maxxy = y`", που αλλάζει την τιμή της *maxxy* σε 4.

Πάντως, αν ψάξεις τα μηνύματα που σου στέλνει ο μεταγλωττιστής, θα βρεις μια προειδοποίηση για την εντολή: "`maxxy = x`" (περιοχή του **if**): «'maxxy' is assigned a value that is never used» (στη *maxxy* εκχωρείται μια τιμή που δεν χρησιμοποιείται ποτέ).

Παρόμοιο πρόβλημα θα έχεις και στην περίπτωση που θα βάλεις ';' μετά τη συνθήκη κάποιας **if**:

```
maxxy = y;
if (x > maxxy);
   maxxy = x;
```

Εδώ, περιοχή του **if** είναι η κενή εντολή και η "`maxxy = x`" είναι η επόμενη εντολή.

Συνοψίζοντας μπορούμε να πούμε:

♦ ***Δεν βάζουμε ποτέ*** ';'
  - ***μετά τη συνθήκη***
  - ***μετά το else.***

Τώρα που έμαθες τις εντολές **if** μπορείς να μάθεις κάτι που κάνουν οι πεπειραμένοι προγραμματιστές για να εντοπίσουν λάθη στα προγράμματά τους. Ορίζουν μια σταθερά:

```
const bool debug( true );
```

Στη συνέχεια, αντί για απλές "`cout << ...` ", που τυπώνουν τις τιμές των συνθηκών επαλήθευσης, όπως είδαμε στην §3.2, βάζουν εντολές:

```
if ( debug )  cout << ...
```

Όταν ανακαλύψουν τα λάθη τους και τα διορθώσουν, αλλάζουν την αρχική δήλωση σε:

```
const bool debug( false );
```

Έτσι, βλέπουν την εκτέλεση του προγράμματος χωρίς τα ενδιάμεσα αποτελέσματα. Μόλις παρουσιαστούν τα νέα λάθη, που η πείρα τους λέει ότι πρέπει να τα περιμένουν[6], ξαναλλάζουν την τιμή της *debug* σε `true`.[7]

Αυτά βέβαια στην περίπτωση που δεν διαθέτουν **Βοηθητικά Προγράμματα για Ανεύρεση Λαθών** (debugging[8] utilities, debuggers), που σου δίνουν τη δυνατότητα να παίρνεις ενδιάμεσα αποτελέσματα με πιο απλούς τρόπους.

5.8 Τι (Πρέπει να) Έμαθες

Τώρα πρέπει να ξέρεις να γράφεις προγράμματα που να έχουν περισσότερους από έναν κλάδους εκτέλεσης.

Θα πρέπει να μπορείς να χρησιμοποιείς συνθήκες για να επιλέγεις ποιες εντολές θα εκτελεσθούν και ποιες όχι, με βάση τις τιμές που έχουν οι μεταβλητές του προγράμματός σου κατά τη διάρκεια της εκτέλεσης. Θα πρέπει να μπορείς να «φωλιάζεις» εντολές **if** ή/και **ifelse** τη μια μέσα στην άλλη.

Θα πρέπει να ξέρεις να ελέγχεις αν τα δεδομένα που διαβάζεις συμμορφώνονται με τις προδιαγραφές.

Τέλος, αν μελετάς και τις παραγράφους με τα αστεράκια, θα πρέπει να μπορείς να αποδείξεις την ορθότητα προγραμμάτων με εντολές **if** ή/και **ifelse**. Θα ξέρεις επίσης και πώς διαπλέκονται οι συνθήκες επαλήθευσης με αυτές που χρησιμοποιούμε στις εντολές επιλογής.

Ασκήσεις

Α Ομάδα

5-1 Αν πριν από κάθε μια εντολή $a = 0$, $b = 1$, $x = 2$, $y = 3$, τι τιμές θα έχουν οι μεταβλητές μετά την εκτέλεση κάθε μιας από τις παρακάτω εντολές;

α) `if (x > y)  x = y; else y = x;`
β) `if (x == y)  a = b; else b = a;`
γ) `if (x > 0)  a = b;`
δ) `if (x = y)  y = x;`
ε) `if (a = sqrt(b))  y = x;`
στ) `if (x > x)  if (y < y)  a = b;`

---

[6] There is always one more bug! (νόμος του Murphy).

[7] Επειδή αυτοί οι έλεγχοι κάνουν το πρόγραμμα κάπως πιο αργό, συνήθως βάζουμε (παρόμοιες) οδηγίες προς τον μεταγλωττιστή όπως θα μάθουμε αργότερα.

[8] Θα δεις στα ελληνικά τον όρο «εκσφαλμάτωση».

ζ) `if (a + b)  y = x;`

5-2 Ο αλγόριθμος υπολογισμών προβλέπει, ότι η μεταβλητή *y* πρέπει να πάρει τις παρακάτω τιμές, ανάλογα με τις τιμές των μεταβλητών *a* και *b*:

$y = a + 100$, αν $a \geq 0$ και $b = 0$
$y = b + 5$, αν $a \geq 0$ και $b \neq 0$
$y = b - 5$, αν $a < 0$ και $b = 0$
$y = a - 100$, αν $a < 0$ και $b \neq 0$

Δώσε τις αντίστοιχες εντολές στην C++.

5-3 α) Είπαμε παραπάνω ότι στην *P* { **if** (*S*) *Et* **else** *Ef* } *Q* η *Et* εκτελείται με προϋπόθεση *P* **&&** *S* και μας δίνει *Q*, ενώ η *Ef* με προϋπόθεση *P* **&&** (**!***S*) και μας δίνει *Q*.

Στη συνέχεια είπαμε ότι, πηγαίνοντας από το τέλος προς την αρχή, αν *Pt* και *Pf* είναι τέτοιες ώστε: *Pt* { *Et* } *Q* και *Pf* { *Ef* } *Q* τότε πριν από την **ifelse** θα πρέπει να ισχύει η (*Pt* **&&** *S*) **||** (*Pf* **&&** (**!***S*)).

Για να είναι τα παραπάνω συμβιβαστά θα πρέπει:

$$(((Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))) \;\&\&\; S) \Rightarrow Pt$$

και

$$(((Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))) \;\&\&\; (!S)) \Rightarrow Pf$$

Να τις αποδείξεις!

β) Είπαμε ακόμη ότι στην *P* { **if** (*S*) *Et* } *Q* η *Et* εκτελείται με προϋπόθεση *P* **&&** *S* και μας δίνει *Q*, ενώ (*P* **&&** (**!***S*)) ⇒ *Q*.

Στη συνέχεια είπαμε ότι, πηγαίνοντας από το τέλος προς την αρχή, αν *Pt* είναι τέτοια ώστε: *Pt* { *Et* } *Q* τότε πριν από την **if** θα πρέπει να ισχύει η (*Pt* **&&** *S*) **||** (*Q* **&&** (**!***S*)).

Για να είναι τα παραπάνω συμβιβαστά θα πρέπει:

$$(((Pt \;\&\&\; S) \;||\; (Q \;\&\&\; (!S))) \;\&\&\; S) \Rightarrow Pt$$

και

$$(((Pt \;\&\&\; S) \;||\; (Q \;\&\&\; (!S))) \;\&\&\; (!S)) \Rightarrow Q$$

Να τις αποδείξεις και αυτές!

Β Ομάδα

5-4 Ποια μαθηματική συνάρτηση υλοποιούν οι παρακάτω εντολές (όπου οι μεταβλητές *a*, *b*, *c* και *d* είναι τύπου `int`):

```
if ( a > b )  y = a; else y = b;
if ( c > y )  y = c;
if ( d > y )  y = d;
```

5-5 Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 30 € για κάθε παιδί που έχει, αν έχει μέχρι τρία (3) παιδιά. Αν έχει περισσότερα από 3 παιδιά η προσαύξηση είναι 50 € για κάθε παιδί. Γράψε πρόγραμμα που θα διαβάζει τον βασικό μισθό και τον αριθμό παιδιών ενός υπαλλήλου και θα υπολογίζει το οικογενειακό επίδομα και τον συνολικό μισθό.

5-6 (Σύνθεση των Ασκ. 2-14 και 2-15) Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 (θετικούς αριθμούς) και θα υπολογίζει και θα τυπώνει τη συνολική αντίσταση *R*. Πριν πάρει τις τιμές των αντιστάσεων, το πρόγραμμα θα ζητάει να διαβάσει, από το πληκτρολόγιο, τον τρόπο σύνδεσης των αντιστάσεων. Οι δεκτές απαντήσεις θα είναι:

- `'P'` για παράλληλη σύνδεση,
- `'S'` για σύνδεση εν σειρά.

Να διατυπωθεί η προϋπόθεση και το πρόγραμμα να την ελέγχει.

5-7 Ξαναγράψε το πρόγραμμα για το τριώνυμο της §5.6, έτσι ώστε:

1. Στην περίπτωση που $a = 0$ να υπολογίζει την απλή λύση $x1 = -c/b$, αν $b \neq 0$.
2. Αν $a = b = 0$ να εξετάζει τις περιπτώσεις $c = 0$ και $c \neq 0$.

3. Στην περίπτωση που *delta* < 0 να υπολογίζει και να τυπώνει τα

$ReX1 = ReX2 = -b/(2a)$

$ImX1 = ImX2 = \sqrt{-delta}\ /\ (2a)$

Η εκτύπωση των μιγαδικών θα πρέπει να γίνεται στη μορφή:

"`(`", *πραγματικό μέρος*, "`,`", *φανταστικό μέρος*, "`)`"

αφού δοθεί μήνυμα ότι οι ρίζες είναι μιγαδικές.

Γ Ομάδα

5-8 Απόδειξε ότι (`int a, b, c, y`):

```
// true
   if ( a > b )  y = a; else y = b;
   if ( c > y )  y = c;
// (y == a || y == b || y == c) && (y ≥ a && y ≥ b && y ≥ c)
```

5-9 Θέλουμε να αντικαταστήσουμε, στην προϋπόθεση του προβλήματος της δευτεροβάθμιας εξίσωσης, τη σύγκριση $a \neq 0$ με κάτι πιο «ρεαλιστικό». Ας υποθέσουμε ότι δεν έχουμε υπερχείλιση κατά τον υπολογισμό των $-b + \sqrt{b^2 - 4ac}$ ή της $-b - \sqrt{b^2 - 4ac}$. Ποια συνθήκη θα πρέπει να πληρούται ώστε να μην έχουμε υπερχείλιση όταν υπολογίζονται οι ρίζες;

5-10 (Συμπλήρωση της Ασκ. 2-7). Γράψε πρόγραμμα που θα διαβάζει τις καρτεσιανές συντεταγμένες ενός σημείου στο επίπεδο και θα υπολογίζει και θα τυπώνει τις πολικές r και ϕ. Αλλά, προσοχή: η atan επιστρέφει τιμή στο $(-\pi/2, \pi/2)$ ενώ θα πρέπει να έχουμε: $-\pi < \phi \leq \pi$.

Αν το σημείο βρίσκεται στον άξονα y, δηλ.: $x = 0$, τότε η ϕ θα γίνεται:

- $\pi/2$ αν $y > 0$,
- $-\pi/2$ αν $y < 0$,

ενώ θα παραμένει αόριστη αν έχουμε και $y = 0$.

Σύγκρινε τη ϕ που παίρνεις με αυτό που σου δίνει η atan2(y, x).

5-11 Απόδειξε ότι:

```
// (ia ≤ na || (ia = na+1) && ib ≤ nb)
   if ( ib > nb || ia ≤ na )  ia = ia + 1;
                      else  ib = ib + 1;
// ia ≤ na + 1
```

5-12 Απόδειξε ότι:

```
// true
   if ( x > y )  maxxy = x;
   if ( x <= y )  maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

5-13 Ένας άλλος τρόπος για να διατυπώσουμε την απαίτηση στο πρόγραμμα για το μέγιστο είναι ο εξής:

$$((x \geq y)\ \&\&\ (maxxy == x))\ ||\ ((y \geq x)\ \&\&\ (maxxy == y))$$

Απόδειξε ότι:

```
// true
   if (x > y)  maxxy = x;
         else  maxxy = y;
// ((x ≥ y) && (maxxy == x)) || ((y ≥ x) && (maxxy == y))
```

5-14 Ψάξε τα εγχειρίδια του ΗΥ σου να δεις ποιό είναι το σύνολο χαρακτήρων που χρησιμοποιεί.

Γράψε ένα πρόγραμμα που θα διαβάζει ένα χαρακτήρα από το πληκτρολόγιο και θα τυπώνει το μήνυμα:

- `"ψηφίο"` αν ο χαρακτήρας ήταν ψηφίο,
- `"γράμμα"` αν ο χαρακτήρας ήταν γράμμα,
- `"κάτι άλλο"` αν δεν είναι ούτε γράμμα ούτε ψηφίο.

Αν έχεις κεφαλαία και μικρά γράμματα θα πρέπει να διαχωρίζονται επίσης.

Δώσε δύο λύσεις:

- Χρησιμοποιώντας τις συναρτήσεις που είδαμε στο προηγούμενο κεφάλαιο.
- Χωρίς να χρησιμοποιήσεις τις συναρτήσεις. Στην περίπτωση αυτή, αν έχεις και Ελληνικά και είναι εύκολος ο διαχωρισμός, να διαχωρίζονται επίσης.

κεφάλαιο

6

Επαναλήψεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κάνεις το δεύτερο βήμα στη χρήση συνθηκών για τον έλεγχο εκτέλεσης ενός προγράμματος: Να προγραμματίζεις επαναλαμβανόμενη εκτέλεση (ομάδων) εντολών.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις προγράμματά που θα έχουν *επαναληπτικούς* υπολογισμούς. Με αυτά αρχίζεις να εκμεταλλεύεσαι τη μεγάλη δύναμη του υπολογιστή: Πολύ μεγάλη ταχύτητα στην εκτέλεση επαναληπτικών αλγορίθμων.

Έννοιες κλειδιά:

- *εντολές επανάληψης - βρόχοι*
- *εντολή* `while`
- *εντολή* `for`
- *αναλλοίωτη επανάληψης*
- *τιμή-φρουρός*
- *τερματισμός επανάληψης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Στον προγραμματισμό παρουσιάζεται συχνά η ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Φυσικά, αυτό που μας ενδιαφέρει είναι να γράψουμε τις εντολές μια φορά μόνον. Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει αυτήν τη δυνατότητα.

6.1 Επαναλήψεις

Ας ξεκινήσουμε με ένα παράδειγμα.

Έστω ότι θέλουμε να υπολογίσουμε και εκτυπώσουμε το άθροισμα και τη μέση αριθμητική τιμή *n* (π.χ.10) πραγματικών αριθμών –ας τους πούμε t_1, t_2,... t_n– που δίνονται στην είσοδο του υπολογιστή.

Με όσα ξέρουμε μέχρι τώρα, θα έπρεπε να δηλώσουμε *n* μεταβλητές *x*1, *x*2, ... *x*10 και να δώσουμε τις εντολές:

```
cin >> x1;  cin >> x2; ... cin >> x10;
sum = x1 + x2 + ... + x10;
```

Βέβαια, ένα τέτοιο πρόγραμμα θα ήταν τρομακτικό, αν μάλιστα το *n* δεν είναι 10, αλλά είναι 100 ή 1000 ή 10000.

Ας προσπαθήσουμε να το γράψουμε αλλιώς. Κοίταξε την παρακάτω εναλλακτική λύση:

```
sum = 0;                    // sum == 0
cin >> x;  sum = sum + x;   // (x == t1) && (sum == Σ(j:1..1)tj)
cin >> x;  sum = sum + x;   // (x == t2) && (sum == Σ(j:1..2)tj)
  . . .
cin >> x;  sum = sum + x;   // (x == tN) && (sum == Σ(j:1..n)tj)
```

Τι κάνει το (κάθε) *m*-οστό ζευγάρι εντολών

```
cin >> x;  sum = sum + x;
```

Διαβάζει από το πληκτρολόγιο τη *m*-οστή τιμή (t_m), την αποθηκεύει στη θέση της μνήμης *x* και την προσθέτει στη μεταβλητή *sum*. Όταν εκτελεσθεί το επόμενο ζευγάρι *η προηγούμενη τιμή που υπάρχει στη x θα σβηστεί*. Αυτό δεν μας δημιουργεί πρόβλημα, αφού από τις προδιαγραφές του προγράμματος φαίνεται ότι ο μόνος προορισμός κάθε τιμής που διαβάζεται είναι να προστεθεί στο άθροισμα.

Με τη *sum* τι γίνεται; Αρχικώς της δίνουμε την τιμή 0. Μετά το πρώτο ζευγάρι εντολών η τιμή της είναι η πρώτη τιμή που διαβάστηκε. Μετά το δεύτερο ζευγάρι η τιμή της *sum* είναι το άθροισμα των δύο πρώτων τιμών, μετά το *m*-οστό ζευγάρι η *sum* έχει ως τιμή το μερικό άθροισμα των *m* πρώτων τιμών. Μετά το *n*-οστό ζευγάρι η *sum* έχει ως τιμή το άθροισμα των *n* τιμών που πληκτρολογήθηκαν.

Όλα αυτά φαίνονται και στις συνθήκες που βάζουμε μετά το κάθε ζευγάρι εντολών. Με το `Σ(j: 1..m)tj` εννοούμε το: $\sum_{j=1}^{m} t_j$.

Ωραία λοιπόν! Φαίνεται ότι και αυτές οι εντολές λύνουν το πρόβλημα. Τι κερδίσαμε που τις γράψαμε; Η λύση του προβλήματός μας διατυπώθηκε ως επανάληψη μιας ακολουθίας εντολών: οι εντολές "`cin >> x;  sum = sum + x;`" εκτελούνται *n* φορές. Μπορούμε να πούμε στον υπολογιστή να τις εκτελέσει *n* φορές με αρκετά συνοπτικό τρόπο:

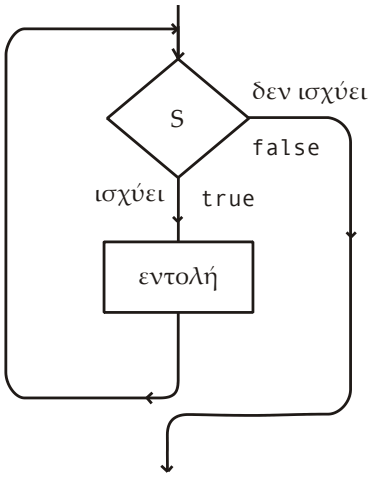
```
sum = 0;  m = 1;
όσο (m <= n) να εκτελείς την εντολή:
{
    cin >> x;  sum = sum + x;
 // (x == tm) && (sum == Σ(j:1..m)tj)
    m = m + 1;
}
```

Αυτό που εννοούμε εδώ είναι το εξής: όσο βρίσκει το $m \le n$ να εκτελεί τις τρεις εντολές που «πακετάραμε» σε μια σύνθετη εντολή. Η *m* είναι ένας **μετρητής** (counter) –συνήθως μεταβλητή τύπου (`unsigned`) `int`. Η τελευταία επαναλαμβανόμενη εντολή αυξάνει κάθε φορά την τιμή της *m* κατά 1· έτσι, αφού ξεκινάει από 1, όταν θα πάψει να ισχύει η συνθήκη "`m <= n`" η σύνθετη εντολή θα έχει εκτελεσθεί *n* φορές. Ακόμη, πρόσεξε ότι, αφού η *m* ξεκινάει από 1 και αυξάνεται κατά 1, είναι σίγουρο ότι κάποτε το *m* θα γίνει μεγαλύτερο από το *n*.

Το ίδιο πράγμα γράφεται στην C++:

```
sum = 0;  m = 1;
while ( m <= n )  {  cin >> x;  sum = sum + x;
                  // (x == tm) && (sum == Σ(j:1..m)tj)
                     m = m + 1;
                  }
```

Το **while** (που σημαίνει: όσο, για όσο, εφ' όσον) είναι λεξικό σύμβολο της C++. Η εντολή **while** είναι μια **επαναληπτική εντολή** (repetitive statement) και στο Πλ. 6.1 μπορείς να δεις την περιγραφή της. Η λειτουργία της εντολής **while** δίνεται επίσης από το λογικό διάγραμμα του Σχ. 6-1. Βλέποντάς το, μπορείς να κατάλάβεις γιατί οι επαναληπτικές εντολές λέγονται και **εντολές ανακύκλωσης** ή **βρόχοι** (loops).



Σχ. 6-1 Λογικό διάγραμμα της **while**.

Η συνθήκη $(x = t_m)$ **&&** $(sum = \sum_{j=1}^{m} t_j)$ δεν παύει να ισχύει, στο σημείο του προγράμματος που τη γράψαμε, όσο εκτελούνται ξανά και ξανά οι επαναλαμβανόμενες εντολές· λέμε λοιπόν ότι είναι **αναλλοίωτη** (invariant) της επανάληψης. Για αναλλοίωτες θα συζητήσουμε πιο εκτεταμένα παρακάτω.

Πρόσεξε το εξής: η **while**, όπως και η **if**, περιμένει μετά τη συνθήκη *μια* εντολή· αν θέλουμε να βάλουμε περισσότερες, όπως είδες ήδη στο παράδειγμά μας, τις «συσκευάζουμε» με ένα άγκιστρο (**{**) στην αρχή και ένα άγκιστρο (**}**) στο τέλος σε μια *σύνθετη εντολή*.

Άλλα παραδείγματα εντολών **while**:

```
while ( a > b )  a = a - b;
```

```
while ( n > 0 )
{
   a = a + n*n;
   n = n - 1;
}
```

Ας επιστρέψουμε τώρα στο πρόβλημά μας που διατυπώθηκε στο παράδειγμα. Το παρακάτω πρόγραμμα –Μέση Τιμή 1– είναι μια λύση με χρήση της εντολής **while**:

```
// πρόγραμμα: Μέση Τιμή 1
#include <iostream>
using namespace std;
int main()
{
   const int n( 10 );

   int m;       // Μετρητής των επαναλήψεων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  m = 1;
   while ( m <= n )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;          // x = tm
      sum = sum + x;           // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;           // Ετοιμαζόμαστε για τον επόμενο αριθμό
   } // while
   avrg = sum / n;
   cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
}
```

Πλαίσιο 6.1

Η Εντολή while

"`while`", "`(`", *συνθήκη*, "`)`", *εντολή*

Συστατικά:

- το `while` είναι λεξη-κλειδί,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή* είναι μια οποιαδήποτε εντολή της C++ και λέγεται *περιοχή της επανάληψης*.

Εκτέλεση της εντολής **while**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο `while`.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη.

Υπολογίζεται ξανά η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο **while**.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη κ.ο.κ.

Αν η τιμή της συνθήκης βρεθεί `false` τότε η εντολή που ακολουθεί τη συνθήκη δεν εκτελείται και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί τη **while**.

Αν η τιμή της συνθήκης είναι `false` την πρώτη φορά που θα ελεγχθεί, η εντολή που ακολουθεί τη συνθήκη δεν θα εκτελεστεί ποτέ!

Οι εντολές "`sum = 0; m = 1`", που δίνονται πριν από την εντολή **while**, εκτελούνται βέβαια μόνο μια φορά και αποτελούν το *μέρος προετοιμασίας* της επανάληψης, δηλ. το μέρος όπου καθορίζονται οι αρχικές τιμές μεταβλητών, που χρησιμοποιούνται μέσα στην περιοχή της **while**.

Στο πρόγραμμά μας το *n*, που καθορίζει το πλήθος των επαναλήψεων της **while**, είναι σταθερά (εδώ *n*=10), και επομένως η επαναληπτική διαδικασία δεν μπορεί να σταματήσει αν δεν διαβαστούν *n* αριθμοί. Μια τέτοια επιλογή όμως, που γίνεται *όταν γράφουμε το πρόγραμμα*, οδηγεί σε ένα πρόγραμμα εξαιρετικώς ανελαστικό. Πιο βολικές είναι οι λύσεις που μας επιτρέπουν να καθορίζουμε το πλήθος των αρχικών δεδομένων (στο παράδειγμά μας τον αριθμό *n*), *όταν αρχίζει η εκτέλεση του προγράμματος*. Έτσι δεν χρειάζεται αλλαγή (εδώ του ορισμού *n* = 100) και νέα μεταγλώττιση του προγράμματος κάθε φορά που αλλάζει το πλήθος των στοιχείων εισόδου. Σε ένα τέτοιο πρόγραμμα –ας το πούμε Μέση Τιμή 2– το *n* είναι *μεταβλητή* και η τιμή της πληκτρολογείται από το χρήστη πριν αρχίσει να πληκτρολογεί τους άλλους αριθμούς:

```
int n;       // Τώρα το πλήθος είναι μεταβλητή
   . . .
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
   . . .
```

Τώρα όμως προσοχή: Θα πρέπει να έχουμε ως προϋπόθεση *n* > 0 και να την ελέγχουμε μόλις διαβάσουμε την τιμή της *n*:

```
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
if ( n > 0 )
{
   sum = 0;  m = 1;
     . . .
}
```

Πλαίσιο 6.2α
Υπολογισμός Αθροίσματος

```
sum = 0;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   sum = sum + x;
      . . .
} // while
// sum == Σx
```

Πλαίσιο 6.2β
Υπολογισμός Γινομένου

```
prod = 1;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   prod = prod * x;
      . . .
} // while
// prod == Πx
```

```
else
   cout << " Το πλήθος πρέπει να είναι θετικό" << endl;
```

Αφήνουμε σε σένα, για άσκηση, να γράψεις ολόκληρο το Μέση Τιμή 2 (Άσκ. 6-1).

Όπως θα δεις και στη συνέχεια, έτσι υπολογίζουμε τα αθροίσματα ακόμη και αν ο έλεγχος της επανάληψης είναι διαφορετικός ή αν οι τιμές δημιουργούνται από το πρόγραμμα και δεν διαβάζονται από το πληκτρολόγιο. Σου το δίνουμε λοιπόν σαν συνταγή στο Πλ. 6.2α. Παρόμοιος είναι και ο υπολογισμός γινομένου (Πλ. 6.2β).

Αυτή η μορφή επανάληψης –που είδαμε στα παραπάνω παραδείγματα– είναι η πιο απλή και λέγεται **μετρούμενη** (counted) επανάληψη. Θα ασχοληθούμε με αυτήν ξανά στη συνέχεια.

6.1.1 Άγνωστο Πλήθος Στοιχείων – Τιμή-Φρουρός

Συχνά το πλήθος των στοιχείων που θέλουμε να δώσουμε στον υπολογιστή δεν είναι γνωστό. Σκέψου, για παράδειγμα, μερικά φύλλα με μετρήσεις από κάποιο πείραμα. Κάθε φορά που τα μετράς βρίσκεις και διαφορετικό πλήθος. Δεν είναι καλύτερο να τα μετρήσει το πρόγραμμα;

Έστω ότι θέλουμε να βρούμε και να τυπώσουμε το άθροισμα, τη Μέση Τιμή και το πλήθος πραγματικών αριθμών που θα πληκτρολογηθούν στην είσοδο του Υπολογιστή. *Ενώ το πλήθος τους δεν είναι γνωστό*, ξέρουμε ότι οι αριθμοί μας είναι *μη-μηδενικοί*.

Το πώς θα υπολογίσουμε το άθροισμα και τη μέση τιμή το ξέρουμε· δεν ξέρουμε όμως *πότε θα σταματήσει η επανάληψη*. Ή αλλιώς: *πώς θα πούμε στον Υπολογιστή ότι τελείωσαν τα στοιχεία εισόδου* και να μην περιμένει άλλα.

Είναι φανερό ότι την ώρα που εκτελείται το πρόγραμμα και δίνουμε τα στοιχεία εισόδου θα πρέπει να δώσουμε στον ΗΥ το σύνθημα «τέλος στοιχείων» με κάποιο τρόπο. Αλλά ποιά είναι η δυνατότητά μας να δώσουμε κάτι στον ΗΥ; Καθορίζεται με απόλυτη ακρίβεια από την εντολή: "`cin >> x`" που εκτελείται ξανά και ξανά. Μπορούμε λοιπόν να δώσουμε μια τιμή στη *x* που να είναι συνθηματική και να σημαίνει «τέλος». Ας δούμε τώρα, τι τιμές μπορεί να πάρει η *x*: τιμές τύπου **`double`**. Και τώρα γεννιούνται τα ερωτήματα: Πώς δεν θα γίνει μπέρδεμα; Γιατί ο ΗΥ δεν θα πάρει να επεξεργαστεί και τη συνθηματική τιμή όπως όλες τις άλλες; Και αν θέλουμε να δώσουμε αυτήν την τιμή για επεξεργασία;

Στο παράδειγμά μας, θα χρησιμοποιήσουμε την πληροφορία ότι *τα στοιχεία είναι μη μηδενικά*. Σχεδιάζουμε λοιπόν το πρόγραμμά μας με βάση την εξής σύμβαση: *Όταν τελειώσουν τα στοιχεία που θέλουμε να δώσουμε θα πληκτρολογήσουμε την τιμή "`0`"*. Πρόσεξε ότι:

- Το "`0`" είναι δεκτή απάντηση στην εντολή: `cin >> x`.
- Από τις προδιαγραφές του προβλήματος ξέρουμε ότι αποκλείεται να δώσουμε το 0 για επεξεργασία.

Το "0" λέγεται **τιμή-φρουρός** (sentinel value) και αυτή η τεχνική χρησιμοποιείται ευρύτατα από τους προγραμματιστές. Μπορούμε λοιπόν να πούμε ότι η **while** θα ξεκινάει ως εξής:

```
while ( x != sentinel )
```

Τώρα όμως κάναμε μια σοβαρή αλλαγή στη λογική του προγράμματος που είχαμε στα προηγούμενα παραδείγματα: Όταν έρχεται η στιγμή να εκτελεσθεί η **while** πρέπει να έχουμε την πρώτη τιμή της *x*. Η πρώτη ανάγνωση θα πρέπει να γίνεται πριν από την **while**:

```
cin >> x;
while ( x != sentinel )
```

Αλλαγών συνέχεια: Μέχρι τώρα η περιοχή της **while** είχε τη μορφή:

```
{
    cin >> x;
    Επεξεργάσου την τιμή που διάβασες στη x
}
```

Τι θα γίνει αν την αφήσουμε έτσι; Θα διαβάσουμε τη δεύτερη τιμή πριν επεξεργαστούμε την πρώτη. Θα πρέπει λοιπόν να αντιστρέψουμε τη σειρά των εντολών:

```
cin >> x;
while ( x != sentinel )
{
   Επεξεργάσου την τιμή που διάβασες στη x;
   cin >> x;      // Διάβασε την επόμενη τιμή
}
```

Και η τελευταία αλλαγή: Ο μετρητής μας, κάθε στιγμή, θα πρέπει να δείχνει πόσες τιμές διαβάστηκαν για επεξεργασία. Θα πρέπει να ξεκινάει από 0 και όχι από 1. Το παρακάτω πρόγραμμα, Μέση Τιμή 3, είναι μια λύση του προβλήματος, με αυτήν την τεχνική.

```
// πρόγραμμα: Μέση Τιμή 3
#include <iostream>
using namespace std;
int main()
{
   const double sentinel = 0;

   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  n = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;            // x == tn
      sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
}
```

Να παρατηρήσουμε ακόμη τα εξής:

1. Μπορεί να αναρωτιέσαι αν μπορούμε να γράψουμε το πρόγραμμα βάζοντας τις εντολές:

```
cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
```

Πλαίσιο 6.3

Γενικό Σχήμα Επανάληψης με Τιμή-Φρουρό

```
Ετοίμασε ή διάβασε την πρώτη τιμή v
while ( η v δεν είναι φρουρός )
{
   Επεξεργάσου τη v;
   Ετοίμασε ή διάβασε την επόμενη τιμή v
} // while
```

μόνο μια φορά. Ναι, μπορούμε! Δοκίμασε να το κάνεις.

2. Το πρόγραμμα μπορεί να γίνει πιο ευέλικτο αν ο *sentinel* δεν είναι σταθερά αλλά *μεταβλητή που η τιμή της θα δίνεται από το πληκτρολόγιο πριν από τα στοιχεία εισόδου*. Τροποποίησε το πρόγραμμα προς αυτήν την κατεύθυνση.

3. Μπορείς, αντί για μια τιμή-φρουρό, να έχεις μια ολόκληρη περιοχή τιμών. Π.χ. αν στο παράδειγμά μας ξέραμε ότι οι τιμές μας είναι *θετικές* θα μπορούσαμε να γράψουμε το πρόγραμμά μας ως εξής:

```
cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
while ( x > 0 )
{
   n = n + 1;            // x == tn
   sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
   cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
} //  while
```

4. Οι τιμές που ελέγχεις με φρουρό δεν είναι απαραίτητο να εισάγονται από το πληκτρολόγιο. Μπορεί να δημιουργούνται από το πρόγραμμα. Μπορεί, για παράδειγμα, να είναι όροι ακολουθίας που ξέρουμε το μαθηματικό της τύπο.

5. Δεν θα μπορούσαμε, αντί να ψάχνουμε για φρουρούς, να δηλώσουμε:

```
char resp;
```

να βάλουμε μια ερώτηση:

```
    cout << " Έχεις άλλες τιμές; (N/O): ");  cin >> resp;
```

και να ελέγχουμε την επανάληψη ως εξής:

```
while ( resp == 'N' )  { . . .
```

Και βέβαια θα μπορούσαμε[1]. (Πρόσεξε ότι στην περίπτωση αυτή δεν φτάνει ο έλεγχος για το `'N'`, αλλά πρέπει να ελέγχεις για ελληνικά, λατινικά, πεζά, κεφαλαία.)

Πάντως η τεχνική της τιμής-φρουρού (και η φιλοσοφία της) είναι πολύ χρήσιμη σε πολλές περιπτώσεις. Το γενικό σχήμα χρήσης το βλέπεις στο Πλ. 6.3.

6.1.2 Επιλεκτική Επεξεργασία

Το πρόβλημα που λύσαμε με τα τρία προγράμματα Μέση Τιμή μπορεί να το συναντήσεις αρκετά συχνά με δεδομένα διαφόρων ειδών, π.χ. μετρήσεις από κάποιο πείραμα, κάποια έρευνα αγοράς, κάποια σφυγμομέτρηση κλπ. Πολύ συχνά όμως, πέρα από τη γενική επεξεργασία που κάνουμε σε όλα τα στοιχεία, θέλουμε να κάνουμε ειδική επεξεργασία σε ένα υποσύνολο τιμών, που επιλέγονται με κάποια κριτήρια. Π.χ.

- Πόσες φορές μέτρησα στο πείραμά μου συχνότητα από 1000 *Hz* μέχρι 1500 *Hz*; Ποιά ήταν η μέση τιμή τους;
- Στη σφυγμομέτρησή μου, πόσοι από αυτούς που στις εκλογές ψήφισαν το κόμμα x, έχουν τελειώσει Πανεπιστήμιο;

[1] Μήπως στην περίπτωση αυτή κάθε τιμή ≠ `'N'` είναι φρουρός στην *resp*; Μήπως;...

- Στην έρευνα αγοράς που έκανα, πόσοι από αυτούς που έχουν πλυντήριο πιάτων κάνουν τα ψώνια τους σε μηνιαία βάση;

Στις περιπτώσεις αυτές το γενικό σχήμα επεξεργασίας είναι το εξής:

```
Κάνε τη γενική επεξεργασία στην τιμή x;
if (για τη x πληρούνται τα κριτήρια επιλογής)
   κάνε την ειδική επεξεργασία στην τιμή x
```

Ας κάνουμε κι εμείς κάτι παρόμοιο: θα τροποποιήσουμε το πρόγραμμα Μέση Τιμή 3 ώστε να μας δίνει επί πλέον:

- το πλήθος,
- το άθροισμα και
- τη μέση τιμή

όσων από τις τιμές που διαβάζει είναι θετικές και μέχρι (και) 10.

Η γενική επεξεργασία είναι αυτή που κάναμε και πιο πριν. Για την ειδική επεξεργασία χρειαζόμαστε: ένα μετρητή (*selN*) για το πλήθος των επιλεγόμενων τιμών, μια μεταβλητή (*selSum*) όπου θα «μαζεύουμε» το άθροισμα των επιλεγόμενων τιμών και μια μεταβλητή (*selAvrg*) για τη μέση τιμή τους. Η επιλογή και η ειδική επεξεργασία γίνεται με την εντολή:

```
      if ( 0 < x && x <= 10 )      // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;      // Αυτά γίνονται μόνο για
         selN = selN + 1;          // τους επιλεγόμενους αριθμούς
      } // if
```

Ολόκληρο το πρόγραμμα Μέση Τιμή 4:

```
// πρόγραμμα: Μέση Τιμή 4
#include <iostream>
using namespace std;
int main()
{
   const double sentinel( 0 );

   int n;          // Μετρητής όλων των στοιχείων
   int selN;       // Μετρητής επιλεγόμενων στοιχείων
   double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;     // Το άθροισμα όλων των στοιχείων
   double selSum;  // Άθροισμα επιλεγόμενων στοιχείων
   double avrg;    // Μέση Αριθμητική Τιμή όλων των στοιχείων
   double selAvrg; // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων

   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
   cout << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
   if ( selN > 0 )
   {
```

```
      selAvrg = selSum/selN;
      cout << " ΑΘΡΟΙΣΜΑ = " << selSum
           << "  <x> = " << selAvrg << endl;
   }
}
```

Εδώ βλέπεις μια **if** μέσα σε μια **while**.

6.2 * Αναλλοίωτες και Τερματισμός

Αντιγράφουμε από το παράδειγμα της §6.1:

```
   sum = 0;  m = 1;
   while ( m <= n )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;          // x = tm
      sum = sum + x;            // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;           // Ετοιμαζόμαστε για τον επόμενο αριθμό
   } // while
```

Είναι σωστό αυτό το πρόγραμμα; Δηλαδή: αν διαβαστούν n (> 0) τιμές t_1 ... t_n, θα ισχύει, μετά την εκτέλεση της **while**, η *sum* == $\sum_{j=1}^{n} t_j$;

Η τιμή του μετρητή m ξεκινάει από 1 και αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή επανάληψης. Άρα την πρώτη φορά που δεν θα ισχύει η `m <= n` –και θα σταματήσει η εκτέλεση της **while**– θα έχουμε m == n + 1. Αν αποδείξουμε ότι τότε θα ισχύει η *I*: *sum* == $\sum_{j=1}^{m-1} t_j$ θα έχουμε αποδείξει αυτό που θέλαμε.

Θα το αποδείξουμε με τη **μέθοδο της μαθηματικής επαγωγής**[2].

Βασικό βήμα: Αρχικώς, πριν αρχίσει η εκτέλεση της **while**, έχουμε, από τις εντολές προετοιμασίας: (*sum* == 0) && (m == 1). «Ταιριάζει» αυτή με την *sum* == $\sum_{j=1}^{m-1} t_j$; Στην *sum* έχουμε το άθροισμα των m – 1 πρώτων μελών του συνόλου { t_1 ... t_N } που έχουν ήδη πληκτρολογηθεί. Αφού αρχικώς δεν έχει πληκτρολογηθεί κάποια τιμή, η *sum* πρέπει να έχει τιμή 0. Αυτό είναι και το νόημα του $\sum_{j=1}^{1-1} t_j = \sum_{j=1}^{0} t_j$. Μπορούμε λοιπόν να πούμε ότι αρχικώς, πριν από τη **while**, ισχύει η *I*.

Επαγωγικό βήμα: Θα πρέπει να αποδείξουμε ότι αν η *I* ισχύει για κάποιο m θα ισχύει και για την επόμενη τιμή της m (δηλαδή: m+1). Στο πρόγραμμά μας όμως πηγαίνουμε στην επόμενη τιμή της m αφού διαβάσουμε, προσθέσουμε και μετρήσουμε την επόμενη τιμή. Θα πρέπει λοιπόν να αποδείξουμε ότι: *αν η I:* (*sum* == $\sum_{j=1}^{m-1} t_j$) *ισχύει όταν αρχίσουν να εκτελούνται οι επαναλαμβανόμενες εντολές θα ισχύει και μετά το τέλος της εκτέλεσής τους*, δηλαδή:

```
// sum == Σ(j:1..m-1)tj
   cin >> x;
   sum = sum + x;
   m = m + 1;
// sum == Σ(j:1..m-1)tj
```

[2] Δες για παράδειγμα Σ. Ανδρεαδάκη κ.ά. «ΑΛΓΕΒΡΑ Β′ ΛΥΚΕΙΟΥ», ΟΕΔΒ 1994.

Αυτό ξέρουμε να το αποδείξουμε: Για να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$ μετά τη `m = m + 1` θα πρέπει πριν από αυτήν να ισχύει η $sum == \sum_{j=1}^{(m+1)-1} t_j$ ή αλλιώς: $sum == \sum_{j=1}^{m} t_j$. Έχουμε δηλαδή:

```
   cin >> x;
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Για να ισχύει η $sum == \sum_{j=1}^{m} t_j$ μετά την `sum = sum + x` θα πρέπει πριν από αυτήν να ισχύει η: $sum + x == \sum_{j=1}^{m} t_j$:

```
   cin >> x;
// sum + x == Σ(j:1..m)tⱼ == Σ(j:1..m-1)tⱼ + tₘ }
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ)
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Αφού, όπως είπαμε, όταν η `cin >> x` εκτελείται για m-οστή φορά διαβάζεται η t_m, μπορούμε να πούμε ότι μετά την εκτέλεσή της θα έχουμε: $x == t_m$. Μαζί με το ότι $\sum_{j=1}^{m} t_j == \sum_{j=1}^{m-1} t_j + t_m$ καταλήγουμε στο ότι πριν από τη "`cin >> x`" θα πρέπει να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$.

Αυτή όμως ισχύει, αφού είναι η προϋπόθεσή μας.

Τι σημαίνουν τα παραπάνω με απλά λόγια;

- Η I ισχύει αρχικώς (βασικό βήμα), για $m == 1$.
- Αφού αποδείξαμε (επαγωγικό βήμα) ότι αν η I ισχύει αρχικώς[3] και εκτελεσθούν οι επαναλαμβανόμενες εντολές, που αυξάνουν την τιμή της m, η I ισχύει και πάλι, η I ισχύει και για $m == 2$.
- Αφού η I ισχύει για $m == 2$, λόγω του επαγωγικού βήματος, θα ισχύει και για $m == 3$

κ.ο.κ. Γενικώς, *η I ισχύει για οποιαδήποτε τιμή πάρει η m με τις επαναλαμβανόμενες εντολές*. Βλέπουμε λοιπόν ότι η εκτέλεση των επαναλαμβανόμενων εντολών αφήνει την I αναλλοίωτη· γι' αυτό και ονομάζεται **αναλλοίωτη της επανάληψης** (repetition invariant).

Παρατηρήσεις ▶

1. Για να αποδείξουμε την ορθότητα μιας **while** θα *πρέπει να έχουμε την αναλλοίωτη*. Πού θα τη βρούμε; Στα προγράμματα που γράφουμε εμείς δεν είναι δύσκολο να τη έχουμε: είναι, στην πραγματικότητα, η λογική περιγραφή της μεθόδου που χρησιμοποιούμε για να λύσουμε το πρόβλημά μας. Αν μας δώσουν μια **while** και μας ζητήσουν να βρούμε την αναλλοίωτη τα πράγματα είναι δύσκολα! Αν όμως υπάρχουν προδιαγραφές τα πράγματα είναι καλύτερα. Διάβασε όμως παρακάτω...
2. Όταν κάνουμε αποδείξεις από το τέλος προς την αρχή φτάνουμε σε κάτι σαν:

```
   while ( S )  E;
// Q
```

Εδώ τι κάνουμε; Αν μπορέσουμε να φέρουμε την Q στη μορφή (`!`S) `&&` Q_I, προσπαθούμε να αποδείξουμε ότι η Q_I είναι αναλλοίωτη της **while**. Φυσικά, η Q_I θα πρέπει να ισχύει και πριν

[3] Οι εντολές "`sum = 0; m = 1;`", που δίνονται πριν από την εντολή **while**, σκοπό έχουν να εξασφαλίσουν ότι ισχύει η I πριν από τη **while**.

από τη **while**. Αυτή η «συνταγή» μπορεί να μη φαίνεται και τόσο εύκολη για ορισμένες περιπτώσεις, όπως π.χ. η μετρούμενη επανάληψη· θα τα πούμε παρακάτω.
3. Πρόσεξε ακόμη το εξής: αν έχουμε **while** (*S*) *E*, με αναλλοίωτη *I*, οι *E* θα εκτελεσθούν μόνον αν ισχύει η *S*. Δηλαδή, αυτό που έχεις να αποδείξεις είναι:

```
// I && S
   E
// I  ◀
```

Τώρα, μπορούμε να διατυπώσουμε και συμβολικώς τον συμπερασματικό κανόνα της **while**:

$$\frac{I \,\&\&\, S\,\{E\}\,I}{I\,\{\textbf{while}(S)\;E\}\,(!S)\,\&\&\,I}$$

Και όταν κάνουμε αποδείξεις από το τέλος προς την αρχή, ποια συνθήκη θα πρέπει να ισχύει πριν από τη **while**; Ποια άλλη; Η *αναλλοίωτη*!

Όταν έχουμε επαναληπτικές εντολές, για να αποδείξουμε ότι το πρόγραμμά μας είναι ***ολικώς*** σωστό, *θα πρέπει να αποδείξουμε ότι η εκτέλεση όλων των επαναληπτικών εντολών θα τερματισθεί*. Πώς γίνεται αυτό; Ας έρθουμε στο παράδειγμά μας: Θεωρούμε την ακολουθία των διαδοχικών τιμών της διαφοράς $n - m$ που είναι ακέραιος αριθμός.

- Αφού η **while** εκτελείται μόνο όταν $m \le n$ έχουμε πάντοτε $n - m \ge 0$, δηλαδή έχουμε μια ακολουθία *φυσικών* αριθμών.
- Η ακολουθία αυτή είναι γνησίως φθίνουσα, αφού κάθε τιμή της *m* είναι κατά 1 μεγαλύτερη από την προηγούμενη.

Τα μαθηματικά μας λένε ότι *δεν είναι δυνατόν να έχουμε γνησίως φθίνουσα ακολουθία*[4] *φυσικών αριθμών με άπειρο πλήθος όρων*. Επειδή δεν είναι δυνατόν η ακολουθία που δημιουργεί η **while** να παραβιάσει αυτό το θεώρημα, η εκτέλεσή της θα σταματήσει κάποτε.

Αυτός είναι ο τρόπος που συνήθως χρησιμοποιούμε για να αποδείξουμε τον τερματισμό. Μερικές φορές δεν είναι και τόσο εύκολο να βρούμε τη γνησίως φθίνουσα ακολουθία φυσικών αριθμών.

6.2.1 * Παραδείγματα

Παράδειγμα 1 ✍

Ας υποθέσουμε ότι η C++ δεν μας δίνει την ακέραιη διαίρεση και την "**%**". Θέλουμε ένα πρόγραμμα που θα διαβάζει δύο ακέραιους *d1*, *d2* –ο πρώτος (διαιρετέος) μη αρνητικός, ο δεύτερος (διαιρέτης) θετικός– και θα υπολογίζει και θα τυπώνει το πηλίκο και το υπόλοιπο της ακέραιης διαίρεσης *d1*:*d2*.

Πρώτα οι προδιαγραφές: Για το υπόλοιπο *y*, σίγουρα θυμάσαι ότι, θα πρέπει να έχουμε $0 \le y < d2$. Για το πηλίκο *p* τι θα έχουμε; Τι άλλο από αυτό που σου μάθανε ως δοκιμή της διαίρεσης: $d1 == p \cdot d2 + y$:
Προϋπόθεση: $(d1 \ge 0)$ **&&** $(d2 > 0)$
Απαίτηση: $(0 \le y < d2)$ **&&** $(d1 == p \cdot d2 + y)$

Ο αλγόριθμος θα πρέπει να σου είναι γνωστός από το Δημοτικό Σχολείο (από τότε που σου λέγανε ότι η διαίρεση είναι πολλές αφαιρέσεις): όσο ο *d2* είναι μικρότερος από ή ίσος με *d1* (ο *d2* χωράει στον *d1*) αφαιρούμε από τον *d1* τον *d2* και μετρούμε την αφαίρεση. Τελικώς το πλήθος των αφαιρέσεων θα είναι το πηλίκο ενώ ότι έχει μείνει από τις διαδοχικές αφαιρέσεις είναι το υπόλοιπο. Δηλαδή:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
```

[4] Όπου λέμε «ακολουθία» εννοούμε ακολουθία με *πεπερασμένο* πλήθος όρων.

```
   while ( d2 <= y )
   {
      y = y - d2;
      p = p + 1;
   } // while
// (0 ≤ y < d2) && (d1 == p*d2 + y)
```

Ας αποδείξουμε τώρα, από το τέλος προς την αρχή, την ορθότητα του προγράμματός μας.

Όπως είπαμε, όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!`*S* `&&` *I*. Στην περίπτωσή μας

- `!`*S* είναι η `!(d2 <= y)`, ή αλλιώς: $d2 > y$.
- *I* θα πρέπει να είναι αναλλοίωτη της επανάληψης.

Πώς θα βρούμε την αναλλοίωτη; Παρατηρούμε το εξής: Η απαίτηση που έχουμε για μετά τη **while** είναι: $(0 \le y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$ ή αλλιώς: $(0 \le y)$ `&&` $(y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αν από αυτήν ξεχωρίσουμε την $y < d2$, που είναι η `!`*S*, το υπόλοιπο θα πρέπει να είναι η αναλλοίωτη. Θα πρέπει λοιπόν να αποδείξουμε ότι η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ είναι η αναλλοίωτη της **while**.

1. *Βασικό βήμα*: Η αναλλοίωτη θα πρέπει να ισχύει πριν από τη **while**. Θα πρέπει δηλαδή να έχουμε:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ μετά την `p = 0` θα πρέπει να έχουμε πριν από αυτήν $(0 \le y)$ `&&` $(d1 == 0{\cdot}d2 + y)$ ή αλλιώς:

$$(0 \le y) \text{ \&\& } (d1 == y)$$

```
   y = d1;
// (0 ≤ y) && (d1 == y) }
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == y)$ μετά την `y = d1` θα πρέπει να ισχύει πριν από αυτήν η: $(0 \le d1)$ `&&` $(d1 == d1)$ ή απλούστερα: $d1 \ge 0$ που συνάγεται από την προϋπόθεση.

2. *Επαγωγικό βήμα*: Θα αποδείξουμε ότι:

```
   // (d2 ≤ y) && (0 ≤ y) && (d1 == p*d2 + y)
      y = y - d2;
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να έχουμε μετά την `p = p + 1` την $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$:

```
      y = y - d2;
   // (0 ≤ y) && (d1 == (p+1)*d2 + y)
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$ μετά την `y = y - d2` θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y - d2)$ `&&` $(d1 == (p+1){\cdot}d2 + y - d2)$ ή αλλιώς: $(d2 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αυτή όμως συνάγεται από την $(d2 \le y)$ `&&` $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$.

Άρα το πρόγραμμά μας είναι *μερικώς σωστό*.

Να δούμε τώρα αν είναι και *ολικώς σωστό*, αν δηλαδή τερματίζεται η εκτέλεσή του. Ας θεωρήσουμε την ακολουθία των διαδοχικών τιμών της *y*. Αυτές είναι:

- ακέραιες,
- μη αρνητικές, αφού η `y = y - d2` εκτελείται μόνον αν $d2 \le y$ και
- κάθε μια είναι *μικρότερη από την προηγούμενή της* αφού έχουμε από την προϋπόθεσή μας ότι $d2 > 0$.

Έχουμε δηλαδή μια γνησίως φθίνουσα ακολουθία φυσικών αριθμών. Αυτή αποκλείεται να έχει άπειρο πλήθος όρων· άρα η εκτέλεση της **while** θα τερματισθεί κάποτε. Δηλαδή το πρόγραμμά μας *είναι ολικώς σωστό*. Καμάρωσέ το:

```
#include <iostream>
using namespace std;
int main()
{
   int d1, d2, p, y;

   cin >> d1 >> d2;
   if ( d1 >= 0 && d2 > 0 )
   { // (d1 ≥ 0) && (d2 > 0)
      y = d1;
      p = 0;
      while ( d2 <= y )  // I: (0 ≤ y) && (d1 == p*d2 + y)
      {
         y = y - d2;
         p = p + 1;
      } // while
   // (0 ≤ y < d2) && (d1 == p*d2 + y)
      cout << " Πηλίκο: " << p << "    Υπόλοιπο: " << y << endl;
      cout << " Η C++ δίνει: " << endl;
      cout << " Πηλίκο: " << (d1 / d2)
           << "    Υπόλοιπο: " << (d1 % d2) << endl;
   }
   else
   // false
      cout << " Λάθος! " << endl;
}
```

Παρατήρηση ▶

Τι θα πει διαίρεση δια 0 (μηδέν); Θα πει ότι η ακολουθία μας *δεν είναι γνησίως φθίνουσα* οπότε η απόδειξη του τερματισμού δεν γίνεται. Ούτε και τερματισμός της εκτέλεσης της **while** γίνεται! Να λοιπόν που μια από τις «κακοτοπιές» που προσέχουμε (διαίρεση δια 0) έχει σχέση με μια **while** που δεν τελειώνει ποτέ[5]! Έτσι, από περιέργεια, αξίζει τον κόπο να βγάλεις την **if**, που ελέγχει την προϋπόθεση, και να δώσεις $d2 = 0$. ◀

Παράδειγμα 2

Ας πούμε τώρα ότι η C++ *δεν* μας δίνει συνάρτηση για την τετραγωνική ρίζα θετικού αριθμού. Ας γράψουμε ένα πρόγραμμα που να την υπολογίζει. Πώς; Τα μαθηματικά[6] μας λένε ότι: αν $x_0 > 0$ και $x_0^2 > a$ τότε η ακολουθία:

$$\{ n: \mathbb{N}^+, x_n: \mathbb{R} \bullet x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}} \right) \}$$

συγκλίνει προς τη $\sqrt{a}$.[7]

Τι προδιαγραφές θα έχουμε; Ή αλλιώς: πόσο καλή προσέγγιση θα έχουμε; Ας πούμε ότι θέλουμε το σχετικό σφάλμα να είναι μικρότερο από κάποιο $\varepsilon > 0$:

$$\frac{\left| x - \sqrt{a} \right|}{\sqrt{a}} < \varepsilon \qquad (1)$$

---

[5] Φυσικά, ο υπολογιστής σου προσέχει ότι ο διαιρέτης είναι 0 πριν κάνει τη διαίρεση και διακόπτει την εκτέλεση του προγράμματος με το σχετικό μήνυμα λάθους.

[6] Η ακολουθία προκύπτει όταν προσπαθούμε να βρούμε μια προσέγγιση της λύσης της εξίσωσης $x^2 - A = 0$ με τη μέθοδο Newton - Raphson.

[7] Δες την απόδειξη στο `http://en.wikipedia.org/wiki/ Methods_of_computing_square_roots` ή στο (Κάππος 1962), ας πούμε.

Μπορούμε εύκολα να γράψουμε μια **while** που να υπολογίζει τους όρους αυτής της ακολουθίας:

```
x = αρχική τιμή;
while ( S )
   x = 0.5*(x + a/x);
```

Τι θα βάλουμε ως αρχική τιμή *x*; Κάποια τιμή που να μας δίνει: $(x > 0)$ **&&** $(x^2 > a)$. Να μια σκέψη:

```
if (a <= 1)  x = a + 1;  else  x = a;
```

αλλά θα πρέπει να αποδείξουμε ότι είναι σωστή:

```
// a > 0
   if (a <= 1)  x = a + 1;  else  x = a;
// (x > 0) && (x² > a) }
```

Για να ισχύει αυτό, αρκεί να ισχύουν οι:

```
// (a > 0) && (a <= 1)      // (a > 0) && (a > 1)
   x = a + 1;          και      x = a;
// (x > 0) && (x² > a)      // (x > 0) && (x² > a)
```

Η απόδειξη των παραπάνω είναι τετριμμένη.

Η *S* ποια θα είναι; Προφανώς η άρνηση της (1), δηλαδή η:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} \geq \varepsilon$$

ή:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} = \frac{\left|x-\sqrt{a}\right|\left|x+\sqrt{a}\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} = \frac{\left|x^2-a\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} \approx \frac{\left|x^2-a\right|}{2a} \geq \varepsilon$$

(στο τελευταίο βήμα χρησιμοποιήσαμε την προσέγγιση: $x \approx \sqrt{a}$).

Τι θα πάρουμε ως ε; Για να σκεφτούμε λίγο παραπάνω την τελευταία σχέση που γράφεται:

$$\left|\frac{x^2}{a}-1\right| \geq 2\varepsilon$$

Το ιδανικό θα ήταν να μηδενίσουμε το αριστερό μέρος. Αλλά ξέρουμε ότι το αριστερό μέρος θα είναι μηδέν αν η διαφορά του x^2/a από το 1 είναι < ε`double` (**DBL_EPSILON**). Επομένως, το καλύτερο που μπορούμε να πάρουμε είναι όταν ε = ½**DBL_EPSILON**. Μπορούμε λοιπόν να γράψουμε:

```
if ( a <= 1 )  x = a + 1;  else  x = a;
while ( fabs(x*x - a ) >= DBL_EPSILON*a)
   x = 0.5*(x + a/x);
```

Δεν θα ψάξουμε να βρούμε την αναλλοίωτη; *Δεν μας χρειάζεται!* Όλοι οι υπολογισμοί αποβλέπουν στο να ανατρέψουμε τη συνθήκη της **while**.

Είναι σίγουρος ο τερματισμός αυτής της **while**; Αφού τα μαθηματικά μας εγγυώνται ότι η ακολουθία των διαδοχικών τιμών της *x* συγκλίνει στην $\sqrt{a}$ έχουμε ότι για κάθε $\delta > 0$ υπάρχει *N* τέτοιο ώστε για κάθε $n > N$ να έχουμε $|x_n - \sqrt{a}| < \delta$. Αν λοιπόν πάρουμε $\delta = \varepsilon\sqrt{a}$, τοτε από κάποιο *n* και μετά θα έχουμε: $|x_n - \sqrt{a}| < \varepsilon\sqrt{a}$, που όπως είδαμε παραπάνω ισοδυναμεί με: $|x^2 - a| < 2\varepsilon a$. Αυτή όμως είναι η **!**$(|x^2 - a| \geq 2\varepsilon a)$, δηλαδή η άρνηση της συνθήκης της **while**.

Αλλά, προσοχή: Η Ανάλυση μας εγγυάται τη σύγκλιση με την προϋπόθεση ότι $a > 0$. Αν $a < 0$ –μπορείς να το δεις με μερικές δοκιμές– η **while** (και το πρόγραμμα) δεν τελειώνει ποτέ! Να λοιπόν άλλη μια περίπτωση (μετά τη διαίρεση δια μηδέν) όπου αυτό που λέγαμε «κακοτοπιά» έχει να κάνει με τον τερματισμό μιας **while**. Η C++ προσεγγίζει την τετραγωνική ρίζα (*sqrt*) με τον τρόπο που είδαμε παραπάνω, αλλά αν δώσεις αρνητικό όρισμα

στην *sqrt*, αντί να κολλήσει σε μια αέναη επανάληψη, προτιμάει να κόψει την εκτέλεση του προγράμματος.

Εδώ είδαμε παράδειγμα προγράμματος στο οποίο τα μαθηματικά μας εγγυώνται και τη μερική και την ολική ορθότητα. Και όλα αυτά που γράφουμε τι είναι; *Προσεκτική υλοποίηση αυτών που μας λένε τα μαθηματικά!*

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
#include <cmath>
using namespace std;
int main()
{
   double a, x;

   cin >> a;
   if ( a > 0 )
   {// a > 0
      if ( a <= 1 )  x = a + 1;  else  x = a;
      while ( fabs(x*x - a) >= DBL_EPSILON*a )
         x = 0.5*(x + a/x);
   // |x² - a| < εa
      cout.precision(16);
      cout << x << "  " << sqrt(a) << endl;
   }
   else
      cout << " μόνον θετικούς παρακαλώ" << endl;
}
```

Παράδειγμα 3

Πρόβλημα: Θέλουμε ένα πρόγραμμα που θα διαβάσει από το πληκτρολόγιο n (> 0) τιμές t_k, k = 1, 2,...n –ας πούμε τύπου `int`– και στο τέλος θα μας δώσει τη μεγαλύτερη από αυτές και τη σειρά (*kmax*) με την οποία πληκτρολογήθηκε. Πριν από τους αριθμούς θα διαβάζει την τιμή της n.

Σε μια θέση της μνήμης, *tmax*, κρατούμε τη μέγιστη τιμή από όσες έχουμε διαβάσει και σε μια άλλη, *kmax*, τη σειρά της. Να οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$

Απαίτηση: $1 \le kmax \le n$ && $tmax == t_{kmax}$ && $\forall j: 1..n \bullet t_j \le tmax$

Ας δούμε πώς μπορούμε να λύσουμε το πρόβλημά μας.

Διαβάζουμε την πρώτη τιμή, που προφανώς είναι και η μέγιστη μέχρι στιγμής:

```
   cin >> t;  // t == t1 }
   k = 1;  tmax = t;  kmax = k;
```

Τώρα διαβάζουμε τη δεύτερη τιμή. Αν είναι μεγαλύτερη από την πρώτη, που τη θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη δεύτερη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη την πρώτη:

```
   cin >> t;   // t == t2
   k = k + 1;  // == 2
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Στη συνέχεια διαβάζουμε την τρίτη τιμή. Αν είναι μεγαλύτερη από αυτήν που θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη τρίτη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη αυτήν που είχαμε μέχρι τώρα:

```
   cin >> t;   // t == t3
   k = k + 1;  // == 3
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Όπως βλέπεις, διαχειριζόμαστε τη 2η και την 3η τιμή με τον ίδιο ακριβώς τρόπο και με τον ίδιο τρόπο θα διαχειριστούμε και τις υπόλοιπες. Στην 1η έχουμε διαφορά. Μπορούμε

λοιπόν να ξεκινήσουμε, όπως είδαμε, με την πρώτη τιμή και να βάλουμε τα υπόλοιπα σε μια **while**:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if (t > tmax)  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

Όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!(k <= n)`, δηλαδή η $k > n$. Αφού όμως η k αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή της επανάληψης και αφού $n > 0$ θα έχουμε ακριβέστερα: $k == n + 1$. Από αυτό και από την απαίτηση προσπαθούμε να μαντέψουμε την αναλλοίωτη της επανάληψης: βάζουμε στην απαίτηση όπου n το k–1 και παίρνουμε:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j\text{: } 1..\ k\text{–}1 \bullet t_j \le tmax$$

Αν αποδείξουμε ότι αυτή είναι η αναλλοίωτη *I*, τότε σίγουρα μετά τη **while** θα έχουμε την απαίτηση. Η απόδειξη δεν είναι δύσκολη αλλά είναι αρκετά μακροσκελής και την αφήνουμε για άσκηση (6-16).

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  t, tmax;
   int  n, k, kmax;

   cin >> n;
   if ( n > 0 )
   {// n > 0
      cin >> t;     // t == t1
      k = 1;
      tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
   // 1 ≤ kmax ≤ n && tmax == tkmax && ∀j:1..n • tj ≤ tmax
      cout << " Μέγιστη τιμή: " << tmax
           << "  (" << kmax << "η) " << endl;
   }
   else
      cout << " το πλήθος πρέπει να είναι θετικό" << endl;
}
```

Στο Πλ. 6.4 βλέπεις το γενικό σχήμα προγράμματος για τον υπολογισμό μεγίστου. Για να υπολογίσεις το ελάχιστο άλλαξε τη συνθήκη στην **if**:

```
if ( t < tmin ) ...
```

Μερικές φορές, όπως θα δεις στη συνέχεια, δεν είναι εύκολο να επεξεργαστείς χωριστά την πρώτη τιμή. Γράφουμε λοιπόν τη **while** έτσι ώστε να επεξεργάζεται όλες τις τιμές (k από 1 μέχρι n). Τώρα όμως υπάρχει το εξής πρόβλημα: την πρώτη φορά που θα εκτελεσθεί η **if** ($t > tmax$) τι τιμή θα έχει η *tmax*;

Η *tmax* θα πρέπει να έχει τέτοια τιμή ώστε:

- να αλλάξει οπωσδήποτε –όποια και αν είναι η πρώτη τιμή της t– και έτσι να πάρει μια ρεαλιστική τιμή,
- να μην υπάρχει πιθανότητα να θεωρηθεί ως μια από τις τιμές που επεξεργαζόμαστε.

Δηλαδή, πρέπει να βάλουμε ως αρχική τιμή της *tmax* μια *απίθανα μικρή τιμή*, π.χ. το $-\infty$!

«Πλην άπειρο και πράσινα άλογα!» θα σκεφτείς, «Γίνονται τέτοια πράγματα;» Δεν γίνονται βέβαια, αλλά για κάθε πρόβλημα που έχεις να λύσεις, είναι πολύ πιθανό να μπορείς να βρεις μια απίθανα μικρή τιμή. Π.χ.

- αν οι τιμές που μελετάς είναι θετικές, μια απίθανα μικρή τιμή είναι το 0 ή το -1 (ή οποιοσδήποτε αρνητικός),
- αν ξέρεις ότι οι τιμές που επεξεργάζεσαι είναι μεταξύ 150 και 250, μια απίθανα μικρή τιμή μπορεί να είναι το 0 ή το 10 κλπ.

Στο παράδειγμα που δώσαμε παραπάνω, αν ξέραμε επιπλέον ότι *όλες οι τιμές που διαβάζουμε είναι θετικές*, θα μπορούσαμε να γράψουμε:

```
tmax = -1;
k = 1;
while ( k <= n )
{
   cin >> t;  // t = tk
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Φυσικά, αν ψάχνεις για ελάχιστη τιμή θα ξεκινήσεις από μια *απίθανα μεγάλη τιμή* ($+\infty$).

Πλαίσιο 6.4

Εύρεση Μέγιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmax = t;  kmax = k;
k = k + 1;  // == 2
while ( συνθήκη )
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Εύρεση Ελάχιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmin = t;  kmin = k;
k = k + 1;  // == 2
while (συνθήκη)
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t < tmin )  {  tmin = t;  kmin = k;  }
   k = k + 1;
} // while
```

6.3 Η Μετρούμενη Επανάληψη

Η **μετρούμενη επανάληψη** είναι πολύ συνηθισμένη και απλή. Στην §6.1 είδαμε την απλούστερη μορφή της:

```
m = 1;
while ( m <= n )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m + 1;  }
```

Από όσα μάθαμε μέχρι τώρα, αν οι «άλλες επαναλαμβανόμενες εντολές» δεν αλλάζουν την τιμή της *m*, θα εκτελεσθούν:

- 0 φορές αν $n < 1$. Η τιμή της *m* δεν θα αλλάξει.
- *n* φορές αν $n \geq 1$. Στο τέλος η τιμή της *m* θα είναι $n + 1$.

Όταν οι «άλλες επαναλαμβανόμενες εντολές» εκτελούνται για *k*-οστή φορά έχουμε $m == k$.

Η *m* είναι η **μεταβλητή ελέγχου** (control variable) της επανάληψης.

Πολλοί προτιμούν μια άλλη μορφή για την ίδια δουλειά:

```
m = n;
while ( m >= 1 )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m - 1;    }
```

ή

```
m = n;
while (m > 0)  {  άλλες επαναλαμβανόμενες εντολές
                  m = m - 1;    }
```

διότι η ακολουθία τιμών της *m* είναι ακριβώς αυτή που χρησιμοποιήσαμε για να αποδείξουμε τον τερματισμό.

Ας δούμε όμως την εξής γενίκευση: αν η(οι) εντολή (-ές) *E* δεν μεταβάλλουν τις τιμές των *m*, *Ma*, *Mt*, *b* (οποιουδήποτε αριθμητικού τύπου) και αν $b > 0$, τότε κατά την εκτέλεση των

```
m = Ma;
while ( m <= Mt )
{
   E;
   m = m + b;
}
```

η(οι) εντολή(-ές) *E* θα εκτελεσθεί(-ούν) *n* φορές όπου:

- $n = 0$ αν $Ma > Mt$,
- $n = Trunc[(Mt - Ma + b)/b]$ φορές αν $Ma \leq Mt$.

Κατά την *k*-οστή φορά που θα εκτελεσθεί(-ούν) η(οι) *E*, η *m* θα έχει τιμή $Ma + (k-1)\cdot b$. Το *b* λέγεται **βήμα** (step).

Ας δούμε ένα

Παράδειγμα

Το παρακάτω πρόγραμμα μας δίνει, για τα πρώτα 15 sec, την απόσταση που διανύει, ανά 0.5 *sec*, ένα κινητό που ξεκινάει από την ηρεμία και κινείται με σταθερή επιτάχυνση 10 m/sec^2.

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf(ios::fixed,ios::floatfield);
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
```

```
   step = 0.5; // sec
   t = 0.0;
   while ( t <= 15.0 )
   {
      x = 0.5*a*t*t;
      cout.width(6);  cout.precision(1);  cout << t << "     ";
      cout.width(7);  cout.precision(2);  cout << x << endl;
      t = t + step;
   } // while
}
```

Το πρόγραμμα αυτό θα μας δώσει τα αποτελέσματα σε μορφή πίνακα:

```
t          x
   sec        m
   0.0        0.00
   0.5        1.25
   1.0        5.00
  . . .      . . .
  14.5     1051.25
  15.0     1125.00
```

6.4 Η Εντολή for

Η C++ έχει μια εντολή, τη **for**, που παραδοσιακά χρησιμοποιείται κυρίως για να γράφουμε μετρούμενες επαναλήψεις. Π.χ. η:

```
   m = Ma;
   while ( m <= Mt )
   {
      E;
      m = m + b;      // b > 0
   }
```

που είδαμε στην προηγούμενη παράγραφο μπορεί να γραφεί:

```
   for ( m = Ma;  m <= Mt;  m = m + b )
   {
      E;
   }
```

ενώ η

```
   m = Ma;
   while ( m >= Mt )
   {
      E;
      m = m - b;      // b > 0
   }
```

μπορεί να γραφεί:

```
   for ( m = Ma;  m >= Mt;  m = m - b )
   {
      E;
   }
```

Η μεταβλητή ελέγχου *m* πρέπει να είναι αριθμητικού ή απαριθμητού τύπου. Συνηθέστερα χρησιμοποιούμε ακέραιο ή απαριθμητό τύπο.

Τα *Ma, Mt, b* μπορεί γενικώς να είναι παραστάσεις. Παρ' όλο που δεν απαγορεύεται, απόφευγε όταν χρησιμοποιείς τη **for** να γράφεις επαναλαμβανόμενες εντολές (*E*) που να τροποποιούν οποιαδήποτε από τις *m, Ma, Mt, b*.

- ***Είναι καλό οι τιμές των Ma, Mt, b να καθορίζονται όταν αρχίζει η εκτέλεση της for και να μη μεταβάλλονται οι τιμές τους στη διάρκεια της εκτέλεσης της for. Η τιμή της μεταβλητής ελέγχου θα πρέπει να μεταβάλλεται μόνο από την*** `m = m ± b`.

Και γιατί όλα αυτά; Με τους παραπάνω περιορισμούς, τα δύο σχήματα επανάληψης με **for** που γράψαμε πιο πάνω δεν χρειάζονται απόδειξη για τον τερματισμό. Χωρίς αυτούς τους περιορισμούς η απόδειξη ορθότητας μπορεί να γίνει αρκετά πολύπλοκη. Γράφουμε λοιπόν μια **while** (και όχι **for**) για να ξέρουμε τι μας περιμένει.

Όπως η **while** έτσι και η **for**, περιμένει *μια επαναλαμβανόμενη εντολή*· αν θέλουμε να βάλουμε περισσότερες, όπως κάναμε στην **if** και στη **while**, τις «συσκευάζουμε» με ένα άγκιστρο (`{`) στην αρχή και ένα άγκιστρο (`}`) στο τέλος σε μια *σύνθετη εντολή*.

Δες πώς θα μπορούσε να γραφεί η επανάληψη του Μέση Τιμή 1 (ή 2) με χρήση της **for**:

```
   sum = 0;
   for ( m = 1;  m <= n;  m = m+1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;       // x = tN
      sum = sum + x;     // (x == tm) && (sum = Σ(j:1..m)tj)
   } // for
   avrg = sum / n;
```

Ας δούμε μερικά παραδείγματα ακόμη:

Παράδειγμα 1

Η μεταβλητή ελέγχου της **for** μπορεί να είναι και τύπου `bool`. Ξαναγράφουμε το πρόγραμμα της §4.3 που δημιουργεί τους αληθοπίνακες:

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;
   int   iP, iQ;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q " << endl;
   for ( iP = 0;  iP <= 1;  iP = iP+1 )
   {
      P = static_cast<bool>(iP);
      for (iQ = 0;  iQ <= 1;  iQ = iQ+1)
      {
         Q = static_cast<bool>(iQ);
         cout << P << "   " << Q << "   " << !P << "   "
              << (P && Q) << "    " << (P || Q) << "    "
              << (P <= Q) << "    " << (P == Q) << "    "
              << (P != Q) << endl;
      } // for (iQ...
   } // for (iP
}
```

Παράδειγμα 2

Ο πιο συνηθισμένος τύπος για τη μεταβλητή ελέγχου είναι ο `int`. Ας δούμε το εξής πρόβλημα: Να γραφεί πρόγραμμα που θα διαβάζει την τιμή του n και θα υπολογίζει και θα τυπώνει το άθροισμα:

$$1^1 + 2^2 + \ldots + n^n$$

Εδώ βλέπουμε ότι έχουμε (α) μετρούμενη επανάληψη (1.. n) και (β) υπολογισμό αθροίσματος. Σύμφωνα με όσα μάθαμε (Πλ. 6.2α) γράφουμε:

```
   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      υπολόγισε τον m-οστό όρο mm;
      sum = sum + mm;
   }
```

Αυτό το πρόγραμμα θα μοιάζει με αυτό της Μέσης Τιμής, με τη διαφορά ότι ο όρος που θα προσθέτουμε στο *sum* δεν προέρχεται από ανάγνωση από το πληκτρολόγιο αλλά από υπολογισμό που γίνεται μέσα στο πρόγραμμα.

Για τον υπολογισμό του m^m θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση **pow**. Αλλά εδώ θα κάνουμε τον υπολογισμό με πολλούς πολλαπλασιασμούς:

$$m^m = \underbrace{m \cdot \ldots \cdot m}_{m\varphi o\rho\varepsilon\varsigma}$$

Το πώς υπολογίζουμε ένα γινόμενο το είδαμε στο Πλ. 6.2β. Υπολογίζουμε λοιπόν το m^m ως εξής:

```
product = 1;  k = 1;          product = 1;
while ( k <= m )          ή  for ( k=1; k <= m; k=k+1 )
{
   product = product*m;          product = product*m;
   k = k + 1;
} // while (k ...
```

Να λοιπόν το πρόγραμμά μας:

```
#include <iostream>
using namespace std;
int main()
{
   int  n, sum;
   int  m, k, product;

   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      product = 1;
      for ( k = 1;  k <= m;  k = k + 1 )
      {
         product = product*m;
      } // for (k...
      sum = sum + product;
   } // for (m...
   cout << " n = " << n << "  Άθροισμα Σειράς = "
        << sum << endl;
} // main
```

Παράδειγμα 3

Ας έρθουμε τώρα στο πρόγραμμα της §6.4: Εδώ η μεταβλητή ελέγχου είναι τύπου **double**. Μπορούμε να γράψουμε την επανάληψη με μια **for** ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 8 );
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
   step = 0.5; // sec
   for ( t = 0.0;  t <= 15.0;  t += step )
   {
```

```
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
   } // for
}
```

Πάντως, όπως θα μάθεις αργότερα, είναι καλύτερο να σκεφτείς ότι οι επαναλαμβανόμενες εντολές εκτελούνται 31 φορές και να γράψεις:

```
   int    k;
     . . .
   t = 0.0;
   for ( k = 1;  t <= 31;  k++ )
   {
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
      t = t + step;
   } // for
```

Η **for** της C++ έχει πολύ περισσότερες δυνατότητες. Προς το παρόν είδαμε –και θα χρησιμοποιούμε– μόνον αυτές που μας χρειάζονται.

6.5 Λαθάκια και Σοβαρά Λάθη

Τα λάθη στις επαναληπτικές εντολές μπορεί να είναι πολύ πιο «δραματικά» και αυτό έχει να κάνει με τον τερματισμό. Το δίδαγμα από εδώ είναι:

- ***Σε κάθε εντολή while πρέπει να φροντίζεις ώστε κάποτε, κατά τη διάρκεια της εκτέλεσης, η συνθήκη της εντολής να γίνεται*** `false`**.**

Τα ίδια ισχύουν και για τη **for**, αλλά οι περιορισμοί που έχουμε βάλει στη χρήση της είναι πολύ ισχυρότεροι.

Ένας πολύ απλός τρόπος να παραβείς το παραπάνω δίδαγμα είναι να βάλεις ένα `";"` μετά την παρένθεση της συνθήκης της **while**:

```
while ( S );
```

Στην περίπτωση αυτή ζητάς την εκτέλεση της *κενής εντολής* όσο ισχύει η συνθήκη της **while**. Αν λοιπόν η συνθήκη ισχύει αρχικώς και αρχίσει η εκτέλεση της **while**, η κενή εντολή δεν μπορεί να την ανατρέψει. Αν δεν το πιστεύεις δοκίμασέ το. Αλλά να θυμάσαι:

- ***Δεν βάζουμε ποτέ "*** `;` ***" μετά την παρένθεση της συνθήκης της while.***

Όπως βλέπεις, ένα «τόσο δα λαθάκι» μπορεί να έχει σοβαρά επακόλουθα.

6.6 Τι (Πρέπει να) Έμαθες

Θα πρέπει να μπορείς να γράψεις προγράμματα στα οποία υπάρχει ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Θα πρέπει να μπορείς να γράψεις τέτοια προγράμματα είτε ξέρεις το πλήθος των επαναλήψεων πριν αρχίσει η εκτέλεσή τους (μετρούμενη επανάληψη) είτε θα τις συνεχίσεις μέχρι να ισχύσει κάποια συνθήκη (π.χ. να διαβαστεί η τιμή-φρουρός).

Θα πρέπει να έμαθες ακόμη να κάνεις μερικές πολύ συνηθισμένες επαναληπτικές δουλειές:

- υπολογισμό αθροίσματος ή γινομένου,

- υπολογισμό μεγίστου ή ελαχίστου

για τιμές που δημιουργεί το πρόγραμμα ή για τιμές που διαβάζει,

Τέλος, θα πρέπει να μπορείς να αποδείξεις την ορθότητα (απλών τουλάχιστον) προγραμμάτων με επαναλήψεις.

Ασκήσεις

Α Ομάδα

6-1 Γράψε το πρόγραμμα Μέση Τιμή 2 που περιγράψαμε στην §6.1. Θα διαφέρει από το Μέση Τιμή 1 στο ότι το *n* είναι μεταβλητή που η τιμή της διαβάζεται πριν από τις τιμές που θα αθροίσουμε.

6-2 Με βάση το πρόγραμμα για το άθροισμα, γράψε πρόγραμμα που θα υπολογίζει και θα εκτυπώνει το γινόμενο *n* πραγματικών αριθμών που δίνονται από το πληκτρολόγιο. Πριν από τις τιμές θα δίνεται η τιμή του *n*. Απόδειξε ότι το πρόγραμμά σου είναι ολικώς σωστό.

6-3 Ξαναδιάβασε το πρόγραμμα για το λογαριασμό της ΔΕΗ στην §6.4. Τροποποίησέ το έτσι ώστε να βγάζει πολλούς λογαριασμούς, διαβάζοντας από το πληκτρολόγιο τις καταναλώσεις. Το πρόγραμμα θα σταματάει αν του δώσουμε *αρνητική* κατανάλωση (κάθε αρνητική τιμή είναι τιμή - φρουρός).

Β Ομάδα

6-4 Τροποποίησε το πρόγραμμα Μέση Τιμή 3 της ώστε να έχει μια μόνον φορά την `cin >> x`. Σου αρέσει όπως έγινε;

6-5 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει την τιμή του *n*-οστού όρου της ακολουθίας, που καθορίζεται από τον τύπο:

α) $a_0 = 0,\ a_{k+1} = a_k k + k,$ για $k \geq 1$

β) $b_0 = b_1 = 0,\ b_k = b_{k-1} + b_{k-2} + k,$ για $k \geq 2$

6-6 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο πραγματικούς αριθμούς – θετικούς ή αρνητικούς– και θα σταματάει μόλις διαβάσει 10 αρνητικούς. Στο τέλος θα μας λέει:

- πόσους αριθμούς διάβασε συνολικώς και
- το άθροισμα των θετικών αριθμών που διάβασε.

6-7 Μετά την ανακοίνωση των αποτελεσμάτων στις εξετάσεις δύο μαθημάτων, έγινε φανερό ότι στο δεύτερο από αυτά έγινε «σφαγή». Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 40 δυάδες πραγματικών αριθμών που αντιπροσωπεύουν τους βαθμούς 40 σπουδαστών στα δύο μαθήματα. Στο τέλος θα πρέπει να μας λέει:

- πόσοι σπουδαστές είχαν στο δεύτερο μάθημα μεγαλύτερο βαθμό από ότι στο πρώτο,
- τους μέσους όρους των βαθμών στα δύο μαθήματα.

6-8 Απόδειξε ότι:

```
// n > 0
   k = 1;  sum = 0;
   while ( k <= n )    // I:  (sum = (k - 1)*k/2)
   {  sum = sum + k;  k = k + 1;  }
// sum = (n + 1)*n/2
```

Γ Ομάδα

6-9 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 150 πραγματικούς αριθμούς. Το πρόγραμμα θα πρέπει να υπολογίσει και στο τέλος να γράψει:

- το άθροισμα των θετικών,
- πόσοι είχαν απόλυτη τιμή μεγαλύτερη από 10,
- ποιος ήταν ο μέγιστος *αρνητικός* από τους αριθμούς που δόθηκαν (δηλαδή, αρνητικός με τη ελάχιστη απόλυτη τιμή).

6-10 Ξαναδιάβασε το πρόγραμμα της ελεύθερης πτώσης, της §3.4 και κάνε τις εξής παραλλαγές:

1. Το πρόγραμμα να τυπώνει ανά 1 sec, από τότε που αφέθηκε και μέχρι να κτυπήσει στο έδαφος
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - το ύψος που βρίσκεται το σώμα,
   - την ταχύτητα που έχει το σώμα.
2. Το πρόγραμμα να τυπώνει ανά 10 *m* διανυθέντος διαστήματος, από τότε που αφέθηκε (διάστημα 0) και μέχρι να κτυπήσει στο έδαφος (διάστημα *h*)
   - το διάστημα που διανύθηκε από την αρχή της πτώσης,
   - το ύψος που βρίσκεται το σώμα,
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - την ταχύτητα που έχει το σώμα.
3. Το πρόγραμμα δεν θα διαβάζει το αρχικό ύψος αλλά θα υπολογίζει και θα τυπώνει τα *tP*, *vP* για *h* = 100 *m*, 200 *m*, ..., 1000 *m*.

6-11 *Μέθοδος των ελαχίστων τετραγώνων.* Αν μας δοθούν *n* σημεία του επιπέδου *xy*, (x_k, y_k), *k* = 1..*n*, η «καλύτερη ευθεία», $y = \alpha x + \beta$, που περνάει από αυτά, είναι αυτή που έχει:

$$\alpha = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - \left(\sum x\right)^2} \quad \text{και} \quad \beta = \frac{\sum y \sum x^2 - \sum x \sum xy}{n\sum x^2 - \left(\sum x\right)^2}$$

όπου:

$$\sum x = \sum_{k=1}^{n} x_k \,,\ \sum y = \sum_{k=1}^{n} y_k \,,\ \sum x = \sum_{k=1}^{n} n_k^2 \,,\ \sum xy = \sum_{k=1}^{n} x_k y_k$$

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του *n* και στη συνέχεια τα (x_k, y_k), *k* = 1..*n* και θα υπολογίζει τα α και β.

6-12 Απόδειξε ότι αν:

```
   int  x, z, u;
```

τότε:

```
// x ≥ 0
   z = 0;  u = 1;
   while ( u <= x )     // I: (z² ≤ x) && (u = (z+1)²)
   {
      z = z + 1;
      u = u + z + z + 1;
   }
// z² ≤ x < (z+1)²
```

Απόδειξε πρώτα ότι η $(z^2 \le x)$ **&&** $(u = (z+1)^2)$ είναι αναλλοίωτη της **while**.

Όπως καταλαβαίνεις, το *z* είναι ακέραια προσέγγιση στην $\sqrt{x}$. Συμπλήρωσε το παραπάνω πρόγραμμα έτσι ώστε να διαβάζει από το πληκτρολόγιο την τιμή της *x* και εφόσον είναι σύμφωνη με την προϋπόθεση να υπολογίζει και να τυπώνει την «ακέραια τετραγωνική ρίζα» της *x* με τον παραπάνω τρόπο και με την `static_cast <int>(sqrt(x))`.

6-13 Απόδειξε ότι αν:

```
   int  x, c;
```

τότε:

```
// x ≥ 0
   c = x;
   while ( c*c*c > x )   // a: x < (c + 1)³
      c = c - 1;
// c³ ≤ x < (c + 1)³
```

6-14 Έστω ότι έχουμε την ακολουθία: $a_k = k!/p^k \mid k = 1,2,...$ όπου p φυσικός > 1. Όποια και να είναι η τιμή της p, η ακολουθία είναι τελικώς αύξουσα, αλλά στη αρχή –στους πρώτους όρους– βλέπεις οτι οι τιμές είναι φθίνουσες. Γράψε πρόγραμμα που

- θα διαβάζει την τιμή του p από το πληκτρολόγιο,
- θα υπολογίζει και θα τυπώνει τους $2p$ πρώτους όρους της ακολουθίας,
- θα μας δίνει την τιμή και την τάξη του ελάχιστου από αυτούς.

6-15 Θέλουμε πρόγραμμα που θα παρακολουθεί έναν παίκτη του μπάσκετ κατά τη διάρκεια ενός παιγνιδιού. Το πρόγραμμα θα παίρνει τα εξής στοιχεία:
`'1'` κάθε φορά που ο παίκτης επιτυγχάνει καλάθι με ελεύθερη βολή,
`'2'` κάθε φορά που ο παίκτης επιτυγχάνει δίποντο,
`'3'` κάθε φορά που ο παίκτης επιτυγχάνει τρίποντο και
`'4'` κάθε φορά που ο παίκτης κάνει φάουλ.
Πληκτρολογώντας `'0'` δείχνουμε στο πρόγραμμα ότι τελείωσε το παιχνίδι. Όταν ο παίκτης συμπληρώσει πέντε φάουλ, θα πρέπει να βγαίνει το μήνυμα: `"ΒΓΑΙΝΕΙ ΕΞΩ ΜΕ 5 ΦΑΟΥΛ"`.

Όταν τελειώσει η συμμετοχή του παίκτη –είτε λόγω αποβολής είτε λόγω τέλους του παιχνιδιού– θα γράφεται στην οθόνη η στατιστική του.

6-16 Απόδειξε ότι η:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j: 1..\, k\text{–}1 \bullet t_j \le tmax$$

είναι αναλλοίωτη της **while** στο:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

6-17 α) Γενίκευσε το πρόγραμμα ανάγνωσης ακεραίου της §4.5 ώστε να διαβάζει οποιαδήποτε τιμή τύπου `unsigned long`.

β) Βελτίωσε το πρόγραμμά σου ώστε να διαβάζει και προσημασμένες τιμές.

6-18 Βελτίωσε το πρόγραμμα που έγραψες στην άσκ. 6.17 ώστε να διαβάζει πραγματικούς της μορφής πρόσημο, ακέραιο μέρος, κλασματικό μέρος.

6-19 *Ο Χρόνος στη C++.* Η C++ μας δίνει τη συνάρτηση *time* που, με την κλήση `time(0)`, μας επιστρέφει τα δευτερόλεπτα που πέρασαν από τη στιγμή 00:00:00 *GMT, της* 1ης Ιανουαρίου 1970. Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <ctime>`".

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο ακέραιους αριθμούς και θα σταματάει όταν διαβάσει την τιμή 0. Αρχίζοντας από τον δεύτερο αριθμό, μετά την ανάγνωση θα μας δίνει τον χρόνο που πέρασε από την προηγούμενη πληκτρολόγηση. Στο τέλος θα μας δίνει:

- το πλήθος των πληκτρολογήσεων,

- το μέγιστο χρονικό διάστημα μεταξύ δύο πληκτρολήσεων,
- τον μέσο χρόνο μεταξύ δύο πληκτρολογήσεων.

κεφάλαιο

7

Συναρτήσεις Ι

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μπορείς να γράφεις δικές σου συναρτήσεις και να τις χρησιμοποιείς στα προγράμματά σου.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς στα προγράμματά σου συναρτήσεις που δεν υπάρχουν στις βιβλιοθήκες της γλώσσας και θα τις γράφεις εσύ. Πέρα από αυτό θα μπορείς να γράφεις συναρτήσεις για να επιμερίσεις την πολυπλοκότητα του προγράμματός σου και να μπορείς να χειριστείς πιο εύκολα προβλήματα ελέγχου και επαλήθευσης.

Έννοιες κλειδιά:

- *(ολική) συνάρτηση*
- *μερική συνάρτηση*
- *σύνολο αφετηρίας, σύνολο τιμών*
- *πεδίο ορισμού*
- *τυπική παράμετρος*
- *πραγματική παράμετρος (όρισμα)*
- *εντολή* `return`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Όταν έχουμε να γράψουμε ένα πρόγραμμα για να λύσουμε κάποιο σύνθετο πρόβλημα, χωρίζουμε συνήθως το όλο πρόβλημα σε επιμέρους προβλήματα και μετά γράφουμε ανεξάρτητα τμήματα προγραμμάτων για το κάθε ένα από αυτά. Τα ανεξάρτητα αυτά τμήματα προγραμμάτων λέγονται **υποπρογράμματα** (subprograms) ή, όπως τις λέει η C++, **συναρτήσεις** (functions) και μπορούμε να τα θεωρήσουμε σαν εντολές ή συναρτήσεις μιας γλώσσας προγραμματισμού που δημιουργούμε για να λύσουμε το πρόβλημά μας.

Στη C++ υπάρχουν δύο διαφορετικές κατηγορίες συναρτήσεων: *με τύπο* –που θα δούμε τώρα– και *χωρίς τύπο* (**void**), που θα δούμε αργότερα.

7.1. Συναρτήσεις με Τύπο – Εισαγωγή

Στα προηγούμενα κεφάλαια γνωρίσαμε μερικές *συναρτήσεις*, από τις βιβλιοθήκες της C++, όπως π.χ. οι συναρτήσεις *sqrt()*, *exp()*, *pow()*, *log()*, *cos()*, *sin()* κλπ., που χρησιμοποιούνται συχνά στα προγράμματά μας. Συχνά όμως, χρειαζόμαστε και άλλες συναρτήσεις, που φυσικά δεν μπορούσαν να προβλέψουν αυτοί που σχεδίασαν τη C++. Γι' αυτό η γλώσσα μας δίνει τη δυνατότητα υλοποίησης οποιασδήποτε (κατ' αρχήν) συνάρτησης μέσα στο πρόγραμμα και φυσικά τη δυνατότητα χρήσης αυτής της συνάρτησης.

Ας ξεκινήσουμε με τα μαθηματικά: αν έχουμε δύο σύνολα A και B, μια **μερική συνάρτηση** (partial function) f από το A στο B είναι ένα σύνολο ζευγών (x, y), όπου $x \in A$ και $y \in B$, που δεν περιέχει ζεύγη που να έχουν το ίδιο x και διαφορετικά y. Το A λέγεται **σύνολο αφετηρίας** (domain) και το B **σύνολο** (ή **πεδίο**) **τιμών** (range) της f. Γράφουμε:

$f: A \nrightarrow B$

Το υποσύνολο του A για κάθε μέλος x του οποίου υπάρχει ζεύγος (x, y) στην f, λέγεται **πεδίο ορισμού** (domain of definition) της f. Αν το πεδίο ορισμού είναι ίσο με ολόκληρο το σύνολο αφετηρίας η f λέγεται **ολική συνάρτηση** (total function) ή απλώς **συνάρτηση** (function). Γράφουμε:

$f: A \to B$

Σε κάθε ζεύγος (x, y), το x λέγεται **πρότυπο** και το y **εικόνα** (image). Γράφουμε $y = f(x)$ ή $x \mapsto y$.

Παραδείγματα

Αν συμβολίσουμε με το $\sqrt{}$ το σύνολο των ζευγών $(x, \sqrt{x})$ τότε έχουμε:

$\sqrt{} : \mathbb{R} \nrightarrow \mathbb{R}$

δηλαδή η τετραγωνική ρίζα είναι μερική συνάρτηση από το $\mathbb{R}$ στο $\mathbb{R}$, διότι στο σύνολο $\sqrt{}$ δεν έχουμε ζεύγη για $x < 0$. Αλλά:

$\sqrt{} : \mathbb{R}_{0+} \to \mathbb{R}$

Παρομοίως, αν πάρουμε το σύνολο f των ζευγών $(x, 1/x)$, έχουμε:

$f: \mathbb{R} \nrightarrow \mathbb{R}$

διότι: ενώ $0 \in \mathbb{R}$, δεν υπάρχει στο f ζεύγος με πρώτο μέλος "`0`". Και στην περίπτωση αυτή όμως μπορούμε να έχουμε μια ολική συνάρτηση:

$f: \mathbb{R}^{*} \to \mathbb{R}$

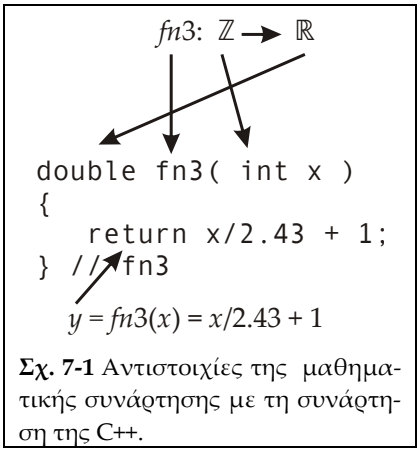
Όπως είδες στα παραδείγματα, στα μαθηματικά μπορούμε να περιορίσουμε το σύνολο αφετηρίας ώστε να γίνει ίσο με το πεδίο ορισμού. Στον προγραμματισμό αυτό δεν είναι πάντοτε εφικτό.

Συνήθως υπάρχει κάποιος *μηχανισμός*, δηλαδή τρόπος υπολογισμού, που μας επιτρέπει από τα πρότυπα x να βρίσκουμε τις εικόνες y ώστε το $(x, y) \in f$. Π.χ.

$$y = f(x) = 2x^2 + \frac{5}{x-1} \quad \text{ή}$$

$$y = f(x) = \begin{cases} -5, & -5.5 \le x \le -4.5 \\ 0, & -4.5 < x < 4.5 \\ 5, & 4.5 \le x \le 5.5 \end{cases}$$

Μια συνάρτηση με τύπο της C++ είναι ένα κομμάτι προγράμματος που υλοποιεί έναν τέτοιο μηχανισμό. Ας πούμε ότι έχουμε τη συνάρτηση *fn3*: $\mathbb{Z} \to \mathbb{R}$ με μηχανισμό υπολογισμού: $y = fn3(x) = \frac{x}{2.43} + 1$. Στο Σχ. 7-1 βλέπεις τι θα γράψουμε στη C++ και τις αντιστοιχίες μεταξύ αυτών που είπαμε πιο πάνω και του ορισμού της συνάρτησης στη C++.



Σχ. 7-1 Αντιστοιχίες της μαθηματικής συνάρτησης με τη συνάρτηση της C++.

Όπως βλέπεις, ο ορισμός μιας συνάρτησης ξεκινάει με μια **επικεφαλίδα**, που αποτελείται:

- από το όνομα τύπου που αντιστοιχεί στο πεδίο τιμών της συνάρτησης –ℝ που μεταφράζεται στον `double`–
- το όνομα της συνάρτησης –στην περίπτωσή μας `fn3`– και
- τον τύπο που αντιστοιχεί στο πεδίο ορισμού –ℤ που μεταφράζεται στον τύπο `int`.

Το *x* είναι μια **παράμετρος** της συνάρτησης· οι παράμετροι αν υπάρχουν γράφονται μέσα σε παρενθέσεις, μετά το όνομα. Στο **σώμα** (body) του υποπρογράμματος γράφεται κομμάτι προγράμματος που υλοποιεί το μηχανισμό υπολογισμού της εικόνας από το πρότυπο.

Μέσα στο σώμα της συνάρτησης θα πρέπει να υπάρχει οπωσδήποτε μια εντολή

`return` *Π*`;`

όπου *Π* μια παράσταση. Η τιμή της *Π*, αφού μετατραπεί στον τύπο *T* της συνάρτησης (`static_cast<`*T*`>`), είναι η τιμή που *επιστρέφει* η συνάρτηση.

Πού γράφουμε τη συνάρτησή μας; *Ο ορισμός μιας συνάρτησης πρέπει να βρίσκεται, κατ' αρχήν, πριν από τη χρήση (κλήση) της*. Το τι σημαίνει αυτό θα γίνει φανερό στη συνέχεια.

Πώς χρησιμοποιούμε μια δική μας συνάρτηση; Όπως ακριβώς χρησιμοποιούμε και τις προδηλωμένες συναρτήσεις της C++, μέσα σε παραστάσεις. Π.χ. μπορούμε να γράψουμε:

```
synist = f1*fn3(h/3) + f2 - f3*cos(phi/3+pi/2);
```

Συνοψίζοντας μπορούμε να πούμε ότι: **συνάρτηση** (function) *με τύπο* στη C++ *είναι ανεξάρτητο υποπρόγραμμα, που υπολογίζει και μεταβιβάζει ένα μοναδικό αποτέλεσμα*. Το αποτέλεσμα αυτό έχει τον δικό του τύπο, ο οποίος μπορεί να διαφέρει από τον τύπο των τυπικών παραμέτρων της συνάρτησης. *Το μοναδικό αποτέλεσμα της συνάρτησης διαβιβάζεται μέσω του ονόματός* της.

7.2. Η Εντολή "`return`"

Είπαμε παραπάνω ότι: με τη "`return` *Π*" καθορίζουμε ότι η συνάρτησή μας θα επιστρέψει την τιμή της Π· ακριβέστερα: *επιστρέφει την τιμή της Π αφού τη μετατρέψει στον τύπο της συνάρτησης*. Αλλά η **return** κάνει και κάτι άλλο: *τελειώνει την εκτέλεση της συνάρτησης στην οποία υπάρχει* και η εκτέλεση συνεχίζεται στο σημείο που κλήθηκε· όσες εντολές ακολουθούν τη **return** δεν θα εκτελεσθούν. Ας δούμε ένα

Παράδειγμα

Για να υπολογίσουμε τη μέγιστη από δύο ακέραιες τιμές γράφουμε την παρακάτω συνάρτηση:

```
int max( int x, int y )
{
```

```
   int fvx;

   if ( x > y )  fvx = x;
          else  fvx = y;
   return fvx;
} // max
```

Ένας άλλος τρόπος να τη γράψουμε είναι ο εξής:

```
int max( int x, int y )
{
   if ( x > y )  return x;
          else  return y;
} // max
```

που είναι οικονομικότερος χωρίς να γίνεται λιγότερο καθαρή η λογική της συνάρτησης. Για δες όμως άλλον έναν τρόπο:

```
int max( int x, int y )
{
   if ( x > y )  return x;
   return y;
} // max
```

Εδώ τι γίνεται; Αν $x > y$ τότε θα εκτελεσθεί η "`return x`" και έτσι η "`return y`" δεν θα εκτελεσθεί ποτέ· αν δεν ισχύει η $x > y$ τότε η "`return x`" θα αγνοηθεί και θα εκτελεσθεί η "`return y`". Άρα, όλα πάνε μια χαρά.

Και οι τρεις μορφές είναι σωστές, αλλά ποια είναι προτιμότερη; Θα προτιμήσουμε την πρώτη διότι συμμορφώνεται με τον κανόνα:

- ***Κάθε συνάρτηση θα πρέπει να έχει μια είσοδο και μια έξοδο (return).***

Αυτό θα μας διευκολύνει σημαντικά στην απόδειξη ορθότητας.

7.3. Μια Ιστορία με Συναρτήσεις

Μας δίνεται το εξής πρόβλημα:

> *Να γραφεί ένα πρόγραμμα που θα διαβάζει τρεις ακέραιους, x1, x2, x3 και θα τους τυπώνει κατ' αύξουσα τάξη.*

Η πιο απλή σκέψη είναι να εξετάσουμε όλες τις δυνατές (6) περιπτώσεις. Ένας άλλος τρόπος είναι:

- βρες τον ελάχιστο και τύπωσέ τον πρώτο,
- βρες το μέγιστο και τύπωσέ τον τελευταίο.

Και ο ενδιάμεσος; Αυτός είναι ο:

$$x1 + x2 + x3 - \text{Μέγιστος} - \text{Ελάχιστος}$$

Ας δούμε τώρα πώς θα υπολογίσουμε το μέγιστο. Θα μπορούσαμε να γράψουμε μια συνάρτηση:

$max3: \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Αυτή θα μεταφραστεί σε μια συνάρτηση C++ με τρεις παραμέτρους και αν προσπαθήσεις να τη γράψεις θα δεις ότι πρέπει να κάνεις αρκετές συγκρίσεις. Ένας άλλος τρόπος είναι να χρησιμοποιήσουμε τη:

$max: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

–που μεταφράσαμε σε μια συνάρτηση της C++ στην προηγούμενη παράγραφο– και να κάνουμε συγκρίσεις ανά δύο: αν $max(x1, x2)$ είναι ο μέγιστος από τους $x1$, $x2$, τότε ο μέγιστος από τους $x1$, $x2$, $x3$ είναι ο $max(max(x1, x2), x3)$.

Με τον ίδιο τρόπο μπορούμε να γράψουμε και να χρησιμοποιήσουμε μια

$min: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

για το ελάχιστο:

```
   int min( int t, int u )
   {
      int fvn;

      if ( t < u )  fvn = t;
              else  fvn = u;
      return fvn;
   } // min
```

Δες ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;

   // max -- Επιστρέφει την τιμή της μέγιστης των παραμέτρων
   int max( int x, int y )
   {
      int fvx;

      if ( x > y )  fvx = x;
              else  fvx = y;
      return fvx;
   } // max

   // min -- Επιστρέφει την τιμή της ελάχιστης των παραμέτρων
   int min( int t, int u )
   {
      int fvn;

      if ( t < u )  fvn = t;
              else  fvn = u;
      return fvn;
   } // min

int main()
{
   int x1, x2, x3;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   cout << min( min(x1, x2), x3 ) << "  "
        << ( x1+x2+x3 - min(min(x1,x2),x3) - max(max(x1,x2),x3) )
        << "  " << max( max(x1, x2), x3 ) << endl;
} // main
```

Ας δούμε τώρα πώς δουλεύει αυτό το πρόγραμμα. Οι συναρτήσεις γράφτηκαν *πριν από τις κλήσεις τους*, που υπάρχουν στη `main`. Αλλά η εκτέλεση αρχίζει από την πρώτη εντολή της `main` και βλέπουμε στην οθόνη μας:

```
 Δώσε τρεις ακέραιους: 12 43 5<enter>
```

Σύμφωνα με αυτά που ξέρουμε: το "`12`" θα γίνει τιμή της *x1*, το "`43`" της *x2* και το "`5`" της *x3*.

Ας πούμε ότι τα ορίσματα της "`cout << ...`" υπολογίζονται με τη σειρά που γράφονται[1]. Το πρώτο που θα γίνει είναι να υπολογιστεί η κλήση:

```
   min( min(x1, x2), x3 )
```

Οι `min(x1, x2)` και `x3` είναι οι **πραγματικές παράμετροι** (actual parameters) ή **ορίσματα** (arguments) που αντιστοιχούν στις **τυπικές παραμέτρους** (formal parameters) της συνάρτησης *min*: Η `min(x1, x2)` αντιστοιχεί στην `t` και η `x3` στη `u`.

- ***Ο υπολογισμός κλήσης μιας συνάρτησης ξεκινάει από τον υπολογισμό των πραγματικών παραμέτρων.***

Για τον υπολογισμό της δεύτερης παραμέτρου δεν χρειάζεται κάτι ιδιαίτερο: απλώς παίρνουμε από τη μνήμη την τιμή της *x3* (5). Η πρώτη παράμετρος `min(x1, x2)` είναι μια

[1] Και να μην ισχύει αυτό, τα πράγματα δεν αλλάζουν.

κλήση συνάρτησης (και πάλι της *min()*)· εδώ έχουμε τις αντιστοιχίες: `x1` (με τιμή 12) στην `t` και `x2` (με τιμή 43) στη `u`. Ας δούμε πώς θα γίνει ο υπολογισμός της.

Κατ' αρχάς παραχωρείται στη συνάρτηση *min()* μνήμη για όλα τα τοπικά αντικείμενα:

- μια θέση μνήμης για τιμές τύπου `int`: τυπική παράμετρος `t`,
- μια θέση μνήμης για τιμές τύπου `int`: τυπική παράμετρος `u`,
- μια θέση μνήμης για τιμές τύπου `int`: μεταβλητή `fvn`.

Στη συνέχεια αντιγράφονται οι τιμές των πραγματικών παραμέτρων στις αντίστοιχες τυπικές. Έτσι, η `t` παίρνει τιμή "`12`" (από τη `x1`) και η `u` τιμή "`43`" (από τη `x2`).

Μετά από αυτό εκτελούνται οι εντολές που υπάρχουν στο σώμα της συνάρτησης μέχρι να βρεθεί εντολή **return**. Στην περίπτωσή μας εκτελείται η **ifelse** από την οποία η `fvn` παίρνει τιμή "`12`".

Με την εκτέλεση της **return** διακόπτεται η εκτέλεση της συνάρτησης. Η συνάρτηση επιστρέφει τη μνήμη που της παραχωρήθηκε (*t*, *u*, *fvn*). Η τιμή της παράστασης που υπάρχει στη **return** (στην περίπτωσή μας της "`fvn`") αντικαθιστά την κλήση της συνάρτησης.

Έτσι η "`min(min(x1, x2), x3)`" γίνεται: "`min(12, 5)`". Για τον υπολογισμό της ξαναγίνονται τα ίδια: της παραχωρείται μνήμη (για τις *t*, *u*, *fvn*), αντιγράφεται το "`12`" στην *t* και το "`5`" στη *u*, εκτελείται η **ifelse** και η *fvn* παίρνει τιμή "`5`". Με την εκτέλεση της "`return fvn`" τελειώνει η εκτέλεση της *min()* και η κλήση "`min(min(x1, x2), x3)`" αντικαθίσταται από το "`5`" (που είναι και η πρώτη τιμή που θα τυπωθεί).

Ολόκληρη αυτή η ιστορία θα επαναληφθεί και όταν θα έρθει η ώρα για τον υπολογισμό του τρίτου ορίσματος της "`cout << ...`", όπου έχουμε:

```
. . . x3 - min( min(x1,x2),x3 ) - max(. . .
```

Παρόμοια θα γίνουν και με τη *max()*.

Ας δούμε τώρα δύο ακόμη σημεία σχετικά με το πρόγραμμά μας:

1. Σε σχέση με το παράδειγμα, θα πρέπει να σημειώσουμε και το εξής: η κλήση και η εκτέλεση μιας συνάρτησης καταναλώνει συνήθως κάποιο υπολογίσιμο χρόνο. Συνεπώς πρέπει να αποφεύγουμε τις αναφορές σε συναρτήσεις που δεν είναι απαραίτητες, π.χ. στις περιπτώσεις που οι αναφορές στη συνάρτηση παίρνουν τις ίδιες πραγματικές παραμέτρους. Έτσι είναι προτιμότερο να γράψουμε τη `main()` ως εξής:

```
int main()
{
   int x1, x2, x3;
   int xmin, xmax;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   xmin = min( min(x1, x2), x3 );
   xmax = max( max(x1, x2), x3 );
   cout << xmin << "  " << x1 + x2 + x3 - xmin - xmax << "  "
        << xmax << endl;
} // main
```

Δηλαδή, να δηλώσουμε δυο μεταβλητές:

```
   int xmin, xmax;
```

στις οποίες να αποθηκεύσουμε αντίστοιχα τις "`min(min(x1, x2), x3)`" και "`max(max(x1, x2), x3)`". Στη συνέχεια, στην εντολή εξόδου χρησιμοποιούμε από δύο φορές τη *xmin* και την *xmax* αντί να ξανακαλούμε τις *min()* και *max()*. Έτσι, έχουμε δυο μόνον αναφορές στη *min* και δυο στη *max*, αντί για τέσσερις που είχαμε αρχικά.

Πλαίσιο 7.1

Ορισμός Συνάρτησης

- ξεκινάει με το όνομα τύπου του αποτελέσματος,
- στη συνέχεια ακολουθεί το όνομα της συνάρτησης, που είναι σαν όλα τα ονόματα της C++,
- μετά, μέσα σε παρενθέσεις, παρατίθενται τα τυπικά ορίσματα, αν υπάρχουν, και
- τέλος υπάρχει η μια σύνθετη εντολή.

Ο ορισμός μιας συνάρτησης είναι ένα κομμάτι προγράμματος που περιγράφει τον τρόπο που θα υπολογισθεί η τιμή της συνάρτησης όταν κληθεί. Για τον υπολογισμό της τιμής η συνάρτηση μπορεί να εξοπλισθεί με τοπικά αντικείμενα (σταθερές, μεταβλητές κλπ). Επικοινωνεί με το πρόγραμμα ή τη συνάρτηση, που την καλεί, με παραμέτρους ή καθολικά αντικείμενα.

Μέσα στο σώμα της συνάρτησης πρέπει να υπάρχει μια τουλάχιστον εντολή `return` *Π*, όπου *Π* μια παράσταση που να ορίζει την τιμή που επιστρέφει.

2. Οι προγραμματιστές που έχουν πείρα σε C δεν θα έγραφαν το πρόγραμμα όπως το γράψαμε, αλλά ως εξής:

```
#include <iostream>
using namespace std;

   int max( int x, int y );
   int min( int t, int u );

int main()
{
   int x1, x2, x3;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   cout << min( min(x1, x2), x3 ) << "  "
        << ( x1+x2+x3 - min(min(x1,x2),x3) - max(max(x1,x2),x3) )
        << "  " << max( max(x1, x2), x3 ) << endl;
} // main
   // max -- Επιστρέφει την τιμή της μέγιστης των παραμέτρων
   int max( int x, int y )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

   // min -- Επιστρέφει την τιμή της ελάχιστης των παραμέτρων
   int min( int t, int u )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Και αυτό που είπαμε, ότι μια συνάρτηση πρέπει να ορίζεται πριν από την κλήση της, δεν ισχύει; Θα το διατυπώσουμε κάπως πιο ελαστικά:

♦ ***Πριν από οποιαδήποτε κλήση μιας συνάρτησης θα πρέπει να υπάρχει ο ορισμός ή, απλώς, η δήλωσή της.***

Οι δηλώσεις των δύο συναρτήσεων θα μπορούσαν να γίνουν και ως εξής:

```
   int max( int, int );
   int min( int, int );
```

ή ακόμη:

```
   int max( int, int ), min( int, int );
```

7.4. Η Συνάρτηση στο Πρόγραμμα

Στην προηγούμενη παράγραφο είδαμε τι περίπου γίνεται με τις συναρτήσεις. Τώρα θα τα ξαναπούμε αλλά σε μεγαλύτερη έκταση. Θα πρέπει όμως να σου δώσουμε μια συμβουλή: *Αν δεν καταλάβεις τα πάντα από αυτήν την παράγραφο, μην προχωρήσεις παρακάτω*. Πέρασε τα προγράμματα στον ΗΥ και σιγουρέψου ότι παίρνεις ακριβώς τα ίδια αποτελέσματα με αυτά που σου δίνουμε. Αν έχεις αμφιβολία για οτιδήποτε, κάνε μερικά πειράματα μέχρι να τη λύσεις.

Ένα πρόγραμμα της C++ είναι ένα *σύνολο από συναρτήσεις*. Μια από αυτές τις συναρτήσεις έχει το όνομα "`main`" και από αυτήν αρχίζει η εκτέλεση του προγράμματος (δες την §7.6).

7.4.1 Εμβέλεια και Χρόνος Ζωής

Κάθε συνάρτηση είναι μια *σύνθετη εντολή* (σώμα της συνάρτησης) με μια επικεφαλίδα. Στην επικεφαλίδα, δηλώνεται και ένα όνομα που χαρακτηρίζει τη συνάρτηση. Η σύνθετη εντολή αποτελείται από τις δηλώσεις των σταθερών, των τύπων και των μεταβλητών και τις άλλες εντολές. Στις εντολές αυτές μπορεί να περιλαμβάνονται *άλλες σύνθετες εντολές με το περιεχόμενο που περιγράφουμε* (δηλ. δηλώσεις κλπ).

Κάθε οντότητα (σταθερά, μεταβλητή κλπ) που δηλώνεται μέσα σε μια συνάρτηση είναι **τοπική** (local) στη συνάρτηση. *Τοπικά είναι και τα ονόματα των παραμέτρων μιας συνάρτησης*: τα χειριζόμαστε σαν να δηλώνονται στη σύνθετη εντολή που ακολουθεί την επικεφαλίδα. Το όνομα μιας τοπικής οντότητας είναι άγνωστο έξω από τη συνάρτηση που δηλώνεται (εκεί μπορεί να χρησιμοποιείται για κάποια άλλη οντότητα).

Για παράδειγμα, ας ξαναγυρίσουμε στο πρόγραμμα της §7.4. Εδώ:

- Οι μεταβλητές *x*1, *x*2, *x*3 είναι τοπικές στη `main()`· π.χ. αν βάλεις μια "`cout << x1 << x2 << x3`" μέσα στην *max()* (ή μέσα στη *min()*) θα πάρεις μήνυμα λάθους από τον μεταγλωττιστή (unknown identifier ή κάτι παρόμοιο), που θα σου πει ότι δεν ξέρει αυτά τα ονόματα.
- Η *fvx* δηλώνεται στην *max()* και είναι γνωστή μόνο μέσα σε αυτήν. Αν προσπαθήσεις να χρησιμοποιήσεις την *fvx* στη `main()` ή στη *min()* θα πάρεις μήνυμα λάθους από το μεταγλωττιστή. Παρομοίως, η *fvn* είναι γνωστή μόνο στη *min()*.
- Ξαναγράφουμε τη *min()* ως εξής:

```
int min( int x, int y )
{
   int fvn;

   if ( x < y )  fvn = x;
          else  fvn = y;
   return fvn;
} // min
```

Ας έρθουμε στις *x*, *y*: υπάρχουν παράμετροι με το όνομα αυτό και στη *max()* και στη

Πλαίσιο 7.2

Κανόνας Εμβέλειας

Κάθε δήλωση ισχύει μόνον στη συνάρτηση όπου έγινε. Αυτό ισχύει και για τις παραμέτρους της συνάρτησης.

Κανόνας Διάρκειας Ζωής

Κάθε αντικείμενο μιας συνάρτσης υπάρχει όσο εκτελείται η συνάρτηση όπου έχει δηλωθεί.

min()· δεν γίνεται μπέρδεμα; Όχι! Οι παράμετροι της *min()* είναι γνωστές μόνο μέσα στη *min()*, ενώ οι παράμετροι της *max* είναι γνωστές μόνο μέσα στη *max()*.

Λέμε ότι η **εμβέλεια** (scope)

- των ονομάτων *x1*, *x2*, *x3* είναι η `main()`,
- των *x*, *y* (που δηλώνονται στη *max()*) και *fvx* είναι η *max* και
- των *x*, *y* (που δηλώνονται στη *min()*) και *fvn* είναι η *min*.

Τώρα ας έρθουμε σε ένα άλλο πρόβλημα: Για πόσο χρόνο «ζουν» οι μεταβλητές μιας συνάρτησης; Στην προηγούμενη παράγραφο λέγαμε ότι: «[*όταν κληθεί η min()*] *κατ' αρχάς παραχωρείται στη συνάρτηση min() μνήμη για όλα τα τοπικά αντικείμενα: α) μια θέση μνήμης για τιμές τύπου* `int` *για την τυπική παράμετρο t, β) μια θέση μνήμης για τιμές τύπου* `int` *για την τυπική παράμετρο u, γ) μια θέση μνήμης για τιμές τύπου* `int` *για τη μεταβλητή fvn*» και «*Με την εκτέλεση της* `return` *διακόπτεται η εκτέλεση της συνάρτησης. Η συνάρτηση επιστρέφει τη μνήμη που της παραχωρήθηκε (t, u, fvn).*» Γενικώς ισχύει ο «Κανόνας Διάρκειας Ζωής» που βλέπεις στο Πλ. 7.2.

Αργότερα θα επεκτείνουμε τους κανόνες του Πλ. 7.2.

7.4.2 Παράμετροι

Ας δούμε τώρα τώρα ένα άλλο θέμα. Λέγαμε στην προηγούμενη παράγραφο ότι, αφού υπολογισθούν οι τιμές των παραγματικών παραμέτρων και παραχωρηθεί η μνήμη που χρειάζεται για τις τυπικές, «*αντιγράφονται οι τιμές των πραγματικών παραμέτρων στις αντίστοιχες τυπικές.*» Η αντιγραφή γίνεται αφού πρώτα γίνει η κατάλληλη *αλλαγή τύπου*, αν αυτή είναι δυνατή βέβαια. Ας δούμε ένα παράδειγμα. Ας πούμε ότι έχουμε:

```
int ai( int x )
{    . . .
   cout << x << endl; . . . }

int al( long int x )
{    . . .
   cout << x << endl; . . . }

int ac( char x )
{    . . .
   cout << x << endl; . . . }

int ab( bool x )
{    . . .
   cout << x << endl; . . . }

int ad( double x )
{    . . .
   cout << x << endl; . . . }

int af( float x )
{    . . .
   cout << x << endl; . . . }
```

και καλούμε:

```
   x = ai( 65.789 );
   x = al( 65.789 );
   x = ac( 65.789 );
   x = ab( 65.789 );
   x = ad( 65.789 );
   x = af( 65.789 );
```

Όπως βλέπεις, σε όλες τις κλήσεις, άσχετα από τον τύπο του τυπικού ορίσματος, έχουμε βάλει ως όρισμα μια τιμή τύπου `double`.

Οι εντολές εξόδου που υπάρχουν στις συναρτήσεις θα δώσουν:

Πλαίσιο 7.3

Κλήση Συνάρτησης

Ο υπολογισμός κλήσης μιας συνάρτησης γίνεται ως εξής:

- πρώτα υπολογίζονται οι τιμές των πραγματικών παραμέτρων –για τις παραμέτρους τιμής,
- οι τιμές αυτές αντιγράφονται στις αντίστοιχες τυπικές παραμέτρους, αφού γίνουν μετατροπές τύπου, όπου είναι απαραίτητο,
- εκτελούνται οι εντολές του υποπρογράμματος και υπολογίζεται η τιμή της συνάρτησης,
- η τιμή της συνάρτησης επιστρέφεται με την εντολή **return**.

Αυτή η τιμή αντικαθιστά την κλήση της συνάρτησης στην παράσταση για να γίνουν οι άλλες πράξεις.

```
65
65
A
1
65.789
65.789
```

Καταλαβαίνεις ότι, όπου χρειάστηκε, έγιναν οι μετατροπές τύπου. Π.χ. στην πρώτη η *x* πήρε τιμή `int`(65.789) = 65, στην τρίτη `char`(`int`(65.789)) = `'A'`, στην τέταρτη `bool`(`int`(65.789)) = `true` = 1.

Θα μπορούσαμε να θεωρήσουμε το πέρασμα παραμέτρου σαν εντολή εκχώρησης; Π.χ. για την κλήση:

```
. . .min(x1, x2). . .
```

θα μπορούσαμε να πούμε ότι έχουμε τις εντολές:

```
   t = x1;  u = x2;
```

Με αυτά που έχουμε μάθει μέχρι τώρα, αυτό δεν είναι λάθος. Πάντως πιο σωστό είναι να το παρομοιάσουμε με δήλωση των *t*, *u* με αρχικές τιμές:

```
int  t( x1 ), u( x2 );
```

Μετά από αυτές τις αρχικές τιμές μπορούμε να αλλάζουμε τις τιμές των παραμέτρων, αλλά οι αλλαγές αυτές *δεν περνούν στη συνάρτηση που έκανε την κλήση*. Για παράδειγμα, ας πούμε ότι έχουμε:

```
#include <iostream>
using namespace std;

long int succ( long int n )
{
   n = n + 1;
   return n;
} // succ

int main()
{
   int x = 100, y;

   y = succ( x );
   cout << x << "  " << y << endl;
} // main
```

που θα δώσει:

```
100  101
```

Με την κλήση: "`y = succ(x)`" η τιμή της *x* (= 100) της `main()` έγινε αρχική τιμή της *n* της *succ()*. Στη *succ()* η τιμή της *n* αυξήθηκε κατά 1 και η αυξημένη τιμή (101) επιστράφηκε

ως τιμή της συνάρτησης και αποθηκεύτηκε στην *y* της `main`. Όπως φαίνεται και από το αποτέλεσμα, *η τιμή της x δεν άλλαξε*. Πάντως, αν έχεις καταλάβει όσα είπαμε για το πώς γίνεται το πέρασμα των τιμών των παραμέτρων, τα παραπάνω είναι αυτονόητα.

Οι παράμετροι που λειτουργούν σαν «μονόδρομοι», μεταφέροντας στοιχεία από την κλήση προς τη συνάρτηση μόνον λέγονται **παράμετροι τιμής** (value parameters). Αργότερα θα μάθουμε ότι υπάρχουν μηχανισμοί α) για να παίρνουμε τις αλλαγές τιμών των παραμέτρων[2] β) για να μην επιτρέπουμε τέτοιες αλλαγές.

7.4.3 Αρχικές Τιμές Μεταβλητών

Στο Κεφ. 2 λέγαμε ότι «*η C++ σου επιτρέπει μαζί με τη δήλωση να δώσεις στη μεταβλητή σου και αρχική τιμή.*» Ας δούμε τώρα αυτήν τη δυνατότητα πιο εκτεταμένα.

Για να δηλώσουμε μια μεταβλητή *v* τύπου *T* μπορούμε να δώσουμε:

```
T v( Π );
```

όπου *Π* μια παράσταση που μπορεί να περιέχει σταθερές, μεταβλητές που έχουν ήδη τιμή και συναρτήσεις, αν φυσικά η δήλωσή μας βρίσκεται μέσα στην εμβέλειά τους. Η τιμή της *Π* υπολογίζεται με τις τιμές που έχουν οι μεταβλητές που υπάρχουν σε αυτήν όταν εκτελείται η δήλωση, όταν δηλαδή παραχωρείται μνήμη για τη *v*.

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;

int f( double x )
{
   return (2*x+1)/(x*x+1);
} // f

int main()
{
   double x( 1.5 );
   int n, a( 1 );
   double q( sqrt(2) ), qp1( q + 1 + a/2.0 ), r( f(x/3) );

   cout << " n = " << n << "   a = " << a << endl;
   cout << q << "  " << qp1 << "  " << f(q) << "  " << r << endl;
   a = 2;
   cout << qp1 << endl;
} // main
```

που δίνει:

```
n = 50920484   a = 1
1.41421  2.91421  1  1
2.91421
```

Ας εξετάσουμε προσεκτικά τα αποτελέσματα:

- Στην πρώτη γραμμή παίρνουμε τις τιμές της *n* και της *a* μόλις αρχίσει η εκτέλεση του προγράμματος. Η *n* είναι αόριστη και η τιμή της είναι τυχαία. Η *a* όμως έχει αρχική 1.
- Η αρχική τιμή της *q* είναι *sqrt*(2) (≈1.41421). Η αρχική τιμή της *qp*1 ορίζεται με χρήση των τιμών των *q* και *a* που είναι ήδη ορισμένες. Η τιμή της *r* καθορίζεται με κλήση της συνάρτησης *f* (η `(2*x+1)/(x*x+1)` έχει τιμή τύπου `double` (1.6) αλλά αυτή μετατρέπεται σε `int` (1) αφού αυτός είναι ο τύπος της συνάρτησης).
- Παρομοίως μπορείς να ερμηνεύσεις και την τιμή (1) της *f*(*q*).

[2] Ήδη στην §4.5 μάθαμε ότι με την "`cin.get( a );`" αλλάζει η τιμή της *a*.

- Αλλάξαμε την τιμή της *a* σε 2. Η τιμή της *qp*1 δεν αλλάζει, όπως άλλωστε το περιμένουμε.

Οι τοπικές μεταβλητές μιας συνάρτησης μπορεί να έχουν αρχική τιμή την τιμή κάποιας παραμέτρου. Δες πώς μπορούμε να γράψουμε τη *max()*:

```
int max( int x, int y )
{
   int fvx( y );

   if (x > y)  fvx = x;

   return fvx;
} // max
```

7.5. Η Συνάρτηση "main"

Η `main()` δεν είναι τίποτε άλλο από μια ακόμη συνάρτηση. Αλλά η C++ βάζει μια υποχρέωση στον προγραμματιστή:

- ***Σε κάθε πρόγραμμα θα πρέπει να υπάρχει μια συνάρτηση*** `int main`***· από αυτήν αρχίζει η εκτέλεση του προγράμματος.***[3]

Δεν θα πρέπει να έχει και μια εντολή **return**; Βεβαίως! Και, πιθανότατα, η C++ που δουλεύεις να βγάζει και σχετική προειδοποίηση, αν ακολουθείς το παράδειγμά μας και δεν βάζεις στο τέλος της `main()` μια "`return 0`". Πάντως, το πρότυπο της C++ λέει ότι: *αν η εκτέλεση φτάσει στο τέλος της* `main()` (*επειδή όλα πήγαν καλά και δεν υπήρξε εντολή* **return**) *θα πρέπει να εκτελείται η εντολή* "`return 0`". Αυτό το "`0`" επιστρέφεται στο ΛΣ και σημαίνει ότι η εκτέλεση του προγράμματος ολοκληρώθηκε επιτυχώς. Το ΛΣ σου δίνει τρόπους για να ελέγξεις αυτήν την τιμή.

Αργότερα θα μάθουμε ότι η `main()` (μπορεί να) έχει και παραμέτρους.

7.6. Παράμετρος "unsigned";

Πριν προχωρήσουμε στα παραδείγματά μας θα αναδείξουμε ένα πρόβλημα που μπορεί να σου προκύψει αν βάζεις παραμέτρους `unsigned` (π.χ. `unsigned int`) στις συναρτήσεις σου.

Ας πούμε ότι γράφουμε μια συνάρτηση που υπολογίζει την «ακέραιη τετραγωνική ρίζα φυσικού αριθμού» –μόνον με προσθέσεις– με βάση αυτά που είδαμε στην άσκ. 6-12:

```
unsigned int intSqrt( unsigned int x )
{
   int  z( 0 ), u( 1 );
// x >= 0
   while ( u <= x )     // I: (z² <= x) && (u = (z+1)²)
   {
      z = z + 1;
      u = u + z + z + 1;
   }
// z² <= x < (z+1)²
   return z;
} // intSqrt
```

Τύπος παραμέτρου; `unsigned int`, τι πιο φυσιολογικό! Δες τώρα δυο δοκιμές χρήσης:

```
   n = 1024;
   cout << n << "  " << intSqrt(n) << endl;
   n = -1024;
   cout << n << "  " << intSqrt(n) << endl;
```

Αποτέλεσμα:

[3] Στα προγράμματα για Windows η αντίστοιχη συνάρτηση ονομάζεται *WinMain*.

```
1024  32
-1024  699732
```

Δηλαδή, δούλεψε και στη δεύτερη δοκιμή, αλλά... τι έβγαλε! Ξανακάνουμε τη δεύτερη δοκιμή αλλά βάζουμε μια εντολή εξόδου μέσα στη συνάρτηση:

```
unsigned int intSqrt( unsigned int x )
{
   int  z( 0 ), u( 1 );
cout << x << endl;
   . . .
```

Αποτέλεσμα:

```
4294966272
-1024  699732
```

Όπως βλέπεις, το -1024 περνάει στη συνάρτηση ως 4294966272 και όλα δουλεύουν χωρίς οποιαδήποτε ένδειξη για το ότι κάτι δεν πάει καλά. Αργότερα θα καταλάβεις πώς και γιατί συμβαίνουν αυτά.

Προς το παρόν:

- ***Μην βάζεις στις συναρτήσεις σου παραμέτρους τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

Στην περίπτωσή μας θα έχουμε:

```
unsigned int intSqrt( int x )
{
   int  z( 0 ), u( 1 );

   if ( x >= 0 )
   { // x >= 0
      while ( u <= x )     // I: (z² <= x) && (u = (z+1)²)
      { . . .
```

Στα παραδείγματα στη συνέχεια θα δεις έναν τρόπο αντίδρασης αν δεν ισχύει η προϋπόθεση. Αργότερα θα δούμε και άλλους.

7.7. Παραδείγματα

Να δούμε τώρα ολοκληρωμένα προγράμματα που περιέχουν ορισμούς και χρήσεις συναρτήσεων.

Παράδειγμα 1

Να γραφεί πρόγραμμα που να διαβάζει δυο ακέραιους, *m*, *n* και να υπολογίζει και να τυπώνει το πλήθος των συνδυασμών *m* αντικειμένων ανά *n*.

Όπως είναι γνωστό από τα Μαθηματικά, αυτό είναι:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

όπου, το *n*! (*n* παραγοντικό), ορίζεται ως εξής:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 1 \end{cases} \quad \text{ή ισοδυνάμως:} \quad n! = \begin{cases} 1 & n = 0 \\ 1 \cdot \ldots \cdot (n-1) \cdot n & n \geq 1 \end{cases}$$

Αν είχαμε μια συνάρτηση με το όνομα, ας πούμε, *factorial()*, που να μας υπολογίζει το παραγοντικό του ορίσματος, το πρόγραμμά μας θα ήταν πολύ απλό:

```
   cin >> m >> n;
   comb = factorial(m) / (factorial(n) * factorial(m-n));
   cout << comb << endl;
```

Ας τη γράψουμε, μεταφράζοντας τον δεύτερο ορισμό. Ξεκινούμε από την επικεφαλίδα. Ποια είναι τα πεδία ορισμού και τιμών της συνάρτησής μας:

!: $\mathbb{N} \rightarrow \mathbb{N}^*$

Πώς θα μεταφράσουμε τα $\mathbb{N}$, $\mathbb{N}^*$ στη C++; Προφανώς στον **unsigned int** και για τα δύο σύνολα. Επειδή όμως το *n*! αυξάνεται πολύ γρήγορα, καθώς αυξάνεται το *n*, καλύτερα να βάλουμε σύνολο τιμών το **unsigned long int**. Θα έπρεπε δηλαδή να γράψουμε μια συνάρτηση:

factorial: unsigned int → unsigned long int

αλλά μετά από αυτά που είδαμε στην προηγούμενη παράγραφο θα γράψουμε μια:

factorial: int ↛ unsigned long int

Προχωρούμε μεταφράζοντας τον μηχανισμό:

```
unsigned long int factorial( int a )
{
   unsigned long int fv;

   if ( a < 0 )
   {
      cout << " η factorial κλήθηκε με αρνητικό" << endl;
      exit( EXIT_FAILURE );
   }
   else
   {
      if ( a == 0 )
         fv = 1;
      else
         Υπολόγισε το fv = 1*2*...*(a-1)*a;
   }
    return fv;
} // factorial
```

Και αυτό το "**exit(EXIT_FAILURE)**" τι είναι; Η *exit()* είναι μια συνάρτηση που δεν επιστρέφει τιμή –σαν την *assert()*– που η δήλωσή της υπάρχει στο **cstdlib**. Η εκτέλεσή της έχει ως αποτέλεσμα τη *διακοπή της εκτέλεσης του προγράμματος* (όχι μόνον της συνάρτησης). Η τιμή της παραμέτρου που βάζουμε πηγαίνει στο ΛΣ και μπορεί να ελεγχθεί. Συνήθως, μια μη μηδενική τιμή της παραμέτρου δείχνει ότι η εκτέλεση του προγράμματος διακόπηκε επειδή υπήρξε κάποιο πρόβλημα. Στο **cstdlib** ορίζεται επίσης η σταθερά **EXIT_FAILURE** ως "1".

Με βάση τα παραπάνω, μπορούμε να γράψουμε μια πιο απλή αλλά ισοδύναμη μορφή:

```
unsigned long int factorial( int a )
{
   unsigned long int fv;

   if ( a < 0 )
   {
      cout << " η factorial κλήθηκε με όρισμα " << a << endl;
      exit( EXIT_FAILURE );
   }
   if ( a == 0 )
      fv = 1;
   else
      Υπολόγισε το fv = 1*2*...*(a-1)*a;
   return fv;
} // factorial
```

Από εδώ και πέρα κάπως έτσι θα βάζουμε στις συναρτήσεις μας τον έλεγχο προδιαγραφών.

Στο Πλ. 6.2β είδαμε πώς υπολογίζουμε ένα γινόμενο. Για την περίπτωσή μας:

```
      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
```

Οι *fv* και *k* είναι *τοπικές* μεταβλητές, που δηλώνονται *μέσα στη συνάρτηση*.

Πριν δώσουμε το τελικό πρόγραμμα, ας παρατηρήσουμε ότι ο διαχωρισμός της *a* == 0 είναι άχρηστος μια και ο υπολογισμός της περίπτωσης *a* > 0 μας δίνει σωστό αποτέλεσμα και για *a* == 0.

```
#include <iostream>
#include <cstdlib>
using namespace std;

   // factorial -- Υπολογίζει το a!
   unsigned long int factorial( int a )
   {
      unsigned long int fv;
      int k;

      if ( a < 0 )
      {
         cout << " η factorial κλήθηκε με όρισμα " << a << endl;
         exit( EXIT_FAILURE );
      }
      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
      return fv;
   } // factorial

int main()
{
   int m, n, comb;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   if ( n < 0 || m < 0 )
      cout << " θέλω m >= 0 και n >= 0 " << endl;
   else if ( m < n )
      cout << " θέλω m >= n" << endl;
   else
   {
      comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb << endl;
   }
} // main
```

Κοίταξε την εντολή:

```
      comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
```

Εδώ έχεις μια παράσταση με τρεις κλήσεις της συνάρτησης *factorial()*. Ας δούμε ένα παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m <= n <= 50: 5 2
```

Με την εκτέλεση της `cin >> m >> n` θα έχουμε: *m* = 5 και *n* = 2:

- Ο υπολογισμός της `factorial( m )` θα γίνει ως εξής: Η *a* της *factorial()* θα πάρει την τιμή του *m* = 5. Θα εκτελεσθούν οι εντολές της *factorial()* και τελικώς θα υπολογισθεί η τιμή που επιστρέφει (= 120).
- Παρομοίως θα γίνει ο υπολογισμός της κλήσης `factorial( n )` (= 2).
- Τέλος, για να υπολογισθεί η `factorial( m - n )`, πρώτα υπολογίζεται η τιμή του *m* – *n* (= 3). Αυτή γίνεται τιμή της *a* στην *factorial()* και μετά την εκτέλεση των εντολών της, επιστρέφεται η τιμή 6.

Η παράσταση που πρέπει να υπολογισθεί είναι πια η[4]: 120 / (2 * 6) και η *comb* παίρνει την τιμή 10:

[4] Ο προσεκτικός αναγνώστης θα έχει σημειώσει ότι κάνουμε περισσότερες πράξεις από όσες χρειάζεται. Αλλά, αυτό που θέλουμε να δείξουμε είναι η χρήση της συνάρτησης σε ένα πρόγραμμα.

```
Συνδυασμοί των 5 ανά 2 = 10
```

Παράδειγμα 2

Να γραφούν δύο συναρτήσεις που θα υπολογίζουν το Ελάχιστο Κοινό Πολλαπλάσιο (ΕΚΠ, least common multiple, lcm) και το Μέγιστο Κοινό Διαιρέτη (ΜΚΔ, gratest common divisor, gcd) δύο φυσικών αριθμών. Να γραφεί πρόγραμμα που θα διαβάζει δυο φυσικούς αριθμούς και καλώντας τις συναρτήσεις θα δίνει το ΕΚΠ και τον ΜΚΔ τους.

Να σου υπενθυμίσουμε εδώ ορισμένα πράγματα, από την Αριθμητική: Αν x, y φυσικοί αριθμοί τότε ΕΚΠ(x, y)·ΜΚΔ(x, y) = x·y. Αν βρούμε λοιπόν το ένα από τα δύο, τότε το άλλο υπολογίζεται εύκολα.

Αλλά, στην Αριθμητική είχες μάθει μια μέθοδο για να υπολογίζεις τον ΜΚΔ: τον Ευκλείδειο αλγόριθμο[5]. Ας την θυμίσουμε:

```
Πάρε το υπόλοιπο της ακέραιης διαίρεσης x δια y (x % y)
Ψάξε να βρεις τον ΜΚΔ(y, x % y).
Δηλ., βάλε Νέο x = y και Νέο y = x % y.
      ( ΜΚΔ(x,y) = ΜΚΔ(y, x % y) )
Συνέχισε έτσι, μέχρι να βρεις Νέο y == 0.
```

και ας ξεκινήσουμε από τη συνάρτηση για τον ΜΚΔ. Τα παραπάνω γράφονται σε ψευδοκώδικα:

```
while ( y != 0 )          // ΜΚΔ(x,y) = ΜΚΔ(y,x % y)
{
   b = y;                 // Φύλαξε την παλιά τιμή του y
   Νέο y = x % y;
   Νέο x = b;             // παλιά τιμή του y
}
return x;                 // ΜΚΔ(x,0) = x
```

Αυτά μεταφράζονται εύκολα σε C++, αλλά πρώτα να δούμε τα πεδία ορισμού και τιμών της συνάρτησής μας. Ο ΜΚΔ δεν ορίζεται όταν x == y == 0. Έχουμε λοιπόν:

ΜΚΔ: $\mathbb{N} \times \mathbb{N} \setminus \{ (0,0) \} \to \mathbb{N}$

Εδώ έχουμε δύο περιπλοκές.

- Το πεδίο ορισμού είναι καρτεσιανό γινόμενο. Τι κάνουμε στην περίπτωση αυτή; *Βάζουμε στη συνάρτησή μας δύο παραμέτρους!*
- Η συνάρτηση *gcd()* που θα γράψουμε στην C++ θα είναι μερική. Θα πρέπει να εξαιρέσουμε το (0,0).
- Τέλος, η *gcd()* θα είναι μερική και διότι, με βάση αυτά που είπαμε πιο πάνω, θα αποφύγουμε παραμέτρους `unsigned int` και θα χρησιμοποιήσουμε `int`:

```
gcd: int × int ⇸ unsigned int
```

Να η συνάρτηση:

```
unsigned int gcd( int x, int y )
{ // x == x0 && y == y0
   unsigned int  b;

   if ( (x == 0 && y == 0) || x < 0 || y < 0 )
   {
      cout << " η gcd κλήθηκε με " << x << ", " << y << endl;
      exit( EXIT_FAILURE );
   }
// (x != 0 || y != 0) && x >= 0 && y >= 0
   while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
   {
      b = y;  y = x % y;  x = b;
   } // while
```

[5] Ο Ευκλείδης αποκαλεί **ανθυφαίρεση** την πράξη εύρεσης υπολοίπου της διαίρεσης δύο φυσικών αριθμών. Για τον λόγο αυτόν η μέθοδος αυτή ονομάζεται και *ανθυφαιρετική*.

```
      return x;          // ΜΚΔ(x,0) = x
   } // gcd
```

Δες τώρα ολόκληρο το πρόγραμμα και συνεχίζουμε τη συζήτηση.

```
#include <iostream>
#include <cstdlib>
using namespace std;

   // gcd -- Υπολογίζει τον Μέγιστο Κοινό Διαιρέτη των ορισμάτων
   unsigned int gcd( int x, int y )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

   // lcm -- Υπολογίζει το Ελάχ. Κοινό Πολλαπλάσιο των ορισμάτων
   //        Χρησιμοποιεί την gcd
   unsigned int lcm( int x, int y )
   {                              // ΕΚΠ(x,y)*ΜΚΔ(x,y) = x*y
      unsigned int fv;

      if ( (x == 0 && y == 0) || x < 0 || y < 0 )
      {
         cout << " η lcm κλήθηκε με " << x << ", " << y << endl;
         exit( EXIT_FAILURE );
      }
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      fv = x * y / gcd(x, y);
      return fv;
   } // lcm

int main()
{
   int x1, x2;

   cout << " Δώσε δυο θετικούς ακέραιους: ";   cin >> x1 >> x2;
   if ( x1 <= 0 || x2 <= 0 )
       cout << " Είπα: ΘΕΤΙΚΟΥΣ" << endl;
   else
      cout << " ΕΚΠ = " << lcm( x1, x2 )
           << "  ΜΚΔ = " << gcd( x1, x2 ) << endl;
} // main
```

Σε αυτό το πρόγραμμα βλέπεις τη **main()** και άλλες δυο συναρτήσεις, από τις οποίες η δεύτερη, η *lcm()*, χρησιμοποιεί την πρώτη, την *gcd()*.

Παρατηρήσεις: ▶

1. Αν $x \neq 0$ **||** $y \neq 0$ τότε ο *θετικός ΜΚΔ* υπολογίζεται ως

$$\text{θΜΚΔ}(x, y) = \text{ΜΚΔ}(|x|, |y|)$$

Τροποποίησε τη *gcd()* ώστε να υπολογίζει τον θΜΚΔ.

2. Ναι μεν κάνουμε επίδειξη πώς καλούμε μια συνάρτηση, την *lcm()*, που καλεί μια άλλη, την *gcd()*, αλλά αν θέλουμε να υπολογίσουμε ΕΚΠ και ΜΚΔ δύο φυσικών η *gcd()* θα κάνει τους ίδιους υπολογισμούς δύο φορές. Αργότερα θα το ξανασκεφτούμε. ◀

Παράδειγμα 3

Όπως ξέρουμε, η συνάρτηση τυποθεώρησης **static_cast<int>** αν κληθεί με όρισμα πραγματικό *αποκόπτει* το κλασματικό του μέρος. Το ίδιο κάνει και η **static_cast<long int>**.

Συχνά όμως θέλουμε να *στρογγυλοποιήσουμε* (round) μια πραγματική τιμή στον πλησιέστερο ακέραιο. Ας γράψουμε λοιπόν μια συνάρτηση που θα κάνει αυτή τη δουλειά:

myRound: $\mathbb{R} \to \mathbb{Z}$

Η συνάρτηση που θα γράψουμε θα παίρνει τιμές από το **double** και θα δίνει αποτέλεσμα στον **long int**.[6] Θα μπορεί να χειριστεί οποιαδήποτε τιμή τύπου **double**; Όχι βέβαια!

[6] Στο cmath μπορείς να βρεις την επικεφαλίδα μιας τέτοιας συνάρτησης· είναι η *lround*. Δες τον πίνακα του Παραρτ. D.

Θα μπορεί να στρογγυλοποιήσει τιμές στο διάστημα

(*LONG_MIN* − ½, *LONG_MIN* + ½]

στο *LONG_MIN* αλλά όχι τιμές μικρότερες από αυτές. Παρομοίως, θα μπορεί να στρογγυλοποιήσει τιμές στο διάστημα

[*LONG_MAX* − ½, *LONG_MAX* + ½)

στο *LONG_MAX* αλλά όχι τιμές μεγαλύτερες από αυτές. Επομένως πεδίο ορισμού της συνάρτησης που θα γράψουμε θα είναι το

(*LONG_MIN* − ½, *LONG_MAX* + ½)

και η

```
myRound: double ⇸ long int
```

θα είναι μερική συνάρτηση.

Και τώρα: πώς θα κάνουμε τη στρογγυλοποίηση; Ας πούμε ότι έχουμε το 7.3, που θα πρέπει να στρογγυλοποιηθεί στο 7. Εδώ τα πράγματα είναι απλά: με τη `static_cast<long int>(7.3)` πετυχαίνουμε τον στόχο μας. Αν όμως έχουμε το 7.8; Τώρα σκέψου το εξής τέχνασμα: αν το κλασματικό μέρος είναι μεγαλύτερο από (ή ίσο με) 0.5 (π.χ. 0.8, στον 7.8) – οπότε θα πρέπει να στρογγυλοποιηθεί στον αμέσως μεγαλύτερο ακέραιο (8)– και προσθέσουμε στον αριθμό μας 0.5 αυτός θα φτάσει ή και θα ξεπεράσει τον επόμενο ακέραιο (8.3). Αν λοιπόν πάρουμε τη `static_cast<long int>(7.8 + 0.5)` μας δίνει τη σωστή τιμή (8). Πρόσεξε ότι αυτό το τέχνασμα δεν θα αλλάξει το σωστό υπολογισμό για το 7.3: `static_cast<long int>(7.3 + 0.5) == 7`. Αν η τιμή είναι *αρνητική* τότε *αφαιρούμε* 0.5:

```
         if ( x >= 0 )  fv = static_cast<long int>(x + 0.5);
                  else  fv = static_cast<long int>(x - 0.5);
```

Στη συνέχεια βλέπεις ολόκληρη τη *myRound()* σε ένα πρόγραμμα που τη δοκιμάζει.

```
#include <iostream>
#include <cstdlib>
#include <climits>
using namespace std;

   // myRound -- στρογγυλοποιεί το όρισμα στον πλησιέστερο ακέραιο
   long int myRound( double x )
   {
      long int fv;

      if ( x <= LONG_MIN - 0.5 || LONG_MAX + 0.5 <= x )
      {
         cout << " η myRound κλήθηκε με όρισμα: " << x << endl;
         exit( EXIT_FAILURE );
      }
   // LONG_MIN - 0.5 < x && x < LONG_MAX + 0.5
      if ( x >= 0 )  fv = static_cast<long int>( x + 0.5 );
               else  fv = static_cast<long int>( x - 0.5 );
      return fv;
   } // myRound

int main()
{
   double t;

   cout << " Δώσε έναν πραγματικό. 0 για τέλος: ";   cin >> t;
   while ( t != 0 )
   {
      cout << " myRound(" << t << ") = " << myRound( t ) << endl;
      cout << " Δώσε έναν πραγματικό. 0 για τέλος: ";   cin >> t;
   } // while
   cout << " myRound(" << t << ") = " << myRound( t ) << endl;
} // main
```

Η *myRound()* είναι χρήσιμη συνάρτηση και, αφού στη συνέχεια θα αποδείξεις την ορθότητά της (Ασκ. 7-18), ας δούμε πώς μπορούμε να περιγράψουμε τη σχέση μεταξύ x και *myRound*(x);

Έστω ότι $x \geq 0$. Αν το κλασματικό μέρος, *κλάσμα*(x), είναι: $0 \leq$ *κλάσμα*(x) $< ½$ τότε η x στρογγυλοποιείται με αποκοπή του κλασματικού μέρους, δηλαδή:

$$myRound(x) = x - \text{κλάσμα}(x) \quad \text{ή} \quad \text{κλάσμα}(x) = x - myRound(x)$$

Από αυτήν παίρνουμε: $0 \leq x - myRound(x) < ½$ που ισοδυναμεί με:

$$myRound(x) \leq x < myRound(x) + ½ \quad \text{ή} \quad x - ½ < myRound(x) \leq x$$

Αν $½ \leq$ *κλάσμα*(x) < 1 τότε η x στρογγυλοποιείται στον μεγαλύτερο ακέραιο: είναι σαν να προσθέτουμε στη x το $1 -$ *κλάσμα*(x). Δηλαδή:

$$myRound(x) = x + 1 - \text{κλάσμα}(x) \quad \text{ή} \quad \text{κλάσμα}(x) = x + 1 - myRound(x)$$

Από αυτήν παίρνουμε: $½ \leq x + 1 - myRound(x) < 1$ που ισοδυναμεί με:

$$myRound(x) - ½ \leq x < myRound(x) \quad \text{ή} \quad x < myRound(x) \leq x + ½$$

Αν λοιπόν $x \geq 0$ τότε:

$$myRound(x) - ½ \leq x < myRound(x) + ½ \quad \text{ή} \quad x - ½ < myRound(x) \leq x + ½$$

Με παρόμοιους συλλογισμούς μπορείς να βρεις ότι αν $x < 0$ τότε:

$$myRound(x) - ½ < x \leq myRound(x) + ½ \quad \text{ή} \quad x - ½ \leq myRound(x) < x + ½$$

Παράδειγμα 4

Πολύ συχνά χρειάζεται να στρογγυλοποιήσουμε μια πραγματική τιμή *σε n ψηφία μετά την υποδιαστολή*[7]. Θα γράψουμε λοιπόν μια συνάρτηση *dRound()* που θα κάνει αυτή τη δουλειά:

dRound: $\mathbb{R} \times \mathbb{N} \to \mathbb{R}$

Πώς γίνεται αυτή η στρογγυλοποίηση; Δες ένα παράδειγμα. Έστω ότι θέλουμε να στρογγυλοποιήσουμε το 1.347 σε $n = 2$ ψηφία μετά την υποδιαστολή (1.35). Κοίτα πώς γίνεται αυτό:

$1.347 \cdot 10^2 = 134.7$

myRound(134.7) = 135

$135 / 10^2 = 1.35$

```
// dRound -- στρογγυλοποιεί το όρισμα σε n ψηφία μετά την
//           υποδιαστολή. Χρησιμοποιεί τη myRound
double dRound( double x, int n )
{
   double tenTo( pow(10, n) );

   return myRound(x*tenTo)/tenTo;
} // dRound
```

Και είναι ολική η συνάρτησή μας; Ούτε λόγος! Κατ' αρχάς, αφού χρησιμοποιούμε τη *myRound()* θα πρέπει να έχουμε:

$$LONG\_MIN - 0.5 \leq x \cdot 10^n \leq LONG\_MAX + 0.5$$

Το n περιορίζεται από την ακρίβεια του `long int` (όχι του `double`) αλλά και από το μέγεθος του x: Αν ο `long int` έχει 10 ψηφία το πολύ και το ακέραιο μέρος του x έχει 8 ψηφία δεν μπορείς να ζητάς $n = 5$.

Θα μπορούσαμε να βάλουμε ελέγχους για τα παραπάνω, αλλά θα είναι αρκετά πολύπλοκοι· πιο πολύ θα μπερδευτείς παρά θα διδαχθείς.

Όπως βρήκαμε τη σχέση μεταξύ x, *myRound*(x) μπορείς να βρεις και τη σχέση μεταξύ x, n, *dRound*(x, n) (Άσκ. 7-19).

Πάντως η συνάρτηση είναι πολύ χρήσιμη. Πρόσεξε ότι δουλεύει και για $n \leq 0$. Για $n = 0$ δουλεύει όπως η *myRound* (στρογγυλοποιεί σε ακέραιο), για $n = -1$ στρογγυλοποιεί στο

[7] Πρόσεξε: θέλουμε την αλλαγμένη τιμή για να τη χρησιμοποιήσουμε και όχι απλώς για να την τυπώσουμε· η εκτύπωση μπορεί να γίνει με αυτά που μάθαμε στην §1.11.

ψηφίο των δεκάδων, για *n* = −2 στο ψηφίο των εκατοντάδων κ.ο.κ. Μπορείς να την ξαναγράψεις χρησιμοποιώντας την *lround()* αντι για τη *myRound()*.

Με τα παραδείγματα αυτής της παραγράφου, σου είπαμε τα περισσότερα από αυτά που πρέπει να ξέρεις για τις συναρτήσεις. Τα υπόλοιπα στην επόμενη παράγραφο.

7.7.1 *exit()* ή *assert()*

Μήπως αντί για **if** και *exit()* θα μπορούσαμε να χρησιμοποιούμε την *assert()* (§4.3); Βεβαίως! Για να δούμε τη διαφορά από αυτά που εμείς λέμε κάνουμε ένα πείραμα: Αλλάζουμε τη `main()` στο Παράδ. 1 της προηγούμενης παραγράφου:

```
int main()
{
   int m, n, comb;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
   cout << " Συνδυασμοί των "
        << m << " ανά " << n << " = " << comb << endl;
} // main
```

Να ένα παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: 2 5
 η factorial κλήθηκε με όρισμα -3
```

Αλλάζουμε τη *factorial()* ως εξής:

```
   unsigned long int factorial( int a )
   {
      unsigned long int fv;
      int k;

      assert( a >= 0 );

      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
      return fv;
   } // factorial
```

και βάζουμε στην αρχή την "`#include <cassert>`". Παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: 2 5
Assertion failed: a >= 0, file combA.cpp, line 12
```

Ποιο είναι πιο καλό;

- Για τον «άσχετο» χρήστη του προγράμματος και οι δύο αντιδράσεις είναι το ίδιο κακές· μπορεί να προκαλέσουν πανικό!
- Για τον προγραμματιστή που δοκιμάζει το πρόγραμμά του πριν το παραδώσει στον πελάτη/χρήστη η δική μας μορφή δίνει πιο πλήρη περιγραφή του προβλήματος.

Χρησιμοποίησε όποια σου αρέσει περισσότερο· de gustibus et coloribus non disputandum… Πάντως σκέψου το εξής: αν γράφεις ένα μεγάλο πρόγραμμα είναι δυνατόν να περιλάβεις μια συνάρτηση που θα σταματήσει την εκτέλεσή του επειδή ζήτησες να υπολογιστούν «οι συνδυασμοί των 2 ανά 5»; Εντάξει, αυτό δείχνει ότι κάτι δεν πάει καλά με το πρόγραμμα, αλλά δεν θα αποφασίσει η *factorial()* αν θα διακοπεί η εκτέλεσή του. Αργότερα θα μάθουμε και άλλον, πιο ευέλικτο, τρόπο για να διαχειριζόμαστε τέτοια προβλήματα.

7.8. Πώς (μετα)Γράφουμε μια Συνάρτηση

Τουλάχιστον όσοι ασχολούνται με την τεχνολογία και τις θετικές επιστήμες, έχουν συχνά να γράψουν πρόγραμμα που θα υπολογίζει τιμές κάποιας συνάρτησης. Στην εισαγωγή και στα παραδείγματα είδαμε πώς γίνεται κάτι τέτοιο. Τώρα θα δούμε τη διαδικασία μεταγραφής πιο συστηματικά.

Όπως είπαμε, τα πρώτα πράγματα που πρέπει να κάνεις είναι:

- να καθορίσεις το πεδίο ορισμού και το πεδίο τιμών της συνάρτησης,
- να σιγουρευτείς ότι έχεις το μηχανισμό που σου δίνει τις εικόνες από τα πρότυπα.

Ύστερα από αυτά, πρέπει να βρεις τους τύπους της C++ που αντιστοιχούν στα πεδία ορισμού και τιμών. Οι εύλογες αντιστοιχίες είναι $\mathbb{Z}$ και υποσύνολά του στον `int` (`long int`), $\mathbb{R}$ και υποσύνολά του (εκτός από τα υποσύνολα ακεραίων) στο `double`. Αν για παράδειγμα έχουμε:

$\| \|_2 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$

θα πάμε σε:

`EuMetro: double × double × double → double`

Τώρα πρόσεξε *μήπως η συνάρτηση που θα γράψεις δεν είναι ολική* άσχετα από το αν η αρχική είναι ή δεν είναι. Π.χ., στο παράδ. 3 της προηγούμενης παραγράφου είχαμε:
myRound: $\mathbb{R} \to \mathbb{Z}$

`myRound: double ↛ long int`

Και στο παράδ. 2:
ΜΚΔ: $\mathbb{N} \times \mathbb{N} \setminus \{ (0,0) \} \to \mathbb{N}$

`gcd: unsigned int × unsigned int ↛ unsigned int`

Και τι κάνουμε αν καταλήξουμε σε μερική συνάρτηση; Θα πρέπει να γράψουμε τη συνάρτησή μας ώστε να μπορεί να αντιμετωπίσει την περίπτωση που γίνεται κλήση με πραγματικές παραμέτρους εκτός πεδίου ορισμού. Όπως θα μάθουμε αργότερα η C++, παρέχει μηχανισμούς **διαχείρισης εξαιρέσεων** (exception handling) για τέτοιες περιπτώσεις. Προς το παρόν τι κάνουμε;

- Στα παραδείγματά μας χρησιμοποιήσαμε την *exit()* για να σταματήσουμε την εκτέλεση του προγράμματος αμέσως.
- Θα μπορούσαμε να βγάλουμε ένα μήνυμα "domain error" ή κάτι παρόμοιο και να βάλουμε στην συνάρτηση μια απίθανη τιμή (αν υπάρχει) που θα μπορεί να ανιχνευθεί μετά την κλήση.
- Θα μπορούσαμε να βγάλουμε ένα μήνυμα "domain error" ή κάτι παρόμοιο και να μην κάνουμε οτιδήποτε. Η τιμή που θα επιστρέψει η συνάρτηση στην παράσταση, από όπου έγινε η αναφορά, θα είναι τυχαία και θα οδηγήσει σε παράλογο (;) αποτέλεσμα.

Εμείς θα επιμείνουμε στην πρώτη λύση, αφού ένα πρόγραμμα που ζητάει τον υπολογισμό συνάρτησης με παραμέτρους εκτός πεδίου ορισμού έχει σίγουρα λάθος.

Ας δούμε τώρα άλλη μια συνηθισμένη περιπλοκή: η συνάρτησή μας, f, είναι περιοδική, με περίοδο T, δηλ. $f(x+kT) = f(x)$ για ακέραιες τιμές του k. Στην περίπτωση αυτή θα πρέπει να έχουμε ένα (πρωτεύον) διάστημα, ας πούμε το $[a, b)$, με μήκος μια περίοδο ($b = a + T$), όπου να έχουμε μηχανισμό που μας δίνει τις εικόνες από τα πρότυπα. Το πρώτο πράγμα που κάνουμε στην περίπτωση αυτή είναι αναγωγή της τιμής της παραμέτρου (x) σε κάποιο x_0, στο πρωτεύον διάστημα, για το οποίο $f(x_0) = f(x)$. Αυτό μπορεί να γίνει με τις εντολές:

```
   x0 = x;
   while ( x0 >= b )  x0 = x0 - T;
// (x0 < b) && (f(x0) == f(x))
   while ( x0 < a )   x0 = x0 + T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Πρόσεξε ότι η $f(x_0)$ == $f(x)$ είναι αναλλοίωτη και για τις δύο **while**, αφού οι διαδοχικές τιμές που μπορεί να πάρει η x_0 διαφέρουν από την τιμή της x κατά ακέραιο πλήθος περιόδων.

Πάντως η αναγωγή μπορεί να γίνει και πιο γρήγορα. Η C++ μας δίνει τη συνάρτηση (γνωστή από τα μαθηματικά, όπου τη γράφουμε συνήθως ως $\lfloor x \rfloor$) *floor*: `double` → `double`, τέτοια ώστε *floor*(x) να είναι ο μέγιστος *ακέραιος* που είναι μικρότερος ή το πολύ ίσος με x. Με τη βοήθειά της μπορούμε να βρούμε το x_0 πιο γρήγορα (Ασκ. 7-11):

```
   m = floor((x-a)/T);
   x0 = x - m*T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Παρατηρήσεις: ▶

1. Να επαναλάβουμε αυτό που είπαμε και πιο πάνω: Αν έχεις παραμέτρους τύπου `double` (ή `float` ή `long double`) και θέλεις να ελέγξεις συγκεκριμένες τιμές, συγκρίσεις όπως "`x0 == v`" δεν έχουν νόημα. Θα πρέπει να ελέγχεις με κάτι σαν: "`fabs(x0 - v) <= eps`".
2. Αν η συνάρτησή σου είναι περιοδική και έχεις όρισμα πολύ μεγαλύτερο από την περίοδο τότε, κατά την αναγωγή, η x `- m*T` μπορεί να σου δώσει 0 (μηδέν).
3. Αν η συνάρτησή σου είναι περιοδική και το διάστημα ορισμού του μηχανισμού είναι της μορφής $(a, b]$ έχεις το εξής πρόβλημα: η μέθοδος αναγωγής που δώσαμε σου εγγυάται ότι $a \le x_0 < b$. Αν έχεις a == x_0 θα πρέπει να προσθέσεις μια περίοδο για να πας στο b[8]:

```
   m = floor( (x-a)/T );
   x0 = x - m*T;
// (a ≤ x0 < b) && (f(x0) == f(x))
   if (x0 <= a)  x0 = x0 + T;
// (a < x0 ≤ b) && (f(x0) == f(x)) ◀
```

Τα παραπάνω συνοψίζονται σε μια «συνταγή» που τη βλέπεις στο Πλ. 7.4. Στη συνέχεια βλέπεις ένα παράδειγμα εφαρμογής της συνταγής.

Παράδειγμα 🡲

Μια συνάρτηση ορίζεται μαθηματικά ως εξής:

$$v(x) = -\frac{1}{|x+2|} - \frac{1}{|x|} - \frac{1}{|x-2|} \quad \text{για } x \in [-1,1)$$

Η συνάρτηση είναι περιοδική με περίοδο $T = 2$. Να γίνει η υλοποίησή της στη C++.

1. Η συνάρτησή μας ορίζεται σε όλα τα σημεία του $[-1,1)$ εκτός από το 0 (μηδέν). Αφού η συνάρτηση είναι περιοδική με περίοδο 2, η συνάρτηση ορίζεται σε ολόκληρο το $\mathbb{R}$ εκτός από τα σημεία $2k$ όπου k ακέραιος. Άρα:

   v: $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \} \to \mathbb{R}$

δηλαδή: πεδίο ορισμού είναι το $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \}$ και πεδίο τιμών είναι το $\mathbb{R}$.

2. Ο μηχανισμός που μας δίνει τις εικόνες από τα πρότυπα δίνεται με ακρίβεια.
3. Στο σύνολο $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \}$ όπως και στο $\mathbb{R}$ αντιστοιχεί ο `double`.
4. Επικεφαλίδα:

```
double v( double x )
```

5. Ενώ, όπως είδαμε παραπάνω, η v στα μαθηματικά είναι ολική, στη C++ θα έχουμε:

```
v: double ⇸ double
```

που φυσικά *δεν είναι* ολική.

6. Η αναγωγή της παραμέτρου στο διάστημα: $[-1,1)$ μπορεί να γίνει με τις εντολές:

```
   m = floor( (x-(-1))/T );
   x0 = x - m*T;
// (-1 ≤ x0 < 1) && (v(x0) == v(x))
```

[8] Στην περίπτωση αυτή μπορείς να κάνεις την αναγωγή με τη «δίδυμη» της *floor*, τη *ceil*. Δες την Άσκ. 7-12.

Πλαίσιο 7.4

Πώς (μετα)γράφουμε μια Συνάρτηση στη C++

Για να (μετα)γράψεις μια μαθηματική συνάρτηση σε C++ κάνε τα εξής:

1. Καθόρισε με ακρίβεια το πεδίο ορισμού (κυρίως) και το σύνολο τιμών.

2. Καθόρισε με ακρίβεια το μηχανισμό που μας δίνει τις εικόνες ($f(x)$) από τα πρότυπα (x).

3. Καθόρισε προσεκτικά τους τύπους της C++ που αντιστοιχούν στο πεδίο ορισμού και στο σύνολο τιμών. Για αριθμητικές συναρτήσεις οι αντιστοιχίες είναι:

- $\mathbb{N}$, $\mathbb{N}^*$ ή υποσύνολά τους στον `int` ή τον `long int`, για το πεδίο ορισμού, στον `unsigned int` ή τον `unsigned long int`, για το πεδίο τιμών.
- $\mathbb{Z}$, $\mathbb{Z}^*$ ή υποσύνολά τους στον `int` ή τον `long int`.
- $\mathbb{R}$ ή υποσύνολο του $\mathbb{R}$ στον `double` ή τον `long double` ή τον `float`,

4. Τώρα μπορείς να γράψεις την επικεφαλίδα της συνάρτησης και να δηλώσεις τους τύπους των ορισμάτων.

5. Μετά το βήμα 3, μπορείς να αποφανθείς κατά πόσον η συνάρτηση που θα γράψεις είναι *ολική* (ορίζεται σε ολόκληρο το σύνολο αφετηρίας) ή *μερική*.

6. Αν η συνάρτηση είναι περιοδική, γράψε τις εντολές που βρίσκουν τιμή (x_0) στο διάστημα ορισμού, ισοδύναμη με την αρχική (x), δηλαδή τέτοια ώστε: $x_0 \in$ διάστημα ορισμού και $f(x_0) = f(x)$.

7. Αν η συνάρτηση είναι μερική, γράψε τις εντολές που εξαιρούν τις τιμές του ορίσματος για τις οποίες δεν ορίζεται η συνάρτηση:

```
if ( x δεν ανήκει στο πεδίο ορισμού )
{
   cout << " η ... κλήθηκε με x = " << x << endl;
   exit( EXIT_FAILURE );
}
else
   Υπολόγισε την τιμή της συνάρτησης
```

8. Μετάφρασε σε C++ το μηχανισμό που καθόρισες στο βήμα 2.

7. Η συνάρτησή μας είναι μερική. Μετά την αναγωγή της x στη x_0 στο $[-1,1)$, όλα τα σημεία στα οποία δεν ορίζεται η V ανάγονται στο 0. Θα πρέπει να εξαιρέσουμε αυτήν την τιμή και μόνο:

```
if ( x0 == 0 )
{
   cout << " η v κλήθηκε με όρισμα: " << x << endl;
   exit( EXIT_FAILURE );
}
```

Πρόσεξε ότι στο μήνυμα δεν βάζουμε τη *x0* αλλά τη *x*. Ακόμη, πρόσεξε ότι η σύγκριση "`x0 == 0`" δεν έχει και πολύ νόημα. Πιο σωστό θα ήταν κάτι σαν "`fabs(x0) < eps`", όπου *eps* μια μικρή τιμή που εξαρτάται από τον τύπο `double` και το πρόβλημά σου.

8. Η μετάφραση του μηχανισμού είναι τετριμμένη:

```
fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
```

Να ολόκληρη η συνάρτηση:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double x0( x ), fv, m;
```

```
      m = floor( (x-a)/T );
      x0 = x - m*T;
   // (-1 ≤ x0 < 1) && (v(x0) == v(x))
      if ( x0 == 0 )
      {
         cerr << " η v κλήθηκε με όρισμα: " << x << endl;
         exit(EXIT_FAILURE);
      }
   // (-1 ≤ x0 < 1) && (v(x0) == v(x))
      fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
      return fv;
   } // v
```

7.9. * Οι Συναρτήσεις στις Αποδείξεις

Όπως είδαμε και πιο πριν, στις παραστάσεις που είχαμε *sqrt*(*x*) βάζαμε (κάπως αισιόδοξα) $\sqrt{x}$, για την οποία τα μαθηματικά μας δίνουν πολύ συγκεκριμένες προδιαγραφές. Ζητούσαμε όμως να έχουμε $x \geq 0$.

Κάπως έτσι θα χειριζόμαστε και τις δικές μας συναρτήσεις. Βάζουμε προϋπόθεση στις παραμέτρους και απαίτηση στην τιμή που επιστρέφει:

```
   Tf f(T1 p1, T2 p2,... Tn pn)
   {
      Tf fv;
       :
   // Pd(p1, p2,... pn)
        :
   // Qd(p1, p2,... pn, fv)
      return fv;
   } // f
```

Η προϋπόθεση στις παραμέτρους (ή τουλάχιστον ένα μέρος της) μας είναι ήδη γνωστή: *θα πρέπει να ανήκουν στο πεδίο ορισμού της συνάρτησης*. Όπως κάνουμε στα προγράμματά μας, αλλά όπως κάναμε και στα παραδείγματα, παραπάνω, θα πρέπει και εδώ να ελέγχουμε την προϋπόθεση.

Ύστερα από αυτό, κάθε φορά που έχουμε κάποια κλήση της *f*, ας πούμε *f*(*t*1, *t*2,... *tn*), θα πρέπει να σιγουρεύουμε ότι ισχύει η *Pd*(*t*1, *t*2,... *tn*). Για την παράσταση στην οποία υπάρχει η *f*(*t*1, *t*2,... *tn*) θα έχουμε *Qd*(*t*1, *t*2,... *tn*, *f*(*t*1, *t*2,... *tn*)).

Προσοχή σε ένα σημείο: οι τιμές των παραμέτρων μπορεί να αλλάζουν μέσα στη συνάρτηση· αλλά στην *Qd*(*p*1, *p*2,... *pn*, *fv*) θεωρούμε ότι οι *p*1, *p*2,... *pn* έχουν τις τιμές που είχαν αρχικά. Ένας τρόπος για να μην μπλεχτείς είναι να αντιγράψεις τις τιμές των παραμέτρων σε τοπικές μεταβλητές. Δες και το

Παράδειγμα

Θα αποδείξουμε ότι η συνάρτηση που γράψαμε για τον υπολογισμό του ΜΚΔ δύο φυσικών αριθμών (την παραθέτουμε στη συνέχεια, κάπως αλλαγμένη, για ευκολία) είναι σωστή. Θα στηριχθούμε στο θεώρημα της Αριθμοθεωρίας που λέει:

(Π1) *Αν οι* $|\alpha| + |\beta| > 0$ *(αν δηλαδή δεν είναι και οι δύο μηδέν) τότε ΜΚΔ*$(\alpha,\beta) = ΜΚΔ(\beta,\upsilon)$, *όπου υ το υπόλοιπο της διαίρεσης α:β*.

και στο προφανές:

(Π2) *Αν* $\alpha \neq 0$ *τότε ΜΚΔ*$(0,\alpha) = \alpha$.

```
   unsigned int gcd( int x, int y )
   { // x == x0 && y == y0
      unsigned int  b;

      if ( (x == 0 && y == 0) || x < 0 || y < 0 )
```

```
      {
         cout << " η gcd κλήθηκε με " << x << ", " << y << endl;
         exit( EXIT_FAILURE );
      }
   // (x == x0 && y == y0) &&
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
      {
         b = y;  y = x % y;  x = b;
      } // while
      return x;         // ΜΚΔ(x,0) = x
   } // gcd
```

Τι έχουμε να αποδείξουμε εδώ; Το εξής:

```
   // (x == x0 && y == y0) &&
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
      {
         b = y;  y = x % y;  x = b;
      } // while
   // x == ΜΚΔ( x0, y0 )
```

δηλαδή:

- $Pd(x_0, y_0)$ είναι η: (x_0 >= 0) && (y_0 >= 0) && (!(x_0 == 0 && y_0 == 0))
- $Qd(x_0, y_0, x)$ είναι η: x == $ΜΚΔ(x_0, y_0)$

Και με την **if** δεν θα ασχοληθούμε; Η **if** υπάρχει για να ελέγξει την προϋπόθεση ή αλλιώς: αν οι παράμετροι έχουν τιμές μέσα στο πεδίο ορισμού. Αν η συνθήκη της **if** ισχύει η εκτέλεση της συνάρτησης (και του προγράμματος) δεν θα ολοκληρωθεί.

Ας έρθουμε τώρα στην απόδειξη: Αν η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) είναι αναλλοίωτη τότε μετά τη **while** θα έχουμε:

!(y != 0) && (ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0))

ή αλλιώς:

(y == 0) && (ΜΚΔ(x,y) == ΜΚΔ(x_0, y_0))

από την οποία παίρνουμε ΜΚΔ(x,0) == ΜΚΔ(x_0, y_0) και με βάση την (Π2): x == ΜΚΔ(x_0, y_0).

Φτάνει λοιπόν να αποδείξουμε ότι

1. η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) ισχύει πριν από τη **while** και
2. παραμένει αναλλοίωτη από τις επαναλαμβανόμενες εντολές:

```
      // (y != 0) && (ΜΚΔ(x,y) == ΜΚΔ(x0,y0))
         b = y;  y = x % y;  x = b;
      // ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
```

Η 1. είναι προφανής: η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) συνάγεται από την x == x_0 && y == y_0. Και η (x_0 >= 0) && (y_0 >= 0) && (!(x_0 == 0 && y_0 == 0)) δεν μας χρειάζεται; Χρειάζονται για να ορίζεται ο ΜΚΔ(x_0,y_0).

Στη συνέχεια βλέπεις την απόδειξη της 2.:

```
      // ΜΚΔ(y, x % y) == ΜΚΔ(x0,y0)
         b = y;
      // ΜΚΔ(b, x % y) == ΜΚΔ(x0,y0)
         y = x % y;
      // ΜΚΔ(b,y) == ΜΚΔ(x0,y0)
         x = b;
      // ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
```

Συνάγεται η ΜΚΔ(y, x % y) == ΜΚΔ(x_0,y_0) από την προϋπόθεση; Ναι, διότι η προϋπόθεση μας δίνει: ΜΚΔ(x, y) == ΜΚΔ(x_0,y_0) και από το (Π1) έχουμε: ΜΚΔ(x, y) == ΜΚΔ(y, x % y). Από αυτές τις δύο και τη μεταβατικότητα της ισότητας παίρνουμε: ΜΚΔ(y, x % y) == ΜΚΔ(x_0,y_0). Άρα η συνάρτησή μας είναι σωστή.

Αν λοιπόν γράψουμε σε μια παράσταση, στο πρόγραμμά μας, *gcd*(Π_1, Π_2), όπου Π_1 και Π_2 παραστάσεις με τιμές τύπου **unsigned int**, από τις οποίες η μια τουλάχιστον δεν είναι μηδέν, τότε, μετά την εκτέλεση της *gcd* θα έχουμε *gcd*(Π_1,Π_2) **==** ΜΚΔ(Π_1,Π_2).

7.10. Αναδρομή

Ήταν κάποιος που δεν ήξερε καμιά ιστορία
κι όλο έλεγε «ξέρω πολλές ιστορίες· μια απ' αυτές λέει πως
Ήταν κάποιος που δεν ήξερε καμιά ιστορία
κι όλο έλεγε "ξέρω πολλές ιστορίες· μια απ' αυτές λέει πως
Ήταν κάποιος... "»
Γ. Αγγελάκας, Υπέροχο Τίποτα

Στην C++ υπάρχει η δυνατότητα αναδρομικής διατύπωσης μιας συνάρτησης, όπως και στα μαθηματικά υπάρχει η δυνατότητα αναδρομικής διατύπωσης μερικών ορισμών. Ένας πολύ γνωστός αναδρομικός ορισμός είναι αυτός για το *n*!, που τον επαναλαμβάνουμε:

$$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n \geq 1 \end{cases}$$

Ο παραπάνω αναδρομικός ορισμός μπορεί να μεταφρασθεί εύκολα στη C++ σε μια **αναδρομική συνάρτηση** (recursive function) με τον ακόλουθο τρόπο:

```
// rFactorial -- Υπολογίζει το a! με αναδρομική κλήση
unsigned long int rFactorial( int a )
{
   unsigned int k, fv;

   if ( a < 0 )
   {
      cout << "η rFactorial κλήθηκε με όρισμα " << a << endl;
      exit( EXIT_FAILURE );
   }
   if ( a == 0 )  fv = 1;
            else  fv = a*rFactorial( a-1 );
   return fv;
} // rFactorial
```

Βλέπουμε λοιπόν ότι η συνάρτηση *rFactorial()* καλεί τον εαυτό της μέσα στο τμήμα των εντολών της. Ο παραπάνω τρόπος διατύπωσης της λειτουργίας μιας συνάρτησης, όπου μέσα στο σώμα του υποπρογράμματος εμφανίζεται αναφορά στην ίδια τη συνάρτηση λέγεται *αναδρομικός τρόπος* διατύπωσης του αλγορίθμου, ενώ η ίδια η συνάρτηση ονομάζεται *αναδρομική*. Συγκρίνοντας τον παραπάνω αναδρομικό τρόπο διατύπωσης του αλγορίθμου με τον ισοδύναμο επαναληπτικό τρόπο, που είδαμε σε προηγούμενες παραγράφους, μπορούμε να διαπιστώσουμε ότι ο αναδρομικός τρόπος είναι πιό απλός, πιό σύντομος και πλησιέστερος προς τον μαθηματικό ορισμό.

Συχνά όμως ο αναδρομικός τρόπος είναι λιγότερο αποδοτικός από τον επαναληπτικό και σε χρόνο και σε μνήμη. Γι' αυτό πρέπει να χρησιμοποιείται με προσοχή. Μη βιάζεσαι λοιπόν να χρησιμοποιήσεις αυτόν τον τρόπο παρ' όλη την κομψότητα διατύπωσης που προσφέρει. Άφησέ τον για αργότερα, που θα έχεις μεγαλύτερη ωριμότητα και θα έχεις μάθει μερικά πράγματα παραπάνω.

Προς το παρόν δες άλλο ένα παράδειγμα: η συνάρτηση για το ΜΚΔ γραμμένη με αναδρομή.

```
// rGcd -- Υπολογίζει τον Μέγιστο Κοινό Διαιρέτη των ορισμάτων
//         της με αναδρομική κλήση
unsigned int rGcd( int x, int y )
{
   unsigned int  fv;
```

```
    if ( x < 0 || y < 0 || (x == 0 && y == 0) )
    {
       cout << " η rGcd κλήθηκε με " << x << ',' << y << endl;
       exit( EXIT_FAILURE );
    }
    if ( y == 0 ) fv = x;                       // ΜΚΔ(x,0) = x
             else fv = rGcd(y, x % y);// ΜΚΔ(x,y) = ΜΚΔ(y,x % y)
    return fv;
} // rGcd
```

7.11. Ανακεφαλαίωση

Αν η C++ δεν έχει έτοιμη κάποια συνάρτηση (με τύπο) που χρειάζεσαι στο πρόγραμμά σου –πράγμα πολύ πιθανό– θα πρέπει να μπορείς να τη γράψεις.

Μια συνάρτηση

- Αρχίζει με μια επικεφαλίδα όπου καθορίζεται *ο τύπος της τιμής που επιστρέφει* (αντιστοιχεί στο πεδίο τιμών), *το όνομά της* και *οι τυπικές παράμετροί της με τους τύπους τους* (αντιστοιχούν στο σύνολο αφετηρίας).
- Ακολουθεί το σώμα της συνάρτησης, δηλαδή είναι ένα κομμάτι προγράμματος που περιγράφει το πώς υπολογίζεται η τιμή της συνάρτησης από τις τιμές των παραμέτρων.
- Μέσα στη συνάρτηση (επικεφαλίδα και σώμα) μπορεί να δηλώνονται *τοπικές* μεταβλητές, σταθερές κλπ που είναι γνωστές μόνον μέσα στη συνάρτηση και ζουν όσο διαρκεί η εκτέλεσή της.
- Στο σώμα της συνάρτησης υπάρχει μια τουλάχιστον εντολή **return** που τερματίζει την εκτέλεση της συνάρτησης και αντικαθιστά την κλήση της συνάρτησης με την τιμή που υπολόγισε.

Παρ' όλο που στο σώμα μιας συνάρτησης μπορεί να υπάρχουν πολλές εντολές **return**, *είναι προτιμότερο να υπάρχει μια μόνον, στο τέλος*.

Μια συνάρτηση *καλείται με το όνομά της* και τις πραγματικές παραμέτρους που θα δώσουν τις τιμές τους στις αντίστοιχες τυπικές.

Με το να «κρύβεις» κάποιους υπολογισμούς του προγράμματος μέσα σε συναρτήσεις αυξάνεις την ευκολία επαλήθευσης, διόρθωσης, τροποποίησης, και γενικώς διαχείρισής του.

Ασκήσεις

Α Ομάδα

7-1 Γράψε συνάρτηση, που θα επιστρέφει ως τιμή, την τιμή της μέγιστης των παραμέτρων του *a*, *b*, *c* και *d* (πραγματικοί αριθμοί).

7-2 α)Ένας **ημιανορθωτής τάσης** είναι ηλεκτρονική διάταξη με μια είσοδο και μια έξοδο. Αν στην είσοδο βάλουμε μια τάση θετική τότε στην έξοδο παίρνουμε την τάση εισόδου· αλλιώς (μη θετική τάση στην είσοδο) στην έξοδο παίρνουμε 0 (μηδέν). Γράψε συνάρτηση που να προσομοιώνει τη λειτουργία του ημιανορθωτή.
β) Ένας **ανορθωτής τάσης** (χωρίς εξομάλυνση) είναι ηλεκτρονική διάταξη με μια είσοδο και μια έξοδο. Αν στην είσοδο βάλουμε μια τάση θετική τότε στην έξοδο παίρνουμε την τάση εισόδου· αλλιώς (μη θετική τάση στην είσοδο) στην έξοδο παίρνουμε την αντίθετη της τάσης εισόδου. Γράψε συνάρτηση που να προσομοιώνει τη λειτουργία του ανορθωτή.

7-3 (Σύνθεση των ασκ. 2-9 και 5-5) Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 2%, επί του βασικού μισθού, για κάθε χρόνο υπηρεσίας. Ακόμη, ο μισθός

ενός εργαζόμενου προσαυξάνεται κατά 30 € για κάθε παιδί, αν έχει μέχρι τρία (3) παιδιά. Αν έχει περισσότερα από 3 παιδιά η προσαύξηση είναι 50 € για κάθε παιδί.

Γράψε μια:

```
double salary( double base, int years, int noOfCldr )
```

που θα επιστρέφει το συνολικό μισθό. Σκέψου τις πιθανές κακοτοπιές, π.χ.: αρνητικές τιμές παραμέτρων, παράλογα μεγάλες τιμές παραμέτρων κλπ. Αν σου χρειαστούν άλλοι τύποι στοιχείων, όρισέ τους και άλλαξε την παραπάνω επικεφαλίδα.

7-4 Με βάση το πρόγραμμα που έγραψες για την Άσκ. 5-6, γράψε μια:

```
double resistors( char mode, double r1, double r2 )
```

που θα επιστρέφει την τιμή της αντίστασης που προκύπτει αν οι *r*1, *r*2 συνδεθούν εν σειρά (*mode* == 'S') ή παράλληλα (*mode* == 'P').

7-5 Με βάση το πρόγραμμα για τους λογαριασμούς της ΔΕΗ της §5.3, γράψε μια:

```
double elEnergyCost( double cons )
```

που θα επιστρέφει το κόστος της κατανάλωσης. Ξαναγράψε το πρόγραμμα που έγραψες για την άσκ. 6-3 με χρήση της συνάρτησης.

Β Ομάδα

7-6 Γράψε μια:

```
double dTrunc( double x, int n )
```

που θα μας επιστρέφει την *x*, αφού αποκόψει όλα τα ψηφία της μετά το *n*-οστό ψηφίο μετά την υποδιαστολή.

Υπόδ.: Δες πώς γράψαμε τη dRound θυμίσου και αυτά που λέγαμε στην §2.8.

7-7 Έχοντας λύσει την Άσκ. 4-15, δεν θα έχεις πρόβλημα να γράψεις τις παρακάτω συναρτήσεις που συμπληρώνουν αυτές που μας δίνει η C++ για το διαχωρισμό των χαρακτήρων:

```
// isGAlpha -- Επιστρέφει τιμή true αν ο ch είναι γράμμα
//             του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGAlpha( char ch )
// isGUpper -- Επιστρέφει τιμή true αν ο ch είναι κεφαλαίο
//             γράμμα του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGUpper( char ch )
// isGLower -- Επιστρέφει τιμή true αν ο ch είναι μικρό
//             γράμμα του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGLower( char ch )
```

Και τώρα ξαναγράψε το πρόγραμμα που έγραψες για την Άσκ. 4-15, με χρήση αυτών των συναρτήσεων.

7-8 Γράψε τις παρακάτω συναρτήσεις:

```
// upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
//           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
char upCase( char ch )
// loCase -- Αν ο ch είναι κεφαλαίο Λατινικό γράμμα
//           επιστρέφει το αντίστοιχο μικρό, αλλιώς τον ch
char loCase( unsigned char ch )
```

καθώς και τις αντίστοιχες για τα ελληνικά: *gUpCase()*, *gLoCase()*.

Γ Ομάδα

7-9 Με βάση αυτά που είδες στο παράδ. 2 της §6.3 γράψε μια

```
double nrsqrt( double a )
```

που θα μας δίνει την τετραγωνική ρίζα του *a*. Γράψε πρόγραμμα που θα την δοκιμάζει και θα τη συγκρίνει με τη *sqrt* της C++.

7-10 Αν δούμε την κυβική ρίζα ενός πραγματικού, *a*, ως λύση της εξίσωσης $x^3 - a = 0$, μπορούμε να την προσεγγίσουμε με τη μέθοδο *Newton-Raphson*, $x_n = x_{n-1} - f'(x_{n-1})/f(x_{n-1})$, όπου, για την περίπτωσή μας:

$$f(x) = x^3 - a, \qquad f'(x) = 3x^2$$

Θα έχουμε λοιπόν:

$$x_n = \frac{1}{3}\left(2x_{n-1} + \frac{a}{x_{n-1}^2}\right)$$

Μπορούμε να ξεκινήσουμε με $x_0 = a$ και να υπολογίζουμε όρους της ακολουθίας μέχρι να πάρουμε $|x_n^3 - a| < \varepsilon_{\texttt{double}} a$. Πρόσεξε, ότι για `a == 0` έχουμε πρόβλημα, αλλά για την περίπτωση αυτή ξέρουμε τη λύση. Γράψε μια συνάρτηση

```
double cbrt( double a )
```

που υπολογίζει την κυβική ρίζα. Γράψε πρόγραμμα που θα την δοκιμάζει για διάφορες τιμές (*x*) συγκρίνοντας το *cbrt*(*x*) με το *pow*(*x*, 1/3.0) της C++.

***7-11** Αν η *v()* είναι περιοδική συνάρτηση με περίοδο *T* και $b-a = T$ τότε:

α) απόδειξε ότι:

```
// true
   x0 = x;
   while ( x0 >= b )  x0 = x0 - T; // αναλλοίωτη: v(x0) == v(x)
   while ( x0 < a )   x0 = x0 + T; // αναλλοίωτη: v(x0) == v(x)
// (a ≤ x0 < b) && (v(x0) == v(x))
```

β) απόδειξε ότι:

```
// true
   m = floor( (x-a)/T );
   x0 = x - m*T;
// (a ≤ x0 < b) && (v(x0) == v(x))
```

Για τη *floor()* ισχύουν τα εξής: $floor(x) \in \mathbb{Z}$ και $x - 1 < floor(x) \le x$.

***7-12** Αν η *v* είναι περιοδική συνάρτηση με περίοδο *T* και $b-a = T$ τότε απόδειξε ότι:

```
// true
   m = ceil( (x-a)/T );
   x0 = x - m*T;
// (a < x0 ≤ b) && (v(x0) == v(x))
```

Για τη *ceil()* ισχύουν τα εξής: $ceil(x) \in \mathbb{Z}$ και $x \le ceil(x) < x + 1$.

7-13 Γράψε συνάρτηση με μια παράμετρο *k*, που θα επιστρέφει ως τιμή τον *k*-οστό όρο της ακολουθίας, που καθορίζεται από τον τύπο:

$a_0 = 0,\ a_{k+1} = a_k k + k,$ για $k \in \mathbb{N}^*$

Δώσε μια μη αναδρομική και μια αναδρομική λύση στο πρόβλημα.

***7-14** *Η ακολουθία Fibonacci* ορίζεται ως εξής: $f_0 = 0$, $f_1 = 1$ και $f_k = f_{k-2} + f_{k-1}$ για $k > 1$. Γράψε μια *αναδρομική*:

```
int rFib( int k )
```

που θα επιστρέφει ως τιμή το f_k. Γράψε και μια μη αναδρομική συνάρτηση `int fib( int k )` που να κάνει την ίδια δουλειά.

7-15 Γράψε συνάρτηση

```
double mp( double x, int n )
```

που να υλοποιεί την παρακάτω συνάρτηση:

$$m_p(x,n) = \prod_{k=0}^{n-1} \frac{1}{x-k} = \frac{1}{x-0} \times \frac{1}{x-1} \times \ldots \times \frac{1}{x-(n-1)},$$ όπου *n* φυσικός ≥ 1.

***7-16** Γράψε αναδρομική λύση της προηγούμενης άσκησης.

7-17 Μια συνάρτηση ορίζεται μαθηματικά ως εξής:

$$q(x,n)=\begin{cases}-1, & -n<x\le -1\\ x, & -1<x\le 1\\ 1, & 1<x\le n\end{cases}$$

όπου n φυσικός > 1. Η συνάρτηση είναι περιοδική στη x με περίοδο $2n$. Να γραφεί συνάρτηση που την υλοποιεί σε C++.

***7-18** Απόδειξε ότι η *myRound()* (§7.7, παράδ. 3) είναι σωστή, δηλαδή:

```
// true
   if (x >= 0)  fv = (long int) (x + 0.5);
          else  fv = (long int) (x - 0.5);
// (x ≥ 0 && x - ½ < myRound(x) ≤ x + ½) ||
// (x < 0 && x - ½ ≤ myRound(x) < x + ½)
```

Υπόδ.: Έλυσες την άσκ. 3-6; Αν δεν την έλυσες δε γίνεται τίποτε…

***7-19** Απόδειξε ότι για τη *dRound()* (§7.7, παράδ. 4) έχουμε:

Αν $x \ge 0$ τότε

$x - 0.5\cdot10^{-n} < dRound(x, n) \le x + 0.5\cdot10^{-n}$ και

$dRound(x, n) - 0.5\cdot10^{-n} \le x < dRound(x, n) + 0.5\cdot10^{-n}$

Αν $x < 0$ τότε

$x - 0.5\cdot10^{-n} \le dRound(x, n) < x + 0.5\cdot10^{-n}$ και

$dRound(x, n) - 0.5\cdot10^{-n} < x \le dRound(x, n) + 0.5\cdot10^{-n}$

κεφάλαιο

8

Αρχεία I – Text

Ο στόχος μας σε αυτό το κεφάλαιο:
Να μάθεις να διαχειρίζεσαι τη βοηθητική μνήμη με εργαλείο το *σειριακό μορφοποιημένο αρχείο* ή *αρχείο-κείμενο*.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να γράφεις προγράμματα που θα φυλάγουν δεδομένα σε αρχείο ή αρχεία ώστε να τα χρησιμοποιήσεις αργότερα με άλλα προγράμματα. Με λίγη δουλειά παραπάνω θα μπορείς να επεξεργαστείς και αρχεία text που βγάζουν διάφορα «πακέτα» (έτοιμα προγράμματα) ή να ετοιμάσεις στοιχεία εισόδου για τέτοια «πακέτα».

Έννοιες κλειδιά:

- *σειριακό αρχείο*
- *μορφοποιημένο αρχείο ή αρχείο-κείμενο* (*text*)
- *μη-μορφοποιημένο ή δυαδικό* (*binary*) *αρχείο*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Στις συνηθισμένες εφαρμογές των ΗΥ, τα στοιχεία που έχει να επεξεργαστεί ένα πρόγραμμα είναι τόσο πολλά, που δεν είναι δυνατό να χωρέσουν στην κύρια μνήμη του ΗΥ, όλα μαζί. Ακόμη, τα στοιχεία αυτά δεν πρέπει να χάνονται όταν ο ΗΥ δεν λειτουργεί,

πράγμα που δεν συμβαίνει με τα στοιχεία που υπάρχουν στην κύρια μνήμη. Για το λόγο αυτόν, τα στοιχεία αποθηκεύονται σε βοηθητικές μονάδες μνήμης –δίσκους, ταινίες κλπ– και έχουν μια λογική οργάνωση ώστε να μπορούν να χρησιμοποιηθούν εύκολα από τα προγράμματα.

Ο πιο απλός τρόπος οργάνωσης είναι το **σειριακό αρχείο** (serial ή sequential file).

Μπορούμε να φανταστούμε το σειριακό αρχείο σαν μια πεπερασμένη ακολουθία από όμοιες –δηλ. του ίδιου τύπου– τιμές[1]. Ονομάζουμε **μήκος** του αρχείου τον αριθμό των στοιχείων που έχει. Το μήκος του *κενού αρχείου* είναι 0.

Σημείωση:▶

Εκτός από *μήκος* υπάρχει και το **μέγεθος** (size) του αρχείου που είναι ο αριθμός των ψηφιολέξεων που καταλαμβάνει. Για τα αρχεία που θα δούμε σε αυτό το κεφάλαιο –*αρχεία χαρακτήρων*– έχουμε *μήκος* == *μέγεθος*.◀

Παράδειγμα

Το:

```
< -7, -15, 0, 14, 33, -8, 16, 114, 375 >
```

είναι ένα αρχείο με στοιχεία τύπου `int` και μήκος 9.

Το:

```
<'m', 'a', 'i', 'n', '(', ')', '{', '}'>
```

είναι ένα αρχείο με στοιχεία τύπου `char` και μήκος 8 (είναι ένα πρόγραμμα C++).

Το:

```
<true, true, false, true, false, false, true>
```

είναι ένα αρχείο με στοιχεία τύπου `bool` και μήκος 7.

Πώς μπορούμε να «δούμε» το περιεχόμενο ενός αρχείου; Αν είναι ένα αρχείο σαν το δεύτερο του παραδείγματος ξέρεις ήδη την απάντηση: το παίρνουμε σε κάποιον κειμενογράφο και το βλέπουμε. Αυτό είναι ένα παράδειγμα **κειμένου** (text) ή **μορφοποιημένου αρχείου** (formatted file)[2]. Υπάρχει όμως και μια άλλη κατηγορία αρχείων: αυτά που τα στοιχεία τους είναι αντίγραφα των εσωτερικών παραστάσεων του υπολογιστή. Αν, ας πούμε, το πρώτο παράδειγμα είναι γραμμένο έτσι, μπορείς να το πάρεις στον κειμενογράφο αλλά δεν θα μπορείς να καταλάβεις το περιεχόμενό του. Αυτά τα αρχεία λέγονται **δυαδικά** (binary) ή **μη μορφοποιημένα** (unformatted).

Ένα αρχείο-κείμενο:

- Μπορείς να το μεταφέρεις εύκολα σε διαφορετικά υπολογιστικά περιβάλλοντα και διαφορετικούς υπολογιστές.
- Μπορείς να το διαχειριστείς με πρόγραμμα, αλλά και με έναν απλό κειμενογράφο.

Αλλά:

- Αν έχει αριθμητικά δεδομένα, η διαχείρισή τους επιβραδύνεται από το ότι
  - οι τιμές της εσωτερικής παράστασης θα πρέπει να μεταφράζονται σε χαρακτήρες, όταν γράφονται στο αρχείο
  - οι χαρακτήρες (ψηφία) που διαβάζονται από το αρχείο θα πρέπει να μεταφράζονται στην εσωτερική παράσταση.

Ένα δυαδικό αρχείο:

---

[1] Θα χρησιμοποιήσουμε τα σύμβολα "<", ">", "," για να παραστήσουμε ένα αρχείο στο χαρτί και μόνο. Φυσικά: α) δεν τα χρησιμοποιεί ο ΗΥ όταν αποθηκεύει το αρχείο β) δεν θα πρέπει να τα μπερδεύεις με τα ειδικά σύμβολα της C++. Με αυτόν το συμβολισμό, το **κενό** αρχείο –δηλ. αυτό που δεν έχει στοιχεία– παριστάνεται ως: <>.

[2] Καμιά φορά θα το δεις και ως *αρχείο ascii*.

- Παίρνει τα στοιχεία του από τη μνήμη (ή τα δίνει σε αυτή) χωρίς καμμιά μετατροπή και έτσι η επεξεργασία του είναι πιο γρήγορη.

Αλλά:

- Δεν μπορείς να το μεταφέρεις εύκολα.
- Η διαχείρισή του απαιτεί ειδικό πρόγραμμα.

Για τους παραπάνω λόγους τα διάφορα «πακέτα», συνηθέστατα, μπορούν να γράφουν και να διαβάζουν αρχεία-κείμενα, ώστε να μπορούν να ανταλλάσσουν στοιχεία από τη μια εγκατάστασή τους στην άλλη. Άλλα αρχεία που δημιουργούν, για να διαβαστούν από άλλο «πακέτο»

- μπορεί να είναι αρχεία-κείμενα,
- μπορεί να είναι δυαδικά αρχεία,
- συχνά χρειάζονται ειδικά προγράμματα (φίλτρα) είτε είναι δυαδικά είτε είναι κείμενα.

Στο κεφάλαιο αυτό θα ασχοληθούμε αποκλειστικά με αρχεία-κείμενα.

8.1 Σειριακά Αρχεία στην C++

Σε κάθε Λειτουργικό Σύστημα (ΛΣ) υπάρχει ένα κομμάτι που λέγεται **διαχειριστής αρχείων** (file manager) ή **σύστημα για τα αρχεία** (file system). Ο διαχειριστής αρχείων κρατάει **καταλόγους** (directories) όπου για κάθε αρχείο υπάρχουν στοιχεία όπως: η θέση του πάνω στο μέσο αποθήκευσης, το μέγεθος του αρχείου, πότε δημιουργήθηκε το αρχείο, πότε ενημερώθηκε τελευταία φορά κλπ. Μέσα στον κατάλογο, το αρχείο έχει κάποιο *όνομα* που το έχει ορίσει αυτός που το δημιούργησε.

Παράδειγμα

Παρακάτω, βλέπεις την εικόνα καταλόγου που μας δίνει ο διαχειριστής αρχείων του ΛΣ Windows:

```
Volume in drive C has no label
 Serial Number of Volume is  50DA-631A

 Directory of C:\0809bea\t02

14/04/2009  03:10 AM    <DIR>          .
14/04/2009  03:10 AM    <DIR>          ..
13/04/2009  09:59 AM             7,875 AircraftType.cpp
13/04/2009  09:57 AM             3,319 AircraftType.h
24/03/2009  02:52 PM               250 aircraftTypes.txt
13/04/2009  09:05 AM            14,016 ask01a.cpp
13/04/2009  03:28 AM           522,116 ask01a.exe
13/04/2009  09:41 AM            14,538 ask01b.cpp
13/04/2009  09:41 AM           520,081 ask01b.exe
24/03/2009  03:36 PM             7,636 flightHours.txt
17/11/2008  10:08 PM             7,090 MyLib.cpp
13/04/2009  10:00 AM             3,856 nAircraftTypes.txt
13/04/2009  09:34 AM             3,856 nPilots.txt
13/04/2009  10:16 AM           233,472 ooPr_t02.doc
13/04/2009  10:18 AM               604 ooPr_t02.log
13/04/2009  10:20 AM           362,918 ooPr_t02.pdf
13/04/2009  10:17 AM         1,146,318 ooPr_t02.prn
13/04/2009  10:21 AM           371,671 ooPr_t02.ZIP
13/04/2009  06:50 AM           367,616 ooPr_t02b.doc
13/04/2009  03:23 AM             7,126 Pilot.cpp
13/04/2009  03:19 AM             3,441 Pilot.h
24/03/2009  04:03 PM             1,583 pilots.txt
09/04/2009  02:12 PM           677,110 t02.rar
              20 file(s)     4,276,492 bytes
               2 Directories 50,009,026,560 bytes free
```

Κάθε καταχώριση (γραμμή) του καταλόγου έχει: όνομα αρχείου, μέγεθος (σε ψηφιολέξεις), την ημερομηνία και την ώρα που ενημερώθηκε για τελευταία φορά.

Για να χειριστούμε ένα αρχείο μέσα από το πρόγραμμά μας θα πρέπει:

1. το ΛΣ να μας επιτρέψει πρόσβαση στο αρχείο αυτό,
2. να δημιουργήσουμε ένα κανάλι ή, όπως το λέει η C++, ένα **ρεύμα** (stream) από το αρχείο προς το πρόγραμμά μας (όταν διαβάζουμε) ή από το πρόγραμμά μας προς το αρχείο (όταν γράφουμε).
3. να βρούμε την αρχή του, από όπου θα αρχίσει η ανάγνωση ή η εγγραφή στο αρχείο.

Λέμε ότι πρέπει να **ανοίξουμε** (open) το ρεύμα του αρχείου.

Εδώ θα γνωρίσουμε τρεις τύπους (κλάσεις) ρευμάτων της C++:

- τον `ifstream` για ρεύματα από το αρχείο προς το πρόγραμμά μας (**i**nput **f**ile **stream**),
- τον `ofstream` για ρεύματα από το πρόγραμμά μας προς το αρχείο (**o**utput **f**ile **stream**),
- τον `fstream` για ρεύματα που μπορεί να είναι και διπλής κατεύθυνσης.

Για να χρησιμοποιήσεις αυτούς τους τύπους θα πρέπει να περιλάβεις στο πρόγραμμά σου το αρχείο fstream (`#include <fstream>`).

Πριν προχωρήσουμε πιο κάτω θα κάνουμε μια παρένθεση για να πούμε δύο λόγια για τις κλάσεις. Μια **κλάση** (class) είναι ένας τύπος στοιχείων. Οι μεταβλητές και οι τιμές μιας κλάσης ονομάζονται και **αντικείμενα** (objects). Αυτά έχουν **περικλεισμένες** (encapsulated) **μεθόδους** (methods) ή **συναρτήσεις-μέλη** (member functions), για τη διαχείριση των στοιχείων τους. Π.χ. οι τρεις κλάσεις που αναφέραμε πιο πάνω έχουν μια μέθοδο που ονομάζεται *open* για άνοιγμα ρεύματος. Αν λοιπόν έχουμε δηλώσει:

```
ifstream a;
```

ανοίγουμε το ρεύμα *a* από το αρχείο text1.txt προς το πρόγραμμά μας με την:

```
a.open( "text1.txt" );
```

Ένα άλλο χαρακτηριστικό των κλάσεων είναι η **κληρονομιά** (inheritance) που μπορεί να αφήσει μια κλάση σε μια άλλη κλάση-παιδί της, π.χ. διάφορες μέθοδοι, τελεστές κλπ. Όπως θα δεις στη συνέχεια, μετά από τις παραπάνω δηλώσεις θα μπορούμε να διαβάζουμε από το αρχείο text1.txt ως εξής:

```
a >> x;
```

όπως ακριβώς διαβάζουμε από το πληκτρολόγιο με τη:

```
cin >> x;
```

παρ' όλο που το *cin* είναι ρεύμα κλάσης *istream*. Η *ifstream* είναι απόγονος της *istream* και –μέσω αυτής– μιας άλλης κλάσης, της *ios_base*, από την οποίαν κληρονόμησαν και οι δύο τον τελεστή "`>>`".

Αυτά τα ολίγα για τις κλάσεις προς το παρόν· θα τις ξαναδούμε αργότερα.

8.2 Αρχεία και Ρεύματα – Μια Εικόνα

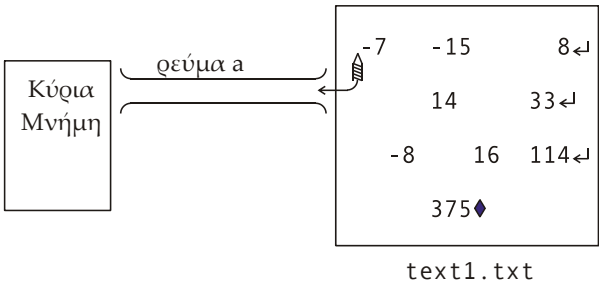
Θα δώσουμε τώρα μια εικόνα για το (σειριακό) αρχείο-κείμενο και με αυτή θα δούμε τη διαχείρισή του.

Ας πούμε λοιπόν ότι το αρχείο είναι «γραμμένο» πάνω σε μια μαγνητοταινία (Σχ. 8-1).

Το αρχείο έχει μια αρχή και ένα τέλος. Το τέλος, εδώ στην εικόνα, το σημειώνουμε με το "♦". Αυτό δεν σημαίνει ότι ο ΗΥ σημειώνει το τέλος αρχείου με τον ίδιο τρόπο, αλλά, όπως θα δεις παρακάτω, αυτό δεν μας ενδιαφέρει. Το αρχείο-κείμενο είναι χωρισμένο σε γραμμές· στο σχήμα μας σημειώνουμε το τέλος γραμμής με το "↵".

Μια μαγνητική κεφαλή (παρόμοια με αυτήν του μαγνητοφώνου) διαβάζει το (γράφει στο) αρχείο. Λέγεται **κεφαλή ανάγνωσης/εγγραφής** (read/write head).

Η παράσταση του Σχ. 8-1 είναι απλουστευμένη αλλά θα μας βοηθήσει να καταλάβουμε τη διαχείριση του αρχείου-κειμένου. Βέβαια κρύβουμε λεπτομέρειες και περιπλοκές. Μια από αυτές είναι η ύπαρξη και ο ρόλος του *ενταμιευτή* που θα μας χρειαστεί σε κάποιο σημείο στη συνέχεια. Παρακάτω λέμε μερικά πράγματα γι' αυτόν.



Σχ. 8-1 Παράσταση αρχείου-κειμένου (text1.txt) και ρεύματος *a* για την ανάγνωσή του. Παριστάνουμε το αρχείο όπως περίπου θα το δεις σε έναν κειμενογράφο. Τα "↵" και "♦" σημειώνουν το τέλος γραμμής και αρχείου αντιστοίχως. Το "⇧" παριστάνει την κεφαλή ανάγνωσης/εγγραφής

Με τον όρο **ενταμιευτής** αποδίδουμε την αγγλική λέξη «buffer». Στα αγγλικά σημαίνει: αυτός που απαλλάσσει κάποιον από μια ενοχλητική δουλειά. Και αυτό ακριβώς κάνει ο ενταμιευτής: απαλλάσσει την ΚΜΕ (Κεντρική Μονάδα Επεξεργασίας, CPU) από τη συνεχή απασχόληση με τις περιφερειακές μονάδες –στην περίπτωσή μας μονάδες βοηθητικής μνήμης. Ο έλεγχος των μονάδων αυτών είναι αρκετά χρονοβόρος. Γι' αυτό, το ρεύμα περιλαμβάνει μια περιοχή μνήμης αρκετά μεγάλη (συνηθισμένες τιμές 1 *kB*, 2 *kB*). Όταν λοιπόν ζητήσουμε ανάγνωση μιας τιμής από το αρχείο δεν φέρνει μόνον αυτό που ζητήσαμε (π.χ. `'-'` και `'7'`) αλλά, π.χ., 1024 χαρακτήρες. Έτσι, οι επόμενες αναγνώσεις δεν απαιτούν πρόσβαση στο δίσκο αλλά μεταφορές μέσα στην κύρια μνήμη. Παρομοίως, όταν γράφουμε στο αρχείο, στην πραγματικότητα τα στοιχεία γράφονται στον ενταμιευτή. Όταν αυτός «γεμίσει», το περιεχόμενό του αντιγράφεται στο αρχείο.

8.3 Πώς Διαβάζουμε Ένα Αρχείο

Ας πούμε λοιπόν ότι στο δίσκο μας υπάρχει το αρχείο-κείμενο text1.txt. Τώρα θα δούμε πώς θα γράψουμε ένα πρόγραμμα που θα διαβάζει αυτό το αρχείο.

Όπως είπαμε και παραπάνω, μας χρειάζεται ένα ρεύμα κλάσης *ifstream*.Θα πρέπει λοιπόν να δηλώσουμε:

```
ifstream  a;
```

και να το ανοίξουμε ώστε να συνδέσει το αρχείο με το πρόγραμμά μας:

```
a.open( "text1.txt" );
```

Το στιγμιότυπο που βλέπεις στο Σχ. 8-1 δείχνει την κατάσταση που βρισκόμαστε μετά την *open()*. Η κεφαλή ανάγνωσης/εγγραφής βρίσκεται στην αρχή του αρχείου.

Ας πούμε ότι έχουμε δηλώσει ακόμη:

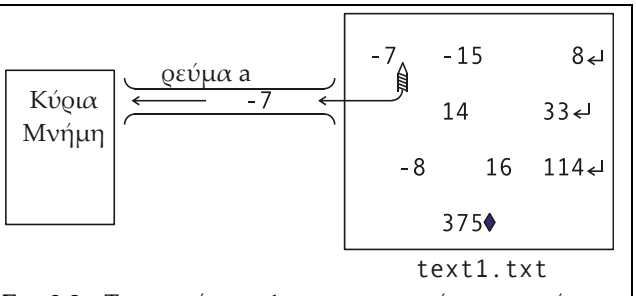
```
short int x, y;
```

και θέλουμε να διαβάσουμε την πρώτη τιμή του αρχείου και να την αποθηκεύσουμε στη *x*. Δίνουμε την εντολή:

```
a >> x;
```

Με την εκτέλεση αυτής της εντολής γίνονται τα εξής:

1. Η κεφαλή περνάει το αρχικό κενό και φτάνει στο `'-'`.
2. Η κεφαλή διαβάζει τους χαρακτήρες `'-'` και `'7'` και σταματάει όταν βρει το κενό.
3. Οι χαρακτήρες διαβιβάζονται στην κύρια μνήμη.



Σχ. 8-2 Το αρχείο text1.txt και το ρεύμα *a* μετά την ανάγνωση της πρώτης τιμής.

4. Οι δύο χαρακτήρες μεταφράζονται στην εσωτερική παράσταση του αριθμού "**-7**" που αποθηκεύεται στη θέση της μνήμης *x*.

Στο Σχ. 8-2 δείχνουμε την κατάσταση μετά την ανάγνωση της πρώτης τιμής.

Στη συνέχεια μπορούμε να επεξεργαστούμε αυτήν την τιμή· μπορούμε να δώσουμε, για παράδειγμα:

```
y = x*x;
cout << "   x = " << x << "   x^2 = " << y << endl;
```

Παίρνουμε αποτέλεσμα:

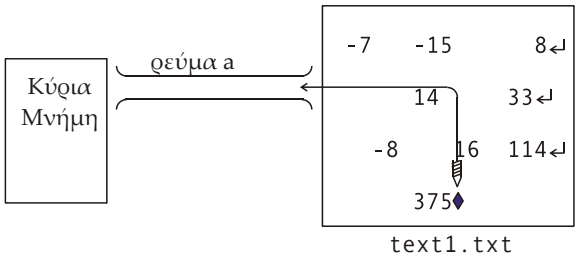
```
x = -7   x^2 = 49
```

Αν τώρα θέλουμε να επεξεργαστούμε τη δεύτερη τιμή, ζητούμε και πάλι εκτέλεση της εντολής:

```
a >> x;
```

Η κεφαλή θα ξεπεράσει τα κενά και θα διαβάσει και θα στείλει μέσω του *a* τους χαρακτήρες '**-**', '**1**', '**5**'. Με τον ίδιο τρόπο διαβάζεται και το 8. Αν ξαναζητήσουμε ανάγνωση τιμής, η κεφαλή θα ξεπεράσει το σημάδι τέλους γραμμής και τα κενά για να διαβάσει τους '**1**', '**4**'.

Συνεχίζοντας με αυτόν τον τρόπο φτάνουμε κάποτε στην κατάσταση που βλέπεις στο Σχ. 8-3. Αν ζητήσεις και πάλι εκτέλεση της `a >> x` αυτή θα αποτύχει και η τιμή της *x* δεν θα αλλάξει. Κάπου μέσα στον υπολογιστή σου «θα σηκωθεί μια σημαία» –κάποιο δυαδικό ψηφίο θα αλλάξει από 0 σε 1– που θα δείχνει ότι φτάσαμε στο τέλος του αρχείου. Μπορούμε να δούμε αυτή τη σημαία; Ναι! Η *ifstream* έχει μια μέθοδο, που ονομάζεται *eof* (χωρίς παραμέτρους) από τις αγγλικές λέξεις **e**nd **o**f **f**ile (τέλος αρχείου). Αν ελέγξεις την τιμή της `a.eof()`



Σχ. 8-3 Τι θα γίνει αν στην κατάσταση αυτή εκτελεστεί η εντολή `a >> x`; Η τιμή της *x* δεν θα αλλάξει. Αν ελέγξουμε την τιμή της `a.eof()` θα είναι μη μηδενική (`true`).

- μετά από μια προσπάθεια ανάγνωσης που απέτυχε διότι φτάσαμε στο τέλος του αρχείου αυτή θα είναι μη μηδενική (ως συνθήκη μεταφράζεται σε `true`)
- μετά από μια επιτυχή προσπάθεια ανάγνωσης αυτή θα είναι μηδενική (που ως συνθήκη θεωρείται `false`).

Φυσικά, αν η ανάγνωση δεν γίνει δεν έχει νόημα να επεξεργαστούμε την τιμή της *x*.

Το παρακάτω απλό πρόγραμμα κάνει μια στοιχειώδη επεξεργασία στο αρχείο που χρησιμοποιήσαμε στα παραδείγματά μας μέχρι τώρα. Υπολογίζει το τετράγωνο του κάθε στοιχείου. Επειδή το πρόγραμμα εκτελείται σε μια υλοποίηση της C++ όπου *SHRT_MAX* = 32767, αν η απόλυτη τιμή κάποιου στοιχείου είναι μεγαλύτερη από 181 (που είναι η ακέραια προσέγγιση της $\sqrt{SHRT\_MAX}$) τότε υπολογίζει το τετράγωνο του `x % 181`.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream  a;
   short int x, xr, y;

   a.open( "text1.txt" );
   a >> x;
   while ( !a.eof() )
```

```
   {
      if ( abs(x) > 181 )  xr = x % 181;
                     else  xr = x;
      y = xr*xr;
      cout << "   x =";  cout.width(4);  cout << x;
      cout << "    xr =";  cout.width(4);  cout << xr;
      cout << "    xr^2 =";  cout.width(5);  cout << y << endl;
      a >> x;
   } // while
   a.close();
} // main
```

Αυτό το πρόγραμμα δίνει τελικώς:

```
x =  -7    xr =  -7    xr^2 =   49
x = -15    xr = -15    xr^2 =  225
x =   8    xr =   8    xr^2 =   64
x =  14    xr =  14    xr^2 =  196
x =  33    xr =  33    xr^2 = 1089
x =  -8    xr =  -8    xr^2 =   64
x =  16    xr =  16    xr^2 =  256
x = 114    xr = 114    xr^2 =12996
x = 375    xr =  13    xr^2 =  169
```

Πρόσεξε την τελευταία εντολή:

```
   a.close();
```

Με την εκτέλεσή της το ρεύμα *a* αποσυνδέεται από το αρχείο text1.txt. Αν θέλουμε μπορούμε να το χρησιμοποιήσουμε για να επεξεργαστούμε άλλο αρχείο.

Το πρόγραμμα αυτό είναι παράδειγμα επεξεργασίας σειριακού αρχείου. Στο Πλ. 8.1 βλέπεις το γενικό σχήμα αυτής της επεξεργασίας. *Ξεκινώντας με a.open(..) και προχωρώντας μέχρις ότου η* `a.eof()` *δώσει μη μηδενική τιμή είναι σίγουρο ότι θα διαβάσουμε όλες τις τιμές του αρχείου*. Αυτό δεν σημαίνει ότι πρέπει να τις επεξεργαστείς όλες υποχρεωτικώς· μπορείς να κάνεις επιλεκτική επεξεργασία. Όπως βλέπεις, το σχήμα επεξεργασίας σειριακού αρχείου είναι όμοιο με το σχήμα επανάληψης με φρουρό.

Παρατηρήσεις ▶

1. Ας δούμε τώρα το εξής πρόβλημα: θέλουμε να διαβάσουμε την *πέμπτη* τιμή του αρχείου (το 33). Πρέπει να κάνουμε τα εξής:

i. Να φέρουμε το αρχείο στην αρχή του (Σχ. 8-1).

ii. Να διαβάσουμε και να αγνοήσουμε τις τέσσερις πρώτες τιμές του αρχείου.

iii. Να διαβάσουμε την πέμπτη τιμή.

Στις διαδικασίες αυτές φαίνεται πολύ καλά ένα βασικό χαρακτηριστικό των σειριακών αρχείων:

- ***Για να διαβάσουμε (ή να γράψουμε) το ν-οστό στοιχείο ενός σειριακού αρχείου πρέπει να περάσουμε κατ' ανάγκη τις ν – 1 προηγούμενες τιμές.***

Για να γίνει όμως αυτό,

- ***Η επεξεργασία κάθε σειριακού αρχείου ξεκινάει πάντα από την αρχή του.***

Πλαίσιο 8.1

Επεξεργασία Σειριακού Αρχείου

```
ifstream a;
   :
a.open( όνομα αρχείου );
a >> x;
while (!a.eof())
{
   Εντολές επεξεργασίας της x
   a >> x;
} // while
a.close();
```

2. Κάτι άλλο που φαίνεται στο πρόγραμμά μας είναι το εξής: μετά από την εκτέλεση της "`a >> x`" υπολογίζεται η "`!a.eof()`" και μόνο αν βρεθεί `true`, αν η ανάγνωση έγινε κανονικώς, επεξεργαζόμαστε την τιμή της *x*. Και αυτή είναι μια γενική αρχή:

- ***Πριν επεξεργαστούμε μια τιμή που (υποτίθεται ότι) διαβάσαμε από ένα αρχείο ελέγχουμε την eof για να δούμε αν η ανάγνωση έγινε κανονικά.***

3. Όταν προσπαθείς να πάρεις μια τιμή από ένα αρχείο-κείμενο με τον "`>>`", εκτός από τα κενά (διαστήματα) και τα σημάδια τέλους γραμμής, «τρώγονται» και οι στηλοθέτες (tabs). Όλα αυτά η C++ τα ονομάζει **λευκά διαστήματα** (white spaces).

4. Αν το αρχείο που επεξεργάζεσαι δεν είναι στον ίδιο υποκατάλογο με το πρόγραμμά σου θα πρέπει να περιλάβεις στο όνομα και τη **διαδρομή** (path). Αν δουλεύεις σε περιβάλλον Windows η διαδρομή θα περιλαμβάνει τον χαρακτήρα `'\'` που, όπως έχουμε πει, πρέπει να γράφεται δύο φορές. Αν έχεις π.χ.:

```
c:\students\datfl\text1.txt
```

θα πρέπει να δώσεις:

```
a.open( "c:\\students\\datfl\\text1.txt" );
```

4. Έχεις τη δυνατότητα να ανοίξεις το αρχείο όταν το δηλώνεις, π.χ.:

```
ifstream a( "text1.txt" );
```

Στην περίπτωση αυτή, δεν θα δώσεις `a.open("text1.txt")` στη συνέχεια.◀

8.3.1 *cin.eof()*

Μπορούμε να ελέγχουμε για «τέλος αρχείου» όταν παίρνουμε δεδομένα από το πληκτρολόγιο μέσω του *cin*; Ναι! Ο χρήστης «πληκτρολογεί το eof» δίνοντας

- `char(3)` (**E**nd of **Te**x**t**, ETX) με πληκτρολόγηση <ctrl-C> ή
- `char(4)` (**E**nd **O**f **T**ransmission, EOT) με πληκτρολόγηση <ctrl-D> ή
- `char(26)` με πληκτρολόγηση <ctrl-Z> (συνήθως σε περιβάλλον Windows).

Δυστυχώς η πληκτρολόγηση εξαρτάται από το ΛΣ και τον μεταγλωττιστή. Για παράδειγμα σε εφαρμογές «κονσόλας» στα Windows ο ETX δεν μπορεί να χρησιμοποιηθεί διότι του «έχει ανατεθεί» άλλος ρόλος (διακόπτει την εκτέλεση.) Χρησιμοποιείται, από την εποχή του MS-DOS, ο <ctrl-Z>. Θα το δεις να δουλεύει αν χρησιμοποιείς gcc (π.χ. Dev C++) ή BC++ 5.02.

8.3.2 Για να Ξαναχρησιμοποιήσεις το Ρεύμα

Ας πούμε ότι μετά την

```
ifstream a( "text1.txt" );
```

διάβασες το αρχείο `text1.txt` μέχρι το τέλος του και σταμάτησες την ανάγνωση μόλις πήρες *a.eof()*. Σε ποια κατάσταση είναι το ρεύμα; Σε «κακή κατάσταση»[3] ή, με άλλα λόγια, έχει ανασταλεί η λειτουργία του: δεν μπορεί να δεχθεί οποιαδήποτε εντολή· σε ορισμένες περιπτώσεις (μεταγλωττιστές) δεν εκτελεί ούτε την *close()*! Αν το ρεύμα έχει ανοιχτεί για διάβασμα δεν πάθαμε ζημιά εκτός από μια περίπτωση: Αν θέλουμε να ξαναχρησιμοποιήσουμε το ρεύμα για να δουλέψουμε με το ίδιο αρχείο ή με κάποιο άλλο.

Για να μπορούμε να ξαναχρησιμοποιήσουμε το ρεύμα θα πρέπει το *ανατάξουμε* δίνοντας την εντολή:

```
a.clear();
```

Ας πούμε ότι μετά την *a.eof()* θέλουμε να χρησιμοποιήσουμε το *a* για να διαβάσουμε ένα άλλο αρχείο, ας πούμε το `text2.txt`. Στην περίπτωση αυτή οι εντολές:

```
a.close();
a.open( "text2.txt" );
```

δεν αρκούν. Το σωστό είναι να γράψουμε:

```
a.clear();
a.close();
a.open( "text2.txt" );
```

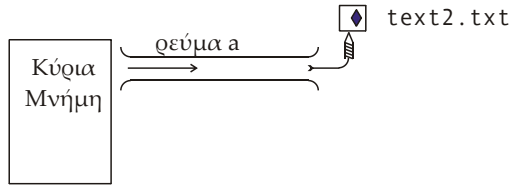
8.4 Πώς Γράφουμε Ένα Αρχείο

Τώρα θα δούμε πώς δημιουργούμε ένα αρχείο-κείμενο. Όπως καταλαβαίνεις τώρα θα πρέπει να δημιουργήσουμε ένα ρεύμα *από το πρόγραμμά μας προς το αρχείο*. Θα πρέπει λοιπόν να δηλώσουμε:

```
ofstream a;
```

Και η κλάση *ofstream* έχει μέθοδο *open()* για το άνοιγμα του αρχείου:

```
a.open( "text2.txt" );
```



Σχ. 8-4 Παράσταση αρχείου-κειμένου (text2.txt) και ρεύματος *a* για τη δημιουργία του. Το στιγμιότυπο δείχνει την κατάσταση ακριβώς μετά την εκτέλεση της `a.open('text2.txt')`. Το αρχείο είναι άδειο.

Εδώ όμως έχουμε διαφορά από την *open()* της *ifstream*: *Αν το αρχείο text2.txt υπήρχε, με την εκτέλεση της open() χάθηκε το περιεχόμενό του!* Το αρχείο καθαρίζεται και είναι έτοιμο για γράψιμο. Στο Σχ. 8-4 βλέπεις σχηματικά την κατάσταση μετά την εκτέλεση της *open()*.

Και τώρα θέλω να γράψω στο αρχείο την τιμή "`23`". Πώς θα γίνει αυτό; Μάλλον το έχεις μαντέψει: όπως η *ifstream* έχει τον "`>>`", που τον ξέρουμε από το *cin*, και η *ofstream* έχει τον, γνωστό μας από τη *cout*, "`<<`". Αν λοιπόν δώσουμε:

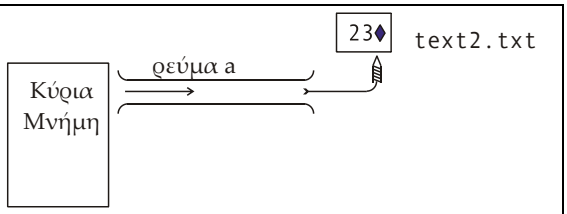
```
a << 23;
```

θα γίνουν αυτά που προσπαθούμε να δείξουμε στο Σχ. 8-5:

1. Η ακέραιη σταθερά «μεταφράζεται» στους χαρακτήρες `'2'`, `'3'`.
2. Οι χαρακτήρες διαβιβάζονται στη μαγνητική κεφαλή.
3. Οι χαρακτήρες γράφονται στο αρχείο.

Στη συνέχεια δίνουμε την εντολή:

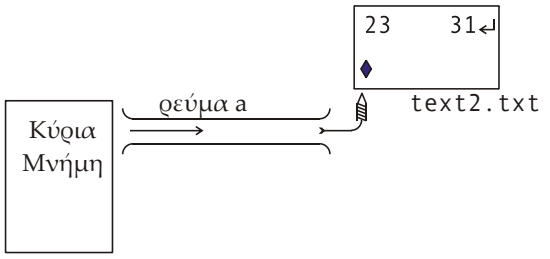
```
a << "    " << 31 << endl;
```



Σχ. 8-5 Παράσταση αρχείου-κειμένου (text2.txt) και ρεύματος *a* μετά την εκτέλεση της εντολής: `a << 23`.

[3] Όρος της C++ "(*std*::*ios*::)*bad*", θα τον δούμε αργότερα...

και το αποτέλεσμα είναι αυτό του Σχ. 8-6. Πρόσεξε τα τέσσερα διαστήματα: ζητήσαμε να γραφούν και γι' αυτό γράφηκαν. Αν δίναμε: "`a << 31`", το "`31`" θα «κολλούσε» στο "`23`" και θα είχαμε "`2331`".



Σχ. 8-6 Η κατάσταση μετά την εκτέλεση της εντολής: "`a << "  " << 31 << endl`". Λόγω του "`endl`" γράφηκε το σημάδι τέλους γραμμής και η κεφαλή είναι έτοιμη να γράψει στην αρχή της επόμενης γραμμής.

Το `endl` μας πηγαίνει στην αρχή νέας γραμμής. Είχαμε πει ότι γράφοντας στο *cout* τον `'\n'` έχουμε το ίδιο αποτέλεσμα· ισχύει αυτό και για τα αντικείμενα της κλάσης *ofstream*, όπως είναι το *a*; Ναι! Για την ακρίβεια το `'\n'` (= `char`(10)) είναι το σημάδι τέλους γραμμής για αρχεία-κείμενα που γράφει η C++. Τώρα όμως μπορούμε να δούμε τη διαφορά:

- Ζητώντας να γραφεί το `endl` πετυχαίνεις δύο πράγματα:
  - να πας σε νέα γραμμή και
  - να αντιγραφεί το όποιο περιεχόμενο του ενταμιευτή (η γραμμή που συμπληρώθηκε) στο αρχείο.
- Ζητώντας να γραφεί το `'\n'` πετυχαίνεις μόνο να πας σε νέα γραμμή. Το περιεχόμενο του ενταμιευτή θα αντιγραφεί όταν αυτός γεμίσει ή όταν ζητηθεί κάτι τέτοιο.

Το `endl` κάνει το πρόγραμμά σου *πιο αργό αλλά ασφαλέστερο*. Δηλαδή: αν, ας πούμε, έχεις διακοπή στην παροχή ηλεκτρικής ενέργειας την ώρα που δημιουργείς το αρχείο, όσες γραμμές ζήτησες να γραφούν, θα έχουν γραφεί στο αρχείο. Σε μια τέτοια περίπτωση, αν γράφεις με το `'\n'`, θα χάσεις όλο το περιεχόμενο του ενταμιευτή.

Όταν τελειώσεις το γράψιμο του αρχείου μην ξεχάσεις να το κλείσεις με μια *close()*. Η *close()* θα φροντίσει να αντιγράψει στο αρχείο όλο το περιεχόμενο του ενταμιευτή.

Το παρακάτω πρόγραμμα διαβάζει από το πληκτρολόγιο ζεύγη ακέραιων τιμών και τα γράφει, ένα ζεύγος ανά γραμμή σε αρχείο-κείμενο με όνομα text2.txt. Η ανάγνωση τελειώνει με το ζεύγος (0,0).

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ofstream a;
   int x, y;

   a.open( "text2.txt" );
   cout << " Δωσε τα x, y. 0,0 για τελος: ";  cin >> x >> y;
   while ( x != 0 || y != 0 )
   {
      a.width( 5 );  a << x;
      a.width( 5 );  a << y << endl;
      cout << " Δωσε τα x, y. 0,0 για τελος: ";  cin >> x >> y;
   } // while
   a.close();
} // main
```

Παρατηρήσεις: ▶

1. "Τι είναι το `a.width(5)`;" Ακριβώς σαν το `cout.width(5)`. Ζητάει να γραφεί το επόμενο στοιχείο στο αρχείο που είναι συνδεμένο με το *a* σε 5 θέσεις τουλάχιστον. Παρομοίως, για τα μέλη της κλάσης *ofstream* δουλεύουν όπως ξέρουμε οι μέθοδοι *precision* και *setf*.
2. Ας αλλάξουμε τη συνθήκη της **while**:

```
   cout << " Δωσε τα x, y. <ctrl-Z> για τελος: ";  cin >> x >> y;
```

```
   while ( !cin.eof() )
   {
      a.width( 5 );  a << x;
      a.width( 5 );  a << y << endl;
      cout << " Δωσε τα x, y. <ctrl-Z> για τελος: ";
      cin >> x >> y;
   } // while
```

Η εκτέλεση των επαναλήψεων θα τερματισθεί αν πληκτρολογήσουμε <ctrl-Z>.[4] Έτσι, δεν μας χρειάζεται φρουρός! ◄

8.5 Ένα «Πραγματικό» Πρόβλημα

Το παράδειγμα επεξεργασίας αρχείου που είδαμε στις προηγούμενες παραγράφους δεν είχε και πολύ νόημα. Εκεί θέλαμε να τραβήξουμε την προσοχή σου στις πράξεις του αρχείου. Τώρα ας δούμε ένα πιο «πραγματικό» πρόβλημα.

Έχουμε ένα αρχείο –με όνομα στο δίσκο exp4.txt– με τιμές τύπου **double**. Το πλήθος τους δεν είναι γνωστό. Θέλουμε να βρούμε και να τυπώσουμε το άθροισμα, τη Μέση Αριθμητική Τιμή και το πλήθος των πραγματικών αριθμών που υπάρχουν στο αρχείο. Ακόμη θέλουμε: το πλήθος, το άθροισμα και τη μέση τιμή όσων από τις τιμές που διαβάζει είναι θετικές και μέχρι 10.

Η αντίδρασή σου θα πρέπει να είναι: «Ωχ, πάλι αυτό!» ή «Μα αυτό το ξέρω!». Ας υποθέσουμε ότι είναι η δεύτερη και ας προχωρήσουμε. Γύρισε στο Κεφ. 6 (§6.1.2) για να δεις το πρόγραμμα *Μέση Τιμή* 4, ξαναδές το γενικό σχήμα επεξεργασίας σειριακού αρχείου (Πλ. 8.1) και ας γράψουμε το *Μέση Τιμή* 5.

Ας πούμε ότι το αρχείο θα το δουλέψουμε με το ρεύμα *a*. Θα πρέπει να το δηλώσουμε:

```
   ifstream a;
```

Με την τιμή - φρουρό τι θα κάνουμε; Τίποτε! *Δεν μας χρειάζεται!* Αντί να ψάχνουμε για φρουρό θα ελέγχουμε μήπως φτάσαμε στο τέλος του αρχείου. Να το γενικό σχήμα επεξεργασίας του αρχείου:

```
   a.open( "exp4.txt" );
   a >> x;
   while ( !a.eof() )
   {
      E
      a >> x;
   } // while
   a.close();
```

Ποιές είναι οι *E*; Νάτες:

```
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
```

Στη συνέχεια βλέπεις το πρόγραμμα *Μέση Τιμή* 5. Οι διαφορές του από το *Μέση Τιμή* 4 είναι πολύ μικρές. Και ο μόνος λόγος που ξαναπαραθέτουμε ολόκληρο, είναι ακριβώς για να δεις αυτήν την ομοιότητα και να επισημάνεις τις διαφορές.

```
// πρόγραμμα: Μέση Τιμή 5
#include <iostream>
#include <fstream>
using namespace std;
int main()
```

[4] Δοκίμασέ το και αν δουλέψει…

```
{
   ifstream a;

   int n;          // Μετρητής όλων των στοιχείων
   int selN;       // Μετρητής επιλεγόμενων στοιχείων
   double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;     // Το άθροισμα όλων των στοιχείων
   double selSum;  // Άθροισμα επιλεγόμενων στοιχείων
   double avrg;    // Μέση Αριθμητική Τιμή όλων των στοιχείων
   double selAvrg; // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων

   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
   a.open( "exp4.txt" );
   a >> x;
   while ( !a.eof() )
   {
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
      a >> x;
   } // while
   a.close();
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   } // if ( n
   cout << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
   if ( selN > 0 )
   {
      selAvrg = selSum/selN;
      cout << " ΑΘΡΟΙΣΜΑ = " << selSum
           << "  <x> = " << selAvrg << endl;
   } // if ( selN
} // main
```

Πρόσεξε ότι κλείνουμε το αρχείο αμέσως μόλις παύουμε να ασχολούμαστε μαζί του.

Ας πούμε τώρα ότι έχουμε το εξής πρόβλημα: Θέλουμε τα αποτελέσματα της επεξεργασίας όχι στην οθόνη αλλά σε ένα αρχείο-κείμενο με το όνομα report.txt. Τι θα πρέπει να αλλάξουμε στο *Μέση Τιμή* 5;

- Θα πρέπει να δηλώσουμε ένα ρεύμα *b* κλάσης *ofstream*, που θα το ανοίξουμε προς το report.txt.
- Ό,τι στέλνουμε στην οθόνη μέσω του *cout* να το στείλουμε στο report.txt μέσω του *b*.

```
// πρόγραμμα: Μέση Τιμή 6
#include <fstream>
using namespace std;
int main()
{
   ifstream a;
   ofstream b;
// . . .
   b.open( "report.txt" );
   b << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      b << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   } // if ( n
   b << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
```

```
   if ( selN > 0 )
   {
      selAvrg = selSum/selN;
      b << " ΑΘΡΟΙΣΜΑ = " << selSum
        << "  <x> = " << selAvrg << endl;
   } // if ( selN
```

Ε! Ξεχάσαμε την "`#include <iostream>`"! Δεν την ξεχάσαμε! Την παραλείψαμε διότι δεν τη χρειαζόμαστε: ούτε το *cin* ούτε το *cout* υπάρχει στο πρόγραμμά μας.

Να πώς περίπου θα είναι το περιεχόμενο του report.txt:

```
Διάβασα 37 αριθμούς
ΑΘΡΟΙΣΜΑ = 560.767  <x> = 15.1559
Διάλεξα 12 αριθμούς ε (0,10]
ΑΘΡΟΙΣΜΑ = 46.6869  <x> = 3.89057
```

8.6 Και Διάβασμα και Γράψιμο

Ας δούμε τώρα το εξής πρόβλημα: Θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει άγνωστο πλήθος πραγματικών αριθμών x_1, x_2, ... –που καθένας τους είναι πολύ μικρότερος από 10000– και θα τους αποθηκεύει σε ένα αρχείο fltnum.txt. Θέλουμε ακόμη:

α) να υπολογίζει και να τυπώνει τη Μέση Τιμή τους <*x*>,

β) να υπολογίζει και να τυπώνει τις τιμές της συνάρτησης: $e^{\frac{<x>-x}{<x>}}$ για τους αριθμούς αυτούς. Δηλαδή τα $y_k = e^{\frac{<x>-x_k}{<x>}}$.

Ξέρουμε να κάνουμε τα πάντα! Ένα σημείο χρειάζεται προσοχή: για να υπολογίσουμε τη μέση τιμή θα πρέπει να έχουμε όλους τους αριθμούς· δηλαδή *θα την έχουμε υπολογίσει όταν θα έχουμε γράψει όλους τους αριθμούς στο αρχείο*. Για να υπολογίσουμε τα y_k θα πρέπει να ξαναδιαβάσουμε τους αριθμούς, όχι βέβαια από το πληκτρολόγιο (όχι και να τους ξαναπληκτρολογήσουμε), αλλά από το αρχείο.

Να λοιπόν το σχέδιό μας:

```
Πέρασε τους αριθμούς στο αρχείο και υπολόγισε πλήθος και μέση τιμή
Διάβασε όλους τους αριθμούς από το αρχείο και για κάθε έναν από
                    αυτούς (xk) υπολόγισε και γράψε το yk = e^((<x>-xk)/<x>)
```

Φυσικά θα πρέπει να προσέξουμε: Το δεύτερο βήμα θα εκτελεσθεί μόνο αν το πλήθος των αριθμών που δώσαμε (*n*) δεν είναι μηδέν:

```
Πέρασε τους αριθμούς στο αρχείο και υπολόγισε πλήθος και μέση τιμή
if ( n > 0 )
{  Διάβασε όλους τους αριθμούς από το αρχείο και για κάθε έναν από
                    αυτούς (xk) υπολόγισε και γράψε το yk = e^((<x>-xk)/<x>)  }
```

Το πρώτο βήμα είναι κατά βάση το *Μέση Τιμή* 3 με τις εξής διαφορές:

- Θα πρέπει να φυλάγουμε στο αρχείο κάθε τιμή που δίνεται.
- Θα πρέπει να αλλάξουμε φρουρό: το 0 δεν αποκλείεται ενώ μια καλή επιλογή είναι το 9999.

```
   const double sentinel = 9999.0;
   ofstream  a;
// . . .
   sum = 0;  n = 0;
   a.open( "fltnum.txt" );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
```

```
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   a.close();
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
   }
```

Για το δεύτερο βήμα τροποποιούμε το πρόγραμμα της προηγούμενης παραγράφου:

```
   ifstream  b;      // για διάβασμα αρχείου

      b.open( "fltnum.txt" );   m = 0;
      b >> x;
      while ( !b.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         b >> x;
      } // while
      b.close();
```

Όλα αυτά θα γίνουν βέβαια αν και μόνον αν $n > 0$.

Να ολόκληρο το πρόγραμμα:

```
// πρόγραμμα: Μέση Τιμή 3+
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const double sentinel = 9999.0;
   ofstream  a;
   ifstream  b; // για διάβασμα αρχείου
   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)
   int m;
   double y;

   sum = 0;  n = 0;
   a.open( "fltnum.txt" );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   a.close();
   cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
   if ( n > 0 )
```

```
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
      b.open( "fltnum.txt" );   m = 0;
      b >> x;
      while ( !b.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         b >> x;
      } // while
      b.close();
   }
} // main
```

Στο πρόγραμμα αυτό χρησιμοποιήσαμε δύο διαφορετικά ρεύματα για να διαχειριστούμε το ίδιο αρχείο· δύο διαφορετικούς «μονόδρομους». Δεν θα μπορούσαμε να χρησιμοποιήσουμε έναν μόνο «δρόμο διπλής κατεύθυνσης»; Ναι! Αρκεί να δηλώσουμε ρεύμα κλάσης *fstream*. Τι διαφορές θα έχουμε στην περίπτωση αυτή;

- Στη δήλωση:

```
   fstream  a;
```

- Στο άνοιγμα: Η *open()* παίρνει και μια δεύτερη παράμετρο, που καθορίζει τον τρόπο διαχείρισης του αρχείου: αν δώσουμε "`ios_base::in`" ανοίγουμε το αρχείο για διάβασμα, αν δώσουμε "`ios_base::out`" το ανοίγουμε για γράψιμο, αν δώσουμε

  "`ios_base::in|ios_base::out`"

  το ανοίγουμε και για διάβασμα και για γράψιμο.[5] Θα πρέπει λοιπόν να το ανοίξουμε ως εξής:

```
   a.open( "fltnum.txt", ios_base::in|ios_base::out );
```

- Στο κλείσιμο και την επαναφορά: Θα κλείσουμε το ρεύμα μια φορά μόνον στο τέλος. Πώς όμως θα επαναφέρουμε το αρχείο στην αρχή του για να το ξαναδιαβάσουμε; Η *fstream* (και η *ifstream*) έχει μια μέθοδο, τη *seekg()*, που όταν κληθεί με όρισμα 0 – στην περίπτωσή μας: "`a.seekg(0)`"– ξαναφέρνει το ρεύμα στην αρχή του αρχείου για να (ξανα)διαβαστεί.

  Τα υπόλοιπα παραμένουν ίδια:

```
// πρόγραμμα: Μέση Τιμή 3+ inout
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const double sentinel = 9999.0;
   fstream  a;  // για διάβασμα αρχείου και γράψιμο
   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)
```

[5] Μοιάζουν λιγάκι με κινέζικα; Δεν πειράζει! Θα τα χρησιμοποιείς όπως ακριβώς τα δίνουμε και αργότερα θα καταλάβεις τι ακριβώς συμβαίνει. Όπως θα μάθουμε αργότερα, αυτό μπορεί να το δεις και ως: "`ios_base::in+ios_base::out`".

```
   int m;
   double y;

   sum = 0;  n = 0;
   a.open( "fltnum.txt", ios_base::in|ios_base::out );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
      a.seekg( 0 );
      a >> x;
      while ( !a.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         a >> x;
      } // while
   }
   a.close();
} // main
```

8.7 Μυστικά και Ψέματα

Το πρόγραμμα της §8.5 ήταν «ψεύτικο», όπως «μισοψεύτικα» είναι και τα προγράμματα της προηγούμενης παραγράφου! Δηλαδή; δεν δουλεύουν; Δουλεύουν μια χαρά, αλλά κανείς δεν θα καθήσει να δημιουργήσει ένα αρχείο-κείμενο τροφοδοτώντας ένα πρόγραμμα σαν τα παραπάνω! Και πώς γίνεται η δουλειά;

Ας πούμε ότι έχεις κάνει κάποιες μετρήσεις και έχεις μαζέψει μερικές σελίδες με νούμερα που θέλεις να τα επεξεργαστείς με κάποιο πρόγραμμα. Τι θα κάνεις; *Θα περάσεις τα νούμερα σε ένα αρχείο-κείμενο από κάποιον κειμενογράφο!* Αλλά, προσοχή: αν δεν χρησιμοποιήσεις έναν απλόν κειμενογράφο, σαν το Notepad των Windows (ή κάτι παρόμοιο) και προτιμήσεις έναν επεξεργαστή κειμένου, σαν το Word, φρόντισε *να φυλάξεις το αρχείο σου σε μορφή text* (ή ascii ή όπως αλλιώς το λέει).

Αυτός ο τρόπος έχει τα εξής πλεονεκτήματα:

- Δεν χρειάζεται να γράψεις πρόγραμμα.
- Έχεις δυνατότητα να κάνεις διορθώσεις[6].

Δηλαδή, τζάμπα μάθαμε να δημιουργούμε αρχεία με πρόγραμμα; Όχι βέβαια! Αλλά θα δημιουργούμε αρχεία με στοιχεία που θα δημιουργούνται από το πρόγραμμά μας ή θα

[6] Αν προσπαθήσεις να κάνεις το πρόγραμμά σου με δυνατότητα διόρθωσης, γίνεται πιο πολύπλοκο και… άσε καλύτερα.

προκύπτουν από επεξεργασία άλλων αρχείων. Όχι αρχεία που θα τα γεμίζουμε από το πληκτρολόγιο.

8.8 Πάγια Ρεύματα

Εκτός από τις κλάσεις *fstream*, *ifstream*, *ofstream*, που δηλώνονται στο `fstream`, υπάρχουν: η *istream* για ρεύματα από το αρχείο προς το πρόγραμμα και η *ostream* για ρεύματα από το πρόγραμμα προς το αρχείο.

Όλες οι διάλεκτοι της C++ παρέχουν τρία ρεύματα τα οποία μπορείς να υποθέσεις ότι δηλώνονται ως εξής:

```
istream  cin;
```

Αυτό είναι το **πάγιο ρεύμα εισόδου** (standard input) και, όπως έχουμε δει, αποκαθιστά ροή από το πληκτρολόγιο προς το πρόγραμμά μας.

```
ostream  cout;
```

Πρόκειται για το **πάγιο ρεύμα εξόδου** (standard output) και, όπως έχουμε δει, αποκαθιστά ροή από το πρόγραμμα προς την οθόνη. Τέλος:

```
ostream  cerr;
```

Αυτό είναι το ρεύμα όπου γράφονται τα μηνύματα λάθους (standard error)· συνήθως αντιστοιχεί στην οθόνη. Για την ίδια δουλειά υπάρχει και ένα τέταρτο πάγιο ρεύμα, το:

```
ostream  clog;
```

Αυτό είναι το ρεύμα προς το **πάγιο αρχείο καταγραφών** (standard log file)· συνήθως αντιστοιχεί στην οθόνη.

Μπορείς να θεωρήσεις ότι αυτά τα ρεύματα ανοίγονται αυτόματα, όταν αρχίζει η εκτέλεση του (κάθε) προγράμματός σου, το πρώτο για διάβασμα και τα άλλα για γράψιμο.

8.9 Αρχείο–Κείμενο: Άλλες Επεξεργασίες

Ο τρόπος που παρουσιάσαμε το αρχείο-κείμενο (text) είναι πρωτοφανής διεθνώς: κανείς δεν ξεκινάει να δουλεύει με τέτοια αρχεία διαβάζοντας και γράφοντας αριθμούς. Τώρα θα δούμε και τις επεξεργασίες που κάνει όλος ο κόσμος.

Όπως είπαμε, μπορούμε να σκεφτόμαστε ένα αρχείο-κείμενο ως εξής::

γραμμή
γραμμή
. . .
γραμμή, τέλος αρχείου

και *γραμμή*:

χαρακτήρας, ..., χαρακτήρας, τέλος γραμμής

ή

τέλος γραμμής

Η δεύτερη περίπτωση είναι η **κενή γραμμή**.

Για να χειριστείς αρχεία-κείμενα μπορείς να χρησιμοποιήσεις, εκτός από τους τελεστές "`<<`" και "`>>`" που ήδη είδαμε, τις μεθόδους *get()*, *peek()*, των κλάσεων *ifstream* και *fstream* και *put()* των κλάσεων *ofstream* και *fstream*.

Το τέλος της γραμμής τί είναι; Κάθε κειμενογράφος έχει ένα ειδικό σημάδι για να το σημειώνει. Όπου χρησιμοποιείται το σύνολο χαρακτήρων ASCII, συνήθως, το τέλος γραμμής σημειώνεται με δυο (μη-εκτυπώσιμους) χαρακτήρες:

- CR (Carriage Return) που είναι 13ος στον πίνακα (`'\r'` για τη C++) και
- LF (Line Feed), που είναι 10ος στο πίνακα (`'\n'` για τη C++).

Με αυτόν τον τρόπο αποθηκεύονται στο κείμενο οι λειτουργίες της οθόνης: όταν πιέζουμε το <enter> για να αλλάξουμε γραμμή: πρέπει να πάμε στην αρχή της γραμμής (CR) και να κατέβουμε στην επόμενη γραμμή (LF).

Όταν διαβάζουμε ένα αρχείο ως κείμενο (text mode) η C++ κάνει τα εξής: όταν βρίσκει ζευγάρι CR LF μας δίνει στο πρόγραμμα μόνο το LF (`'\n'`). Όταν γράφουμε κάνει το αντίστροφο: κάθε φορά που γράφουμε `'\n'`, βάζει CR LF.

Μερικοί κειμενογράφοι βάζουν για τέλος γραμμής τον έναν μόνον από τους δύο χαρακτήρες ή, με την αντίστροφη σειρά, LF CR.

Ας υποθέσουμε λοιπόν ότι θέλουμε να διαβάσουμε και να επεξεργαστούμε χαρακτήρα προς χαρακτήρα, ένα αρχείο text (με τη βοήθεια του `ifstream t`). Ήδη στην §4.4 είδαμε ότι αυτή η δουλειά δεν μπορεί να γίνει με τον τελεστή "`>>`" διότι αυτός «τρώει» τα κενά, τις αλλαγές γραμμής κλπ. Όπως εκεί, έτσι και εδώ θα χρησιμοποιήσουμε τη μέθοδο *get()*. Δίνοντας:

```
t.get( ch );
```

ζητάμε από το ρεύμα *t* να μας φέρει τον επόμενο χαρακτήρα από το αρχείο και να τον αποθηκεύσει ως τιμή της μεταβλητής *ch* (τύπου `char`).

Συνηθέστατα, όταν επεξεργαζόμαστε ένα αρχείο-κείμενο πρέπει να ξεχωρίζουμε τις γραμμές· εκτός από την επεξεργασία κάθε χαρακτήρα χωριστά, κάνουμε και άλλες επεξεργασίες συνολικά στη γραμμή. Πώς ανιχνεύουμε το *τέλος γραμμής*; Σύμφωνα με αυτά που είπαμε, τέλος γραμμής έχουμε όταν: "`ch == '\n'`".

Σημείωση: ▶
Συνήθως –και για διάφορα «σημάδια» τέλους γραμμής– η C++ θα σου επιστρέψει ένα απλό `'\n'`· τουλάχιστον όταν αυτό το σημάδι περιλαμβάνει και τον `'\n'`.◀

Μπορείς να σκεφτείς την επεξεργασία αρχείου text ως εξής:

```
while ( δεν τελείωσε το αρχείο )
{
   Διάβασε μια γραμμή
   Επεξεργάσου τη γραμμή
}
```

Το γενικό σχήμα επεξεργασίας ενός αρχείου text, που βλέπεις στο Πλ. 8.2, έχει περισσότερες λεπτομέρειες. Όπως βλέπεις, έχουμε δύο **while**, τη μια φωλιασμένη μέσα στην άλλη. Η εσωτερική **while** είναι αυτό που λέμε «Διάβασε γραμμή» αλλά έχει ενσωματωμένη την επεξεργασία του κάθε χαρακτήρα που διαβάζεται.

Πλαίσιο 8.2

Επεξεργασία Αρχείου Text

```
ifstream t;
    :
t.open( όνομα αρχείου );
t.get( ch );
while ( !t.eof() )
{
// προχώρα μέχρι το τέλος της γραμμής
   while ( !t.eof() && δεν βρήκαμε τέλος γραμμής )
   {
      Εντολές επεξεργασίας του ch
      t.get( ch );
   } // while (δεν βρήκαμε τέλος γραμμής…
   Εντολές επεξεργασίας γραμμής
   if ( !t.eof() ) πήγαινε στην αρχή της επόμενης γραμμής;
   t.get( ch );
}// while (!t.eof())
t.close();
```

Η εσωτερική **while** γράφεται:

```
while ( !t.eof() && ch != '\n' )
{
   Εντολές επεξεργασίας του ch
   t.get(ch);
} // while (δεν βρήκαμε τέλος γραμμής…
```

Για το πέρασμα στη νέα γραμμή αρκεί η `t.get(ch)` που υπάρχει στο τέλος της περιοχής της εξωτερικής **while**.

Όταν τελειώσει η εκτέλεση της εσωτερικής **while** θα συμβαίνει ένα από τα παρακάτω:

- `t.eof()` (τέλος αρχείου),
- `ch == '\n'` (τέλος γραμμής).

Φυσικά, θα προσπαθήσουμε να περάσουμε σε νέα γραμμή αν δεν έχουμε την πρώτη περίπτωση. Γι' αυτό, στο Πλ. 8.2 λέμε πιο προσεκτικά:

```
   if ( !t.eof() ) πήγαινε στην αρχή της επόμενης γραμμής;
```

Σημείωση: ▶

Με την ευκαιρία να πούμε ότι: Αν είμαστε στο τέλος του αρχείου, δηλαδή η `t.eof()` δίνει `true`, η ακριβώς προηγούμενη `t.get(ch)` έχει βάλει στη *ch* τιμή `char(-1)` (ή `unsigned char(255)`).◀

Όπως είπαμε και παραπάνω, στις κλάσεις *ofstream* και *fstream* υπάρχει η μέθοδος *put()*. Αν λοιπόν γράφουμε σε κάποιο αρχείο-κείμενο μέσω του ρεύματος *s* (ας πούμε, κλάσης *ofstream*), η

```
      s.put( ch );
```

γράφει την τιμή της *ch* (τύπου `char`) στο αρχείο.

Η μέθοδος *get()*, όπως ξέρουμε, υπάρχει και στην κλάση *istream* (την έχει το ρεύμα *cin*). Η μέθοδος *put()* υπάρχει και στην κλάση *ostream* (την έχει και το ρεύμα *cout*).

Στη συνέχεια βλέπεις ένα πολύ απλό πρόγραμμα: αντιγράφει ένα αρχείο-κείμενο (το text1.txt) σε ένα άλλο (το numdta.txt).

```
#include <fstream>
using namespace std;

int main()
{
   ifstream s( "text1.txt" );
   ofstream t( "numdta.txt" );
   char ch;

   s.get(ch);
   while ( !s.eof() )
   {  t.put(ch);  s.get(ch);  } // while
   t.close();  s.close();
} // main
```

Στην επόμενη παράγραφο θα δεις παραδείγματα με πιο πλήρη εφαρμογή του γενικού σχήματος.

8.10 Παραδείγματα

Θα δούμε τώρα δύο χαρακτηριστικά παραδείγματα επεξεργασίας αρχείου text. Και τα δύο θα επεξεργαστούν το εξής κείμενο, που είναι αντιγραμμένο από τον πρόλογο του C.A.R. Hoare στο (Hoare & Shepherdson 1985).

```
The strong connection between the formalization of
mathematical logic and the formalization of computer
programming languages was clearly recognized by Alan
Turing as early as 1947, when, in a talk on 20 February
to the London Mathematical Society, he reported his
```

```
expectation
   'that digital computing machines will eventually
   stimulate a considerable interest in symbolic logic
   and mathematical philosophy. The language in which
   one communicates with these machines, i.e. the language
   of instruction tables, forms a sort of symbolic logic'.
```

Χρησιμοποιώντας έναν κειμενογράφο, γράψε αυτό το κείμενο και φύλαξέ το σε ένα αρχείο με όνομα alturing.txt.

Παράδειγμα 1

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο που έχει το παραπάνω κείμενο και θα μετράει: πόσες γραμμές έχει, πόσα κεφαλαία γράμματα έχει και πόσα ψηφία έχει.

Το τι θα κάνουμε μας είναι γνωστό. Μας χρειάζονται τρεις μετρητές, για τα τρία μεγέθη που θα μετράμε, που αρχικά θα πρέπει να μηδενιστούν. Τα αποτελέσματά μας δεν μπορούν να βγουν πριν από το τέλος της επεξεργασίας του αρχείου. Το πρόγραμμά μας θα είναι περίπου:

```
int main()
{
   ifstream t;    // ρευμα απο το αρχειο με το κειμενο
   int  nRows;    // μετρητής γραμμών
   int  nUpCase;  // μετρητής κεφαλαίων
   int  nDigits;  // μετρητής ψηφίων
   char ch;       // χαρακτήρας που διαβάζουμε

   Μηδένισε τους μετρητές;
   Επεξεργάσου το αρχείο;
   Λέγε τα αποτελέσματα;
}
```

Η πρώτη «εντολή» υλοποιείται εύκολα:

```
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
```

το ίδιο και η τρίτη:

```
// Λέγε τα αποτελέσματα
   cout << " Διάβασα " << nRows << " γραμμές" << endl;
   cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
        << nDigits << " ψηφία" << endl;
```

Ας δούμε τώρα τι γίνεται με την «Επεξεργάσου το αρχείο». Αυτή θα ακολουθεί το γενικό σχήμα επεξεργασίας αρχείου text:

```
// Επεξεργάσου το αρχειο
   t.open( "alturing.txt" );
   t.get( ch );
   while ( !t.eof() )
   {
   // προχώρα μέχρι το τέλος της γραμμής
      while ( !t.eof() && ch == '\n' )
      {
         Εντολές επεξεργασίας του ch
         t.get( ch );
      } // while (δεν βρήκαμε τέλος γραμμής…
      Εντολές επεξεργασίας γραμμής
      if ( !t.eof() ) t.get( ch );
   }// while (!t.eof())
   t.close();
```

Το πρόβλημά μας τώρα είναι να δούμε τι θα είναι οι «*Εντολές επεξεργασίας του ch*» και οι «*Εντολές επεξεργασίας γραμμής*».

Οι «*Εντολές επεξεργασίας του ch*» τι θα κάνουν; Θα ελέγχουν τον κάθε χαρακτήρα που διαβάζεται και εφόσον είναι κεφαλαίο γράμμα ή ψηφίο θα αυξάνεται ο αντίστοιχος μετρητής. Μπορούμε να την υλοποιήσουμε με την:

```
         if ( isupper(ch) ) nUpCase = nUpCase + 1;
```

```
         else if ( isdigit(ch) )  nDigits = nDigits + 1;
```

Τις *isupper* και *isdigit* τις θυμάσαι; Αν όχι, γύρισε πίσω, στον Πίν. 4-3. Για να τις χρησιμοποιήσουμε θα πρέπει να βάλουμε `#include <cctype>`.

Τέλος, ας δούμε και τις «*Εντολές επεξεργασίας γραμμής*»: Κάθε φορά που τελειώνει μια γραμμή, θα πρέπει να αυξάνουμε το μετρητή γραμμών κατά 1:

```
      nRows = nRows + 1;
```

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cctype>

using namespace std;

int main()
{
   ifstream  t;  // ρεύμα απο το αρχείο με το κείμενο
   int  nRows;   // μετρητής γραμμών
   int  nUpCase; // μετρητής κεφαλαίων
   int  nDigits; // μετρητής ψηφίων
   char ch;      // χαρακτήρας που διαβάζουμε
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
// Επεξεργάσου το αρχείο
   t.open( "alturing.txt" );
   t.get( ch );
   while ( !t.eof() )
   {
      while ( !t.eof() && ch != '\n' )
      {
         if ( isupper(ch) ) nUpCase = nUpCase + 1;
         else if ( isdigit(ch) )  nDigits = nDigits + 1;
         t.get(ch);
      }
      nRows = nRows + 1;
      if ( !t.eof() )  t.get(ch);
   } // while (!t.eof())
   t.close();
// Λέγε τα αποτελέσματα
   cout << " Διάβασα " << nRows << " γραμμές" << endl;
   cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
        << nDigits << " ψηφία" << endl;
} // main
```

Το αποτέλεσμα που μας δίνει είναι:

```
 Διάβασα 11 γραμμές
 Μέτρησα 8 κεφαλαία γράμματα και 6 ψηφία
```

Παράδειγμα 2

Θέλουμε να επεξεργαστούμε ξανά το κείμενο που μετρήσαμε στο προηγούμενο παράδειγμα, αλλά με διαφορετικό στόχο: Θέλουμε να πάρουμε το ίδιο κείμενο γραμμένο με κεφαλαία.

Κατ' αρχάς θα πρέπει να δηλώσουμε τα δύο ρεύματα με τα οποία θα χειριστούμε τα αρχεία:

```
   ifstream  a;    // το αρχείο με το κείμενο
   ofstream  b;    // το αρχείο με τα κεφαλαία
```

και να τα ανοίξουμε:

```
   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
```

Την άσκ. 7-8 την έλυσες; Ζητούσαμε να γράψεις μια:

```
   // upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
   //           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
   char upCase( char ch )
```

Αν την έλυσες, τότε τα πράγματα είναι πολύ απλά:

- Οι «*Εντολές επεξεργασίας του ch*» θα μετατρέπουν το χαρακτήρα σε κεφαλαίο και θα τον γράφουν στο νέο αρχείο, δηλαδή:

  `co = upCase(ch);  b.put(co);`

- Οι «*Εντολές επεξεργασίας γραμμής*» δεν είναι τίποτε ιδιαίτερο: απλώς μετά το τέλος της γραμμής θα πρέπει να βάζουμε ένα `endl`.

  Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

   char upCase( char ch );

int main()
{
   ifstream  a;  // το αρχείο με το κείμενο
   ofstream  b;  // το αρχείο με τα κεφαλαία
   char ch, co;

   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
   a.get( ch );
   while ( !a.eof() )
   {
      while ( !a.eof() && (ch != '\n') )
      {  co = upCase(ch);  b.put(co);  a.get(ch);  }
      b << endl;
      if ( !a.eof() )  a.get( ch );
   }
   b.close();
   a.close();
} // main

   // upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
   //           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
   char upCase( char ch )
   {
      char fv;

      if ( islower(ch) )  fv = char( int(ch)-32 );
                    else  fv = ch;
      return fv;
   } // upCase
```

Παρατήρηση: ▶

Φυσικά, θα μπορούσαμε πάρουμε τη λύση πιο απλά:

- Θυμήσου ότι η C++ σου δίνει την *toupper* που κάνει αυτά που ζητούμε από την *upCase*.
- Πάρε το πρόγραμμα αντιγραφής της προηγούμενης παραγράφου και άλλαξε την:

```
{  t.put(ch);  s.get(ch);  }
```

σε:

```
{  b.put(toupper(ch));  a.get(ch);  }
```

Δηλαδή:

```
   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
   a.get( ch );
   while ( !a.eof() )
   {  b.put( toupper(ch) );  a.get( ch );  }
```

```
   b.close();  a.close(); ◀
```

🖘🖘🖘

8.11 Δουλεύοντας με Σιγουριά

Όταν κάνεις μια πράξη εισόδου/εξόδου (I/O) πολλά άσχημα μπορεί να συμβούν. Π.χ.

- να προσπαθήσεις να ανοίξεις για διάβασμα ένα αρχείο που δεν υπάρχει,
- να προσπαθήσεις να γράψεις σε μια δισκέτα που δεν έχει χώρο κ.ο.κ.

Η C++ σου δίνει τη δυνατότητα να ελέγχεις αν όλα πηγαίνουν καλά. Η μέθοδος *fail()* επιστρέφει `true` αν η προηγούμενη πράξη εισόδου/εξόδου απέτυχε αλλιώς `false`. Ας πούμε ότι έχεις δηλώσει:

```
   ifstream s;
```

και στη συνέχεια ζητάς:[7]

```
   s.open( "text1.txt" );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.txt" << endl;
   else // ok, προχώρα
   { // . . .
```

Τον ίδιο έλεγχο μπορείς να κάνεις και μετά την πράξη "`s.get(ch)`" ή την "`s >> x`".

Στη συνέχεια βλέπεις το πρόγραμμα της αντιγραφής αρχείου με πολλούς ελέγχους:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream s;
   ofstream t;
   char ch;

   s.open( "text1.txt" );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.txt" << endl;
   else // ok, προχώρα
   {
      t.open( "numdta.txt" );
      if ( t.fail() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το numdta.txt" << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
      {
         s.get(ch);
         while ( !s.fail() && !t.fail() )
         {  t.put( ch );  s.get(ch);  } // while
         if ( !s.eof() )
            cerr << "πρόβλημα κατά την αντιγραφή" << endl;
         t.close();
         s.close();
      } // if ( t.fail
   } // if ( s.fail
} // main
```

[7] Πάντως, σε πολλά βιβλία, θα δεις και έναν άλλον τρόπο: Αν δεις την *s* ως *συνθήκη*: αν το άνοιγμα του αρχείου έγινε επιτυχώς θα έχει τιμή που ερμηνεύεται ως `true`, αλλιώς θα έχει τιμή που ερμηνεύεται ως `false`. Έτσι, αντί για `if (s.fail())...` θα δεις το `if (!s)...`

Εξηγούμε και με λόγια το τι γίνεται:

- Αν δεν μπορέσαμε να ανοίξουμε το ρεύμα *s* από το αρχείο που διαβάζουμε δεν προχωρούμε.
- Αν τα καταφέραμε, ελέγχουμε αν ανοίξαμε το ρεύμα προς το αρχείο *t* που γράφουμε. Αν δεν άνοιξε, *κλείνουμε και το s* και σταματάμε.
- Αν όλα πήγαν καλά αντιγράφουμε με τη **while**. Πρόσεξε τη συνθήκη της που ελέγχει τα δύο ρεύματα. Η εκτέλεση των επαναλήψεων διακόπτεται αν εμφανιστεί πρόβλημα σε κάποιο από τα δύο.
- Αν τελειώσει το αρχείο η `s.fail()` θα δώσει `true` και οι επαναλήψεις θα σταματήσουν.
- Η **while** τελειώνει αν βρούμε πρόβλημα στο *s* ή στο *t*. Αν όμως το «πρόβλημα» είναι η *s.eof*`()` τότε όλα πήγαν καλά! Το μήνυμα "*πρόβλημα κατά την αντιγραφή*" εμφανίζεται μόνον αν βγήκαμε από τη **while** χωρίς να έχουμε *s.eof*`()`.

Ένας άλλος τρόπος για να δούμε αν το ρεύμα είναι ανοικτό είναι η κλήση της μεθόδου *is_open()*. Αυτή επιστρέφει τιμή `true` αν το ρεύμα είναι ανοικτό, αλλιώς `false`:

```
// . . .
   s.open( "text1.dta" );
   if ( !s.is_open() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.dta" << endl;
   else
   {
      t.open( "numdta.txt" );
      if ( !t.is_open() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το numdta.txt" << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
// . . .
```

8.12 Τρόποι Ανοίγματος (Ρεύματος) Αρχείου

Είπαμε παραπάνω ότι η *open()* δέχεται και μια δεύτερη παράμετρο που έχει να κάνει με τον τρόπο χρήσης του αρχείου και είδαμε ήδη τις εξής περιπτώσεις:

- "`ios_base::in`" για διάβασμα από αρχείο (δεν χρειάζεται αν το ρεύμα είναι *ifstream*).
- "`ios_base::out`" για γράψιμο σε αρχείο (δεν χρειάζεται αν το ρεύμα είναι *ofstream*).
- Ο συνδυασμός "`ios_base::in | ios_base::out`" για αρχείο που το ανοίγουμε για διάβασμα και για γράψιμο (δεν χρειάζεται αν το ρεύμα είναι *ofstream*).

Παρ' όλο που εδώ δεν έχουμε μεταβλητές τύπου `bool`, οι "`ios_base::in`" και "`ios_base::out`" ονομάζονται **σημαίες** (flags) της *open()*. Το γιατί θα το καταλάβεις αργότερα. Θα τις δεις ακόμη και ως *τιμές τύπου* (*ios_base*`::`)*openmode*.

Εκτός από αυτούς μπορείς να χρησιμοποιήσεις τα εξής (και συνδυασμούς τους):""

"`ios_base::binary`"

για αρχεία που έχουν το περιεχόμενό τους όχι σε μορφή κειμένου αλλά όπως είναι στην εσωτερική παράσταση. Θα τα δούμε στη συνέχεια.

"`ios_base::trunc`"

για να σβήσεις το περιεχόμενο του αρχείου με το άνοιγμά του (όπως γίνεται με το άνοιγμα ενός *ofstream*).

Ας πούμε ότι έχουμε ένα αρχείο –το `temp.txt`– και θέλουμε να το ανοίξουμε για γράψιμο και διάβασμα. Πριν από όλα όμως θέλουμε να σβήσουμε το παλιό του περιεχόμενο. Πώς μπορούμε να το κάνουμε;

- Με όσα ξέρουμε μέχρι τώρα: Το ανοίγουμε ως

```
   fstream temp( "temp.txt", ios_base::out );
```

```
temp.close();
```

για να σβήσουμε το αρχικό του περιεχόμενο και μετά το ανοίγουμε πάλι ως:

```
fstream temp( "temp.txt", ios_base::in|ios_base::out );
```

- Με χρήση της "`ios_base::trunc`": Το ανοίγουμε, μια φορά μόνον, ως

```
fstream temp( "temp.txt", ios_base::in|ios_base::out|ios_base::trunc );
```

"`ios_base::ate`"
με το άνοιγμα του αρχείου τοποθετείται στο τέλος του.
"`ios_base::app`"
το αρχείο δεν σβήνεται και κάθε γράψιμο γίνεται στο τέλος του.

Στη συνέχεια δίνουμε ένα παράδειγμα για να συγκρίνεις τις δύο τελευταίες. Χρησιμοποιούμε την εντολή "`temp.seekp(0)`" που σημαίνει «πήγαινε *να γράψεις* στην αρχή του αρχείου». Η μέθοδος *seekp()* είναι διαθέσιμη για ρεύματα *ofstream* και *fstream*.

Παράδειγμα

Έστω ότι έχουμε το αρχείο `temp.txt` με περιεχόμενο:

```
Ιανουάριος
Φεβρουάριος
Μάρτιος
Απρίλιος
```

Μετά το άνοιγμα:

```
fstream temp( "temp.txt",
              ios_base::in|ios_base::out|ios_base::ate );
```

και τις εντολές:

```
temp << "Μάιος" << endl;
temp.seekp( 0 );
temp << "Ιούνιος" << endl;
temp.close();
```

το περιεχόμενο του αρχείου γίνεται:

```
Ιούνιος
ς
Φεβρουάριος
Μάρτιος
Απρίλιος
Μάιος
```

Δηλαδή:

- Με το άνοιγμα πηγαίνουμε στο τέλος του αρχείου και γράφουμε "`Μάιος`".
- Μετά την `temp.seekp(0)`, το γράψιμο της λέξης "`Ιούνιος`" γίνεται στην αρχή του αρχείου.

Αν ανοίξουμε το *temp* με την:

```
fstream temp( "temp.txt", ios_base::out|ios_base::app );
```

οι ίδιες εντολές θα κάνουν το αρχείο:

```
Ιανουάριος
Φεβρουάριος
Μάρτιος
Απρίλιος
Μάιος
Ιούνιος
```

Δηλαδή, παρά την `temp.seekp(0)`, όταν έρχεται η στιγμή να γραφεί η λέξη "`Ιούνιος`" το γράψιμο γίνεται στο τέλος του αρχείου.

Σχετικώς με τη χρήση συνδυασμών από τέτοιες σημαίες να επισημάνουμε τα εξής:

- Όλοι οι επιτρεπτοί συνδυασμοί φαίνοντα στον Πιν. 8-1.

- Μπορείς να τους χρησιμοποιήσεις σε ρεύματα *fstream*, *ifstream*, *ofstream*. Αν κατά το άνοιγμα δεν βάλεις δεύτερη παράμετρο
  - Το *fstream* ανοίγει με `ios_base::in | ios_base::out`.
  - Το *ifstream* ανοίγει με `ios_base::in`.
  - Το *ofstream* ανοίγει με `ios_base::out`.
- Η ύπαρξη της `ios_base::out` στον συνδυασμό έχει ως αποτέλεσμα να σβηστεί το αρχείο, αν υπάρχει. Αυτό *αναιρείται* αν ο συνδυασμός περιέχει και κάποιαν από τις `ios_base::in`, `ios_base::app` ενώ *επιβάλλεται* αν περιέχει την `ios_base::trunc`.

8.13 Χειρισμός Αρχείων με τα Εργαλεία της C

Παρ' όλο που η C++ διαθέτει ένα πολύ πλούσιο και ευέλικτο σύστημα για τη διαχείριση αρχείων, θα δεις συχνά προγράμματα που χρησιμοποιούν τα αντίστοιχα εργαλεία της C. Θεωρούμε λοιπόν αναγκαίο να ρίξουμε μια ματιά στα εργαλειά αυτά αν και δεν θα τα χρησιμοποιούμε στη συνέχεια.

Στη συνέχεια βλέπεις (ξανά) το πρόγραμμα «*Μέση Τιμή 6*» με διαχείριση αρχείων όπως την κάνει η C:

```
0: // πρόγραμμα: Μέση Τιμή 6 C
1: #include <cstdio>
2: using namespace std;
3: int main()
4: {
```

Συνδυασμός τιμών `ios_base`					Ισοδύναμος ορμαθός `cstdio`
`binary`	`in`	`out`	`trunk`	`app`	x
		+			"w"
		+		+	"a"
				+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
	+	+		+	"a+"
	+			+	"a+"
+		+			"wb"
+		+		+	"ab"
+				+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"
+	+	+		+	"a+b"
+	+			+	"a+b"

Πίν. 8-1 (Από το πρότυπο της C++) Αντίστοιχοι τρόποι ανοίγματος αρχείων στη C++ και στη C.
Ένα παράδειγμα για το διάβασμά του: Η τελευταία γραμμή σημαίνει: αν στην *open* της C++ βάλεις
`ios_base::binary | ios_base::in | ios_base::app`
στη *fopen()* της C θα βάλεις `"a+b"`.

```
 5:    FILE* a;
 6:    FILE* b;
 7:
 8:    int n;         // Μετρητής όλων των στοιχείων
 9:    int selN;      // Μετρητής επιλεγόμενων στοιχείων
10:    double x;      // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
11:    double sum;    // Το άθροισμα όλων των στοιχείων
12:    double selSum; // Άθροισμα επιλεγόμενων στοιχείων
13:    double avrg;   // Μέση Αριθμητική Τιμή όλων των στοιχείων
14:    double selAvrg;
15:                   // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων
16:    sum = 0;  n = 0;
17:    selSum = 0;  selN = 0;
18:    a = fopen( "exp4.dta", "r" );
19:    fscanf( a, "%lf", &x );
20:    while ( !feof(a) )
21:    {
22:       n = n + 1;              // Αυτά γίνονται για
23:       sum = sum + x;          // όλους τους αριθμούς
24:       if ( 0 < x && x <= 10 ) // 'Ελεγχος - Επιλογή
25:       {
26:          selSum = selSum + x;  // Αυτά γίνονται μόνο για
27:          selN = selN + 1;      // τους επιλεγόμενους αριθμούς
28:       } // if
29:       fscanf( a, "%lf", &x );
30:    } // while
31:    fclose( a );
32:    b = fopen( "report.txt", "w" );
33:    fprintf( b, " Διάβασα %d αριθμούς\n", n );
34:    if ( n > 0 )
35:    {
36:       avrg = sum / n;
37:       fprintf( b, " ΑΘΡΟΙΣΜΑ = %lf   <x> = %lf\n",
38:                   sum, avrg );
39:    }
40:    fprintf( b, " Διάλεξα %d αριθμούς ε (0,10]\n", selN );
41:    if ( selN > 0 )
42:    {
43:       selAvrg = selSum/selN;
44:       fprintf( b, " ΑΘΡΟΙΣΜΑ = %lf  <x> = %lf\n",
45:                   selSum, selAvrg );
46:    }
47:    fclose( b );
48: } // main
```

Ας δούμε τα κρίσιμα σημεία:

- Στη γρ. 1 έχουμε "`#include <cstdio>`" αντί της γνωστής μας "`#include <fstream>`". Θυμίσου ότι το ίδιο είχαμε κάνει και όταν χρησιμοποιούσαμε τις `printf` (§1.12) και `scanf` (§2.12).
- Στις γρ. 5-6 βλέπεις τις δηλώσεις:

```
FILE* a;
FILE* b;
```

Με αυτά τα δύο βέλη θα χειριστούμε τα δύο ρεύματα. Αργότερα θα πάρουμε μια ιδέα για το τι μπορεί να είναι ο τύπος "`FILE`".

- Στη γρ. 18 ανοίγουμε το πρώτο ρεύμα με την "`a = fopen( "exp4.dta", "r" )`" που δίνει τιμή στο βέλος *a*. Η *fopen()* είναι μια συνάρτηση που παίρνει δύο ορίσματα:
  - το όνομα του αρχείου (στην περίπτωσή μας `"exp4.dta"`) και
  - τον τρόπο ανοίγματος (στην περίπτωσή μας `"r"`).

  ανοίγει το ρεύμα και επιστρέφει τιμή τύπου *FILE*`*`. Αν αποτύχει το άνοιγμα επιστρέφει 0 (`NULL`). Στον πίνακα 8.1 μπορείς να δεις πώς γράφεται ένας τρόπος ανοίγματος της C

αντίστοιχος τρόπου ανοίγματος της C++. Στην περίπτωσή μας (`"r"`) έχουμε το ισοδύναμο του "`ios_base::in`".

- Στη γρ. 19 κάνουμε το πρώτο διάβασμα: "`fscanf( a, "%lf", &x )`". Αυτή είναι ίδια με τη `scanf` (§2.12) αλλά έχει ένα επι πλέον όρισμα, στην αρχή, που καθορίζει το ρεύμα (στη `scanf` το ρεύμα είναι το `stdin`). Στη γρ. 29 διαβάζουμε ξανά με τον ίδιο ακριβώς τρόπο.
- Στη γρ. 20, στη συνθήκη της **while**, ελέγχουμε για τέλος αρχείου καλώντας την *feof*. Όπως βλέπεις, της δίνουμε ως όρισμα το βέλος *a*, που καθορίζει το ρεύμα.
- Στη γρ. 31 κλείνουμε τπ ρεύμα με την "`fclose( a )`".
- Στη γρ. 32 ανοίγουμε ρεύμα για να γράψουμε το αρχείο `report.txt` με την "`b = fopen( "report.txt", "w" )`". Ο τρόπος χρήσης του ρεύματος (`"w"`) είναι ισοδύναμος του "`ios_base::out`".
- Στις γρ. 33, 37, 40, 44 γράφουμε στο αρχείου χρησιμοποιώντας την `fprintf`. Αυτή είναι σαν την `printf` αλλά έχει ένα επί πλέον όρισμα στην αρχή που καθορίζει το ρεύμα.
- Θα μπορούσαμε να είχαμε ανοίξει τα ρεύματα με τις δηλώσεις των *a*, *b*; Βεβαίως, έτσι:

```
FILE* a( fopen("exp4.dta", "r") );
FILE* b( fopen("report.txt", "w") );
```

Στη συνέχεια (ξανα)δίνουμε και το πρόγραμμα αντιγραφής αρχείου με χρήση των εργαλείων της C:

```
 0: #include <cstdio>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    FILE* s;
 6:    FILE* t;
 7:    int ch, res( 0 );
 8:
 9:    s = fopen( "text1.dta", "r" );
10:    if ( !s )
11:       fprintf( stderr,
12:                "δεν μπορώ να ανοίξω το αρχείο text1.dta\n" );
13:    else
14:    {
15:       t = fopen( "numdta.txt", "w" );
16:       if ( !t )
17:       {
18:          fclose( s );
19:          fprintf( stderr,
20:                   "δεν μπορώ να δημιουργήσω το numdta.txt\n" );
21:       }
22:       else // ok και τα δύο αρχεία ανοικτά
23:       {
24:          ch = getc( s );
25:          while ( ch != EOF && res != EOF )
26:          {  res = putc( ch, t );  ch = getc( s );  } // while
27:          if ( !feof(s) )
28:             fprintf( stderr,
29:                      "πρόβλημα κατά την αντιγραφή\n" );
30:          fclose( t );
31:          fclose( s );
32:       } // if ( t.fail
33:    } // if ( s.fail
34: } // main
```

Εδώ, τα καινούρια πράγματα είναι στις γρ. 24, 25, 26:

- Η συνάρτηση *getc()* παίρνει ένα όρισμα –το βέλος που καθορίζει το ρεύμα (που διαβάζουμε)– και μας επιστρέφει τον επόμενο χαρακτήρα από το αρχείο αλλά σε τύπο `int`. Αν υπάρξει πρόβλημα επιστρέφει `EOF` (-1).

Μέθοδος / Τελεστής	Κλάση	Τι κάνει, πού την είδαμε
`>>`	`ifstream`	Φέρνει τιμή από το αρχείο σε μεταβλητή. §8.3, 2.3, 4.5, 4.6
`<<`	`ofstream`	Γράφει τιμές στο αρχείο. §8.4, 1.2, 1.11, 4.4.1
`clear`	`ifstream, ofstream`	Ανατάσσει το ρεύμα μετά από *eof* ή άλλη αποτυχία κάποιας λειτουργίας. §8.3.2
`close`	`ifstream, ofstream`	Αποσυνδέει ένα ρεύμα από το αρχείο. §8.3, 8.4
`endl`	`ofstream`	Γράφει τέλος γραμμής και αντιγράφει τον ενταμιευτή στο αρχείο. §8.4
`eof`	`ifstream`	Μας πληροφορεί αν φτάσαμε στο τέλος του αρχείου. §8.3, 8.3.1, 8.11
`fail`	`ifstream, ofstream`	Επιστρέφει `true` αν η προηγούμενη λειτουργία στο ρεύμα απέτυχε. §8.11
`get`	`ifstream`	Φέρνει ένα χαρακτήρα από το αρχείο στο όρισμά της. §8.9, 8.12, 4.5
`open`	`ifstream, ofstream`	Συνδέει το ρεύμα με το αρχείο. §8.1, 8.3, 8.4
`is_open`	`ifstream, ofstream`	Επιστρέφει τιμή `true` αν το ρεύμα είναι ανοικτό, αλλιώς `false`, §8.11.
`peek`	`ifstream`	Δίνει το χαρακτήρα που θα διαβάσει η επόμενη *get()*. §8.9
`put`	`ofstream`	Γράφει το όρισμά της στο αρχείο. §8.9
`seekg(0)`	`ifstream`	Το ρεύμα ετοιμάζεται να διαβάσει από την αρχή του αρχείου. §8.6
`seekp(0)`	`ofstream`	Το ρεύμα ετοιμάζεται να γράψει στην αρχή του αρχείου. §8.12

Πίν. 8-2 Οι μέθοδοι για τη διαχείριση αρχείων με ρεύματα που μάθαμε. Όλες οι μέθοδοι υπάρχουν και στην κλάση `fstream`. Οι μέθοδοι της `ifstream` υπάρχουν και στην `istream` (τύπος του `cin`). Οι μέθοδοι της `ofstream` υπάρχουν και στην `ostream` (τύπος των `cout`, `cerr`, `clog`).

- Η *putc()* παίρνει δύο ορίσματα: τον χαρακτήρα που θέλουμε να γράψουμε στο αρχείο και το βέλος για το ρεύμα. Αν γράψει τον χαρακτήρα τον επιστρέφει και ως τιμή. Αν αποτύχει επιστρέφει `EOF`. Όπως βλέπεις, στη *res* κρατούμε την τιμή που επιστρέφει κάθε φορά η *putc()*. Όσο δεν παίρνουμε `EOF` ούτε στη *res* ούτε στη *ch* ("`while (ch != EOF && res != EOF)`") συνεχίζουμε τη διαδικασία αντιγραφής.

8.14 Σύνοψη

Στο κεφάλαιο αυτό μάθαμε πώς να χειριζόμαστε αρχεία-κείμενα (text) είτε έχουν αριθμητικά δεδομένα είτε τα θεωρούμε ως αρχεία χαρακτήρων. Τα εργαλεία μας είναι ρεύματα των κλάσεων *ifstream*, *ofstream* και *fstream* και οι μέθοδοί τους που βλέπεις στον Πίν. 8-2.

Θα επαναλάβουμε ακόμη και μερικές βασικές αρχές που μάθαμε στο κεφάλαιο αυτό.

- ***Για να διαβάσουμε (ή να γράψουμε) το ν-οστό στοιχείο ενός σειριακού αρχείου πρέπει να περάσουμε κατ' ανάγκη τις ν −1 προηγούμενες τιμές.***
- ***Η επεξεργασία κάθε σειριακού αρχείου ξεκινάει πάντα από την αρχή του.***
- ***Πριν επεξεργαστούμε μια τιμή που (υποτίθεται ότι) διαβάσαμε από ένα αρχείο ελέγχουμε την eof() ή τη fail() για να βεβαιωθούμε ότι η ανάγνωση έγινε κανονικώς.***

Λύσε οπωσδήποτε τις Ασκ. 8-5, 8-6 (και αργότερα την 9-13) και δες προσεκτικά τις λύσεις που προτείνουμε. Αναφέρονται στην *ενημέρωση* (μεταβολή, εισαγωγή, διαγραφή στοιχείων) σειριακού αρχείου με βάση την αρχή:

- ***Για να ενημερώσουμε ένα σειριακό αρχείο το αντιγράφουμε σε ένα άλλο και κατά την αντιγραφή κάνουμε τις αλλαγές που θέλουμε.***

Ασκήσεις

Α Ομάδα

8-1 Γράψε ένα πρόγραμμα που θα αντιγράφει ένα αρχείο text στην οθόνη. Αριστερά από κάθε γραμμή θα γράφεται ο αριθμός της, ξεκινώντας από 1.

8-2 Σε ένα αρχείο-κείμενο, με όνομα στο δίσκο real.txt, έχουμε πραγματικούς αριθμούς. Γράψε ένα πρόγραμμα που θα μετράει:

- πόσοι αριθμοί μεταξύ 4.5 και 5.5 υπάρχουν στο αρχείο,
- πόσες φορές υπάρχει η τιμή 5.0

Ακόμα, θα δίνει τα ποσοστά των παραπάνω αριθμών σε σχέση με τον συνολικό αριθμό τιμών του αρχείου.

8-3 Σε δύο αρχεία text, με ονόματα 'mydata.txt', 'moredata.txt' και το ίδιο πλήθος γραμμών, υπάρχουν αριθμοί, ένας σε κάθε γραμμή. Γράψε πρόγραμμα που θα τα διαβάζει και θα δημιουργεί ένα νέο αρχείο text, με όνομα "comb.txt", που θα έχει το ίδιο πλήθος γραμμών με κάθε ένα από τα αρχικά και στη ν-οστή γραμμή του θα έχει τέσσερις αριθμούς:

- αυτόν που υπάρχει στην ν-οστή γραμμή του πρώτου αρχείου,
- αυτόν που υπάρχει στην ν-οστή γραμμή του δεύτερου αρχείου,
- το άθροισμα των δύο προηγούμενων,
- το γινόμενο των δύο προηγούμενων.

8-4 Σε ένα αρχείο text, με όνομα protmet.txt, υπάρχουν πρότυπες μετρήσεις από τον έλεγχο μιας συσκευής –ένας αριθμός σε κάθε γραμμή. Το αρχείο testmet.txt είναι ίδιο και έχει τον ίδιο αριθμό γραμμών με το πρώτο αλλά όχι και τους ίδιους αριθμούς· περιέχει δοκιμαστικές μετρήσεις που κάναμε στη συσκευή για να δούμε αν είναι εντάξει. Έστω α ο αριθμός που υπάρχει στην k γραμμή του πρώτου αρχείου και β αυτός που υπάρχει στην k γραμμή του δεύτερου αρχείου. Η συσκευή μας είναι εντάξει αν

για όλα τα k έχουμε: $0.95\alpha \le \beta \le 1.05\alpha$

δηλαδή, αν υπάρχουν αποκλίσεις που δεν υπερβαίνουν το 5%.

Γράψε πρόγραμμα που θα διαβάζει τα δυο αρχεία και

- για κάθε γραμμή (k) που βρίσκει απόκλιση μεγαλύτερη από 5% θα μας τυπώνει το k, το α και το β,
- αν δεν βρει απόκλιση μεγαλύτερη από 5% θα μας τυπώνει στο τέλος το μήνυμα "ΣΥΣΚΕΥΗ ΕΝΤΑΞΕΙ".

Β Ομάδα

8-5 (Ενημέρωση – μεταβολή στοχείων) Σε ένα αρχείο-κείμενο, με όνομα στο δίσκο idid.txt, έχουμε σε κάθε γραμμή δύο πραγματικούς αριθμούς που παριστάνουν μια τάση (ο πρώτος) και ένα ρεύμα (ο δεύτερος), όπως μετρήθηκαν στο εργαστήριο. Αλλά οι τιμές του ρεύματος διαβάστηκαν λαθεμένα στο αμπερόμετρο. Στις πρώτες πέντε (5) γραμμές η τιμή που υπάρχει στο αρχείο είναι 1000πλάσια της πραγματικής και στις επόμενες 11 είναι 10πλάσια της πραγματικής. Οι υπόλοιπες γραμμές είναι σωστές.

Γράψε λοιπόν μια συνάρτηση που θα παίρνει τη θέση της γραμμής (k) και το ρεύμα (i) και θα μας δίνει το σωστό ρεύμα:

```
double correctI( int k, double i )
```

Θέλουμε να αλλάξουμε κάθε τιμή ρεύματος (*i*) του αρχείου σε *correctI*(*k*, *i*), όπου *k* (1η, 2η, 3η κλπ) η γραμμή του αρχείου όπου βρίσκεται η *i*. Αυτό θα το κάνεις με ένα πρόγραμμα που θα αντιγράφει μια προς μια τις γραμμές του idid.txt σε ένα άλλο, με το όνομα midid.txt, αλλά, κάθε φορά θα αλλάζει την τιμή ρεύματος (*i*) σε *correctI*(*k*, *i*) με τη συνάρτηση που έγραψες όπως είπαμε παραπάνω.

8-6 (Ενημέρωση – διαγραφή στοιχείων) Έστω τώρα ότι θέλουμε να σβήσουμε από το αρχείο idid.όλες τις γραμμές με λαθεμένα στοιχεία, δηλαδή τις πρώτες δεκαέξη (16) γραμμές.

Και πάλι, θα πρέπει να γράψεις ένα πρόγραμμα που να δημιουργεί ένα νέο αρχείο με όνομα dідid.txt, όπου θα αντιγράψει μόνον τις γραμμές με σωστά στοιχεία (από τη 17η και μετά). Φυσικά, τα στοιχεία που θα διαγράψεις δεν θα πάνε στα σκουπίδια: θα αντιγραφούν σε ένα άλλο αρχείο με όνομα xidid.txt.

8-7 Ένα αρχείο text, με όνομα στο δίσκο misthoi.txt, είναι γραμμένο ως εξής: Σε κάθε γραμμή έχει τρεις αριθμούς. Ο πρώτος είναι ο βασικός μισθός ενός υπαλλήλου, ο δεύτερος ο αριθμός χρόνων υπηρεσίας του ίδιου υπαλλήλου και ο τρίτος ο αριθμός των παιδιών του. Το πλήθος των στοιχείων είναι άγνωστο. Το αρχείο θα χρησιμοποιηθεί για τη μισθοδοσία των υπαλλήλων ενός οργανισμού.

Το περιεχόμενο του αρχείου δεν έχει ελεγχθεί. Θα γράψεις ένα πρόγραμμα που θα ελέγχει τα εξής:

- αν 800 ≤ βασικός μισθός ≤ 3500,
- αν 0 ≤ χρόνια υπηρεσίας ≤ 35,
- αν 0 ≤ αριθμός παιδιών ≤ 12.

Για κάθε λάθος που θα βρίσκει, θα πρέπει να τυπώνει τον αριθμό του υπαλλήλου, δηλαδή τη σειρά που έχουν τα στοιχεία του στο αρχείο και τη λαθεμένη τιμή. Π.χ.:

```
 7ΟΣ ΥΠΑΛΛΗΛΟΣ. ΜΙΣΘΟΣ: 35650
 7ΟΣ ΥΠΑΛΛΗΛΟΣ. ΑΡΙΘΜΟΣ ΠΑΙΔΙΩΝ: -4
10ΟΣ ΥΠΑΛΛΗΛΟΣ. ΧΡΟΝΙΑ ΥΠΗΡΕΣΙΑΣ: 51
23ΟΣ ΥΠΑΛΛΗΛΟΣ. ΜΙΣΘΟΣ: 955000
```

8-8 Με βάση τα στοιχεία του διορθωμένου αρχείου, dmisthoi.txt, της προηγούμενης άσκησης, υπολογίζονται οι αποδοχές των υπαλλήλων ως εξής:

- για κάθε χρόνο υπηρεσίας ο μισθός προσαυξάνεται κατά 2% πάνω στο βασικό,
- αν ο εργαζόμενος έχει μέχρι τρία (3) παιδιά ο μισθός προσαυξάνεται κατά 30 € για κάθε παιδί, αλλιώς, αν έχει περισσότερα από τέσσερα (4) ο μισθός προσαυξάνεται κατά 50 € για κάθε παιδί.

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει και θα τυπώνει την κατάσταση αποδοχών των υπαλλήλων.

Γ Ομάδα

8-9 Δίνεται ένα αρχείο text. Γράψε ένα πρόγραμμα που θα αντιγράφει το αρχείο σε ένα άλλο, αφού πρώτα πετάξει τα περιττά κενά του πρώτου. Δηλαδή, όπου υπάρχουν στο παλιό ένα ή περισσότερα συνεχόμενα κενά, στο νέο θα υπάρχει μόνο ένα.

8-10 Γράψε πρόγραμμα που θα διαβάζει ένα αρχείο κείμενο που περιέχει πρόγραμμα C++ και θα το αντιγράφει στην οθόνη με αρίθμηση των γραμμών. Π.χ. θα μας δείχνει:

```
1: #include <iostream>
2: #include <math>
3:
4: int main()
5: {
```

```
6:    int a[100], b[25];
       . . .
```

8-11 Γράψε πρόγραμμα που θα διαβάζει ένα αρχείο κείμενο που περιέχει πρόγραμμα C++ και θα το αντιγράφει σε δύο αρχεία ως εξής:

α) Το ένα θα έχει μόνον τις γραμμές που περιέχουν μόνον σχόλια, δηλαδή γραμμές που ξεκινούν με `//`.

β) Το άλλο θα έχει όλες τις υπόλοιπες γραμμές.

8-12 Τεχνικοί, που σχεδίασαν και κατασκεύασαν μια συσκευή, πιστεύουν ότι δύο βασικά της μεγέθη q και u συνδεόνται με τον απλό νόμο: $q = \eta\mu u$. Μια άλλη ομάδα τεχνικών πιστεύει ότι η εξάρτηση είναι πιο πολύπλοκη: $q = g(u)$, όπου g συνάρτηση περιοδική, με περίοδο 2π, που στο $[-\pi,\pi]$ ορίζεται ως εξής:

$$g(u) = \begin{cases} \frac{4}{\pi^2}u(u+\pi), & -\pi \le u \le 0 \\ -\frac{4}{\pi^2}u(u-\pi), & 0 \le u \le \pi \end{cases}$$

Γράψε συνάρτηση που να υλοποιεί τη g σε C++.

Σε ένα αρχείο text, με όνομα στο δίσκο irnapr.txt, υπάρχουν στοιχεία μετρήσεων της συσκευής. Σε κάθε γραμμή του αρχείου υπάρχουν δύο πραγματικοί αριθμοί: ο πρώτος αντιπροσωπεύει τιμή από μέτρηση της u, ενώ ο δεύτερος είναι η αντίστοιχη τιμή της q. Θέλουμε, με επεξεργασία αυτών των τιμών, να δούμε ποιο από τα δύο μοντέλα είναι καλύτερο.

Ας πούμε λοιπόν ότι διαβάζουμε μια γραμμή του αρχείου και αποθηκεύουμε τις τιμές που διαβάσαμε στις u_k και q_k. Υπολογίζουμε τα $\eta\mu u_k$ και $g(u_k)$ και θα λέμε:

- ότι «είναι καλύτερο το μοντέλο της g» αν $|g(u_k) - q_k| < |\eta\mu u_k - q_k|$·

αλλιώς θα λέμε

- ότι «καλύτερο είναι το μοντέλο του ημιτόνου».

Θέλουμε ένα πρόγραμμα που θα διαβάζει ολόκληρο το αρχείο και θα μας λέει στο τέλος:

- πόσες φορές (για πόσες γραμμές του αρχείου) ήταν καλύτερο το μοντέλο του ημιτόνου και πόσες φορές το μοντέλο της g,
- τις μέσες τιμές των $|g(u_k)-q_k| = \frac{1}{N}\sum_{k=1}^{N}|g(u_k)-q_k|$ και $|\eta\mu u_k-q_k| = \frac{1}{N}\sum_{k=1}^{N}|\eta\mu u_k-q_k|$ όπου N το πλήθος των γραμμών του αρχείου.

Ακόμη, το πρόγραμμα θα αντιγράφει το αρχείο σε δύο άλλα αρχεία text με ονόματα στο δίσκο girnapr.txt, sirnapr.txt, ως εξής:

- Αν για μια γραμμή είναι καλύτερο το μοντέλο της g τότε η γραμμή θα αντιγράφεται «εμπλουτισμένη» στο girnapr.txt. Για την ακρίβεια θα γράφονται: u_k, q_k, $g(u_k)$, $|g(u_k) - q_k|$.
- Αν για μια γραμμή είναι καλύτερο το μοντέλο του ημιτόνου τότε μια «εμπλουτισμένη» γραμμή με τα u_k, q_k, $\eta\mu u_k$, $|\eta\mu u_k - q_k|$ θα γράφεται στο αρχείο sirnapr.txt.

8-13 Για το παρόν πρόβλημα θεωρούμε ως λέξη μια ακολουθία χαρακτήρων που είναι γράμματα του λατινικού αλφαβήτου. Π.χ. στο κείμενο

```
if <x, f, x'> is a state transition
```

υπάρχουν 8 λέξεις με μήκη 2, 1, 1, 1, 2, 1, 5, 10 αντιστοίχως.

Μας δίνεται το αρχείο-κείμενο alturing.txt. Γράψε πρόγραμμα που θα το διαβάζει και θα μας λέει α) το πλήθος των λέξεων που έχει β) το μέγιστο μήκος λέξης που συνάντησε.

κεφάλαιο

9

Πίνακες Ι

Ο στόχος μας σε αυτό το κεφάλαιο:

Συχνά έχουμε πολλές μεταβλητές με τις ίδιες ιδιότητες που πρέπει να υποστούν την ίδια επεξεργασία. Θα μάθεις πώς να τις οργανώνεις σε *πίνακες* ώστε να τις χειρίζεσαι με τις γνωστές εντολές επανάληψης.

Προσδοκώμενα αποτελέσματα:

Ο πίνακας είναι ένα πολύ καλό εργαλείο για πάρα πολλές χρήσεις. Εδώ θα δεις μερικές επεξεργασίες πινάκων που θα σου είναι χρήσιμες πολύ συχνά.

Έννοιες κλειδιά:

- *πίνακας*
- *στοιχείο πίνακα, δείκτης*
- *αναζήτηση τιμής σε πίνακα*
- *ταξινόμηση πίνακα*
- *συγχώνευση πινάκων*
- *αποδοτικότητα αλγόριθμου*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει 100 πραγματικούς αριθμούς x_1, x_2, ..., x_{100} και

α) θα υπολογίζει και θα τυπώνει τη Μέση Αριθμητική Τιμή τους <*x*>,

β) θα υπολογίζει και θα τυπώνει τις τιμές της συνάρτησης: $e^{\frac{<x>-x}{<x>}}$ για τους αριθμούς αυτούς. Δηλαδή τα $y_k = e^{\frac{<x>-x_k}{<x>}}$.

Ε, αυτό το λύσαμε στο προηγούμενο κεφάλαιο και μάλιστα όχι για 100 αλλά για όσους αριθμούς και να έχουμε! Γράφουμε σε ένα αρχείο τους αριθμούς όταν πληκτρολογούνται και όταν έχουμε υπολογίσει την <*x*>, διαβάζουμε το αρχείο από την αρχή.

Πόσο καλή είναι αυτή η λύση; Όχι και πολύ καλή. Δεν είναι παραδεκτό να στέλνουμε 100 αριθμούς σε βοηθητική μνήμη και να τους ξαναδιαβάζουμε για να χρησιμοποιηθούν δεύτερη φορά από το ίδιο πρόγραμμα. Πρέπει να τους κρατήσουμε στην κύρια μνήμη όσο τους χρειαζόμαστε.

Αλλά, σύμφωνα με όσα ξέρουμε, θα πρέπει δηλώσουμε 100 μεταβλητές: *x*1, *x*2, ...,*x*100 τύπου `double` και να γράψουμε 100 εντολές:

```
cin >> x1; cin >> x2; ... ; cin >> x100;
```

για να τους διαβάσουμε.

Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει έναν πιο άνετο τρόπο για να λύσουμε το πρόβλημά μας. Το εργαλείο που θα χρησιμοποιήσουμε λέγεται **πίνακας** και θα το γνωρίσουμε στις παραγράφους που ακολουθούν.

9.1 Πίνακες Στοιχείων

Η έννοια του **πίνακα** (array, matrix) είναι γνωστή από τα μαθηματικά. Στη C++ ο πίνακας δεδομένων αποτελείται από έναν αριθμό *ομοειδών στοιχείων*, δηλ. από *στοιχεία του ίδιου τύπου*. Ο τύπος αυτός καθορίζεται όταν δηλώνουμε τη μεταβλητή - πίνακα.

Τα ομοειδή στοιχεία, που αποτελούν τον πίνακα, έχουν πάντα ένα κοινό όνομα, π.χ. τα pinA, pinB και flNm στο παρακάτω παράδειγμα δήλωσης πινάκων:

```
int    pinA [ 20 ];
double pinB [ 50 ];
char   flNm [ 120 ];
```

Η πρώτη δήλωση του παραδείγματος καθορίζει ότι ο πίνακας pinA αποτελείται από 20 στοιχεία που το καθένα έχει τα χαρακτηριστικά μιας μεταβλητής τύπου `int`, η δεύτερη γραμμή δηλώνει ότι ο πίνακας pinB αποτελείται από 50 στοιχεία που έχουν χαρακτηριστικά μεταβλητής τύπου `double` και η τρίτη ότι ο πίνακας flNm αποτελείται από 120 στοιχεία τύπου `char`.

Σχηματικά μπορούμε να βλέπουμε, π.χ. τον πίνακα pinA, σαν ένα μονοδιάστατο πίνακα που αποτελείται από μια γραμμή και 20 στήλες ή από μια στήλη και 20 γραμμές. Το Σχ. 9-1 δείχνει την πρώτη περίπτωση δομής.

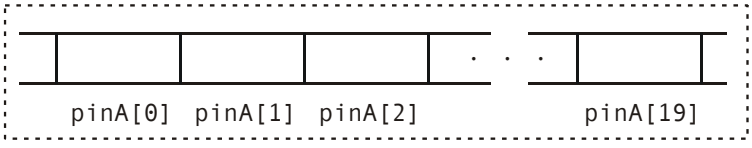
Τα **στοιχεία** (elements) ή οι **συνιστώσες** (components) του πίνακα σημειώνονται πάντα με το κοινό όνομα της μεταβλητής-πίνακα που συνοδεύεται από τον λεγόμενο **δείκτη** (index).

- ***Αν ο πίνακας έχει δηλωθεί με n στοιχεία, οι επιτρεπτές τιμές του δείκτη είναι όλοι οι φυσικοί από 0 μέχρι n – 1.***

Ο δείκτης του στοιχείου πίνακα δίνεται πάντα μέσα σε αγκύλες, όπως δείχνουν τα παρακάτω παραδείγματα:

```
pinA[1],  pinA[5],  pinA[15]
pinB[2],  pinB[10],  pinB[49]
```

Έτσι, τα συμβολικά ονόματα της πρώτης γραμμής καθορίζουν το δεύτερο, το έκτο και το δέκατο έκτο στοιχείο του πίνακα pinA. Ενώ τα ονόματα της δεύτερης γραμμής καθορίζουν το τρίτο, το ενδέκατο και το τελευταίο στοιχείο του πίνακα pinB. Τέλος το pinA`[`*k*`]` καθορί-



Σχ. 9-1 Σχηματική παράσταση πίνακα και των στοιχείων του.

ζει γενικώς το (*k*+1)-οστό στοιχείο του πίνακα pinA, όπου *k* = 0, 1,... 19. Αντιστοίχως το pinB`[`*k*`]` καθορίζει το (*k*+1)-οστό στοιχείο του πίνακα pinB, όπου *k* = 0, 1,... 49.

Το γενικό συντακτικό δήλωσης του μονοδιάστατου πίνακα μπορεί να δοθεί ως εξής:

τύπος, *αναγνωριστικό*, "`[`", *πλήθος*, "`]`"

- Ο *τύπος*, που λέγεται και **τύπος συνιστωσών** (component type), μπορεί να είναι οποιοσδήποτε τύπος, π.χ. `int`, `double`, `char`, `bool`, απαριθμητός κλπ.
- Το *αναγνωριστικό* είναι σαν και αυτά των μεταβλητών.
- Το *πλήθος* είναι μια παράσταση που μπορεί να περιέχει σταθερές, μεταβλητές που έχουν ήδη τιμή και συναρτήσεις, αν φυσικά η δήλωσή μας βρίσκεται μέσα στην εμβέλειά τους. Η τιμή της πρέπει να είναι ακέραιου τύπου και μεγαλύτερη από 0. Το *πλήθος* μπορεί να παραλείπεται.

Κοίταξε τα παρακάτω παραδείγματα:

```
const int N( 50 ), N1( 63 ), N2( 114 );

typedef int  PinAk[ N+1 ];
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };

PinAk    p, q, r;
int      l[ N+1 ], m[ N2-N1+1 ];
bool     signal[ (N1+1)/4 ];
ifstream keimena[ 4 ];
DecDigit bv[ N/2+1 ];
PinAk    mat[ 11 ];
```

Κατ' αρχάς ορίζουμε έναν τύπο, τον *PinAk*· κάθε αντικείμενο αυτού του τύπου, όπως οι p, q, r που δηλώνουμε παρακάτω, είναι ένας πίνακας με 51 (= *N* + 1) στοιχεία τύπου `int`, που αριθμούνται από 0 μέχρι 50, π.χ. p`[`0`]`, p`[`1`]`, …, p`[`50`]`.

Στη συνέχεια αντιγράφουμε τον ορισμό του τύπου *DecDigit* από το Κεφ. 4 και παρακάτω δηλώνουμε τον πίνακα bv που έχει 26 (= *N*/2 + 1) στοιχεία τύπου *DecDigit*.

Ο πίνακας `l` έχει 51 στοιχεία τύπου `int`· θα μπορούσαμε να είχαμε δηλώσει "`PinAk l`". Ο πίνακας m έχει 52 στοιχεία τύπου `int`.

Ο keimena είναι ένας πίνακας που κάθε στοιχείο του είναι ένα ρεύμα τύπου *ifstream*.

Τέλος, στην τελευταία δήλωση, ο mat έχει 11 στοιχεία που το καθένα τους είναι ένας πίνακας τύπου *PinAk*· είναι ένας πίνακας πινάκων ή δισδιάστατος πίνακας. Με τέτοιους πίνακες θα ασχοληθούμε αργότερα.

Μαζί με τη δήλωση ενός πίνακα μπορείς να δώσεις και αρχικές τιμές στα στοιχεία του. Π.χ.:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Στην περίπτωση αυτή δεν χρειάζεται να δηλώσεις και το πλήθος (ο μεταγλωττιστής ξέρει να μετράει). Θα μπορούσαμε να γράψουμε:

```
int monthLength[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Αν δηλώσεις το πλήθος των στοιχείων και στη λίστα αρχικών τιμών βάλεις λιγότερες τιμές τότε οι υπόλοιπες θεωρούνται μηδέν. Π.χ. η

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31 };
```

είναι ισοδύναμη με την:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31, 0, 0, 0, 0 };
```

Όπως καταλαβαίνεις, αν θέλεις να βάλεις αρχική τιμή "`0`" σε όλα τα στοιχεία ενός πίνακα αρκεί να γράψεις ένα "`0`" ανάμεσα στα άγκιστρα:

```
double x[ 100 ] = { 0.0 };
```

Ωραία λοιπόν, δηλώσαμε τον πίνακα. Τώρα τι κάνουμε; Με ολόκληρο τον πίνακα δεν μπορούμε να κάνουμε και πολλά πράγματα, αλλά:

- ***Κάθε συνιστώσα ενός πίνακα έχει όλα τα χαρακτηριστικά μιας μεταβλητής του τύπου συνιστωσών και μπορείς να τη χειριστείς όπως και κάθε μεταβλητή αυτού του τύπου.***

Έτσι, π.χ. το `l[17]` είναι μεταβλητή τύπου `int`. Μπορούμε λοιπόν να της δώσουμε τιμή με μια εντολή εκχώρησης:

```
l[17] = 3215;
```

ή να τη διαβάσουμε από το πληκτρολόγιο:

```
cin >> l[17];
```

Αν η `l[17]` έχει τιμή μπορούμε να τη χρησιμοποιήσουμε σε παραστάσεις:

```
x = l[17] / 5 + 100;
```

να γράψουμε την τιμή της στην οθόνη ή σε κάποιο αρχείο:

```
cout << l[17] << endl;
```

Το signal`[11]` είναι μεταβλητή τύπου `bool` –μπορεί να πάρει τιμές `true` ή `false`, π.χ.:

```
signal[11] = ( l[17] >= x );
if ( signal[11] && x > 1000 )  { ...
```

Όπως είδαμε και πιο πάνω, με τον δείκτη ξεχωρίζουμε τις συνιστώσες ενός πίνακα. Ο δείκτης είναι, γενικά, μια παράσταση. Η τιμή αυτής της παράστασης θα πρέπει να είναι μη αρνητική και μικρότερη από το πλήθος στοιχείων που έχουμε δηλώσει. Μετά τη δήλωση:

```
int  metr[ 26 ];
```

μπορείς να δώσεις:

```
metr[0] = -16;
```

αλλά, δεν μπορείς να δώσεις:

```
metr[27] = 24;
```

ούτε

```
metr[-5] = 33;
```

Αυτό είναι λάθος που, δυστυχώς, *δεν θα το δείξει* ο μεταγλωττιστής. Ακόμη, αν *c* `==` 22, μπορείς να δώσεις:

```
metr[c+1] = c - 17;
```

αλλά, αν *c* `==` 25, η παραπάνω εντολή είναι λάθος: `metr[26]` *δεν υπάρχει!* Αυτό το λάθος φυσικά δεν μπορεί να το εντοπίσει ο μεταγλωττιστής, αλλά, δυστυχώς, μπορεί να μη φανεί ούτε κατά την εκτέλεση. Θα οδηγήσει, πιθανότατα, σε παράλογα αποτελέσματα (μπορεί και να «παγώσει» ο υπολογιστής σου).

- ***Για πίνακα με n στοιχεία είναι υποχρέωση του προγραμματιστή να περιορίζει την τιμή του δείκτη στις επιτρεπτές τιμές (από 0 μέχρι n – 1).***

Όπως φαίνεται από τα παραπάνω παραδείγματα, ο δείκτης του πίνακα μπορεί να είναι, γενικά, μια παράσταση. Έχουμε λοιπόν τη δυνατότητα πρόσβασης στα διάφορα στοιχεία ενός πίνακα, δίνοντας την κατάλληλη τιμή στο δείκτη. Αν θέλουμε να χειριστούμε όλα τα στοιχεία ενός πίνακα, μπορούμε να το πετύχουμε με μια επανάληψη όπου η μεταβαλλόμενη τιμή του δείκτη μας δίνει το ένα στοιχείο μετά το άλλο.

Η C++ δεν μας επιτρέπει διαχείριση ολόκληρου του πίνακα. Για παράδειγμα η εκχώρηση πρέπει να γίνεται στοιχείο προς στοιχείο. Ας πούμε ότι έχουμε δηλώσει:

```
int  a[5], b[5];
```

και κάποτε θέλουμε να εκχωρήσουμε την τιμή που έχει εκείνη τη στιγμή ο a στον b. *Δεν επιτρέπεται να δώσουμε:* `b = a`· θα πρέπει να δώσουμε την εντολή:

```
   for ( k = 0; k <= 4; k = k+1 )  b[k] = a[k];
```

Τώρα, μπορούμε να ξαναγυρίσουμε στο πρόβλημα της εισαγωγικής παραγράφου και να δούμε μια πιο όμορφη λύση.

Παράδειγμα

Ας έλθουμε τώρα στο πρόβλημα που δώσαμε στην εισαγωγή. Στο πρόγραμμα-λύση που γράψαμε στο προηγούμενο κεφάλαιο η *x* ήταν μια απλή μεταβλητή. Κάθε φορά που διαβάζαμε μια νέα τιμή, η προηγούμενη τιμή της *x* χάνονταν, αφού φυσικά την είχαμε φυλάξει στο αρχείο.

Τώρα θα δηλώσουμε τη x ως:

```
   double   x[ N ];
```

Σε κάθε εκτέλεση της περιοχής της (πρώτης) **for** θα δίνουμε τιμή και σε διαφορετικό στοιχείο του *x*. Όταν υπολογίσουμε τη <*x*>, οι 100 αριθμοί που αποθηκεύτηκαν στον x θα είναι στη διάθεσή μας για δεύτερη χρήση:

```
// πρόγραμμα: Μέση Τιμή 1+
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const int N( 100 );// το πλήθος των στοιχείων
   double x[ N ];     // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;        // Το μερικό (τρέχον) άθροισμα.
                      // Στο τέλος έχει το ολικό άθροισμα.
   double avrg;       // Μέση Αριθμητική Τιμή των x (<x>)
   int m;
   double y;

   sum = 0;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x[m];
   // πρόσθεσέ την στη sum
      sum = sum + x[m];
   } // for
   avrg = sum / N;
   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 3 );
   cout << " ΑΘΡΟΙΣΜΑ = " << sum
        << "  <x> = " << avrg << endl;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      y = exp( (avrg - x[m])/avrg );
      cout << " x[";
      cout.width(3);  cout << m << "] = ";
      cout.width(6);  cout << x[m] << "   y[";
      cout.width(3);  cout << m << "] = " ;
      cout.width(6);  cout << y << endl;
   } // for
} // main
```

Το παραπάνω πρόγραμμα έχει δύο **for**, όπου η μεταβλητή *m* παίζει διπλό ρόλο. Χρησιμοποιήθηκε:

α) ως μεταβλητή ελέγχου της **for** και

β) ως δείκτης πίνακα (που διατρέχει τις τιμές: 0 . . *N*-1).

Αυτή η λύση είναι σαφώς καλύτερη από αυτήν της προηγούμενης παραγράφου. Αυτό το πρόγραμμα είναι φανερά πιο γρήγορο από το πρώτο και αυτό διότι αποφεύγει τη χρονοβόρα διαδικασία γραψίματος/διαβάσματος από τη βοηθητική μνήμη.

Στη σύνθεση του προγράμματος, μπορεί κανείς να παρασυρθεί από τον τύπο $y_k = e^{\frac{<x>-x_k}{<x>}}$ και να δηλώσει το y ως πίνακα. «Το x είναι πίνακας, βλέπουμε και δείκτη στο y... Βάλε έναν πίνακα, να τελειώνουμε!» Αυτό που μας ζητάνε, όμως, είναι να υπολογίσουμε και να τυπώσουμε τα y_k. Μετά δεν μας χρειάζονται πια. Το y θα μπορεί να είναι απλή μεταβλητή τύπου `double`. Άλλωστε, «ζωγραφίζουμε» και τα αποτελέσματά μας έτσι ώστε να βγάζουν δείκτες, για να ευχαριστηθεί και ο χρήστης:

```
ΑΘΡΟΙΣΜΑ = 1772.217  <x> = 17.722
x[  0] = -2.217   y[  0] =  3.081
x[  1] = 58.900   y[  1] =  0.098
x[  2] = 35.850   y[  2] =  0.360
x[  3] = -0.879   y[  3] =  2.857
     . . .
```

Το κράτημα 100 θέσεων `double` στην κύρια μνήμη θα ήταν καθαρή σπατάλη.

Παρατήρηση: ▶

Γλυτώσαμε λοιπόν από το αρχείο! Έτσι, νομίζεις. Πληκτρολογείς 100 αριθμούς για να δοκιμάσεις το πρόγραμμα και βλέπεις ότι έχει λάθος. Μετά, στη δεύτερη δοκιμή, τι κάνεις; Τους ξαναπληκτρολογείς; Δεν είμαστε καλά! Διόρθωσε παιδί μου το πρόγραμμα να τους φυλάξει σε ένα αρχείο! ◀

Αυτό που λέμε στην παρατήρηση θα το πούμε ως γενική συμβουλή:

- ***Όταν πληκτρολογούμε δεδομένα, που πιθανότατα θα ξαναχρειαστούμε, τα φυλάγουμε οπωσδήποτε σε αρχείο.***

9.2 Συνηθισμένες Δουλειές με Πίνακες

Ας δούμε τώρα μερικές πολύ συνηθισμένες δουλειές που γίνονται με πίνακες. Για τα παρακάτω κομμάτια προγράμματος υποθέτουμε ότι έχουμε δηλώσει:

```
const int N(...);
double x[N];
```

Φυσικά, ο τύπος μπορεί να μην είναι `double`, αλλά `int`, `char` κλπ.

Πριν προχωρήσουμε να τονίσουμε το εξής: Κατ' αρχήν, μια μεταβλητή που προορίζεται να παίζει ρόλο δείκτη θα πρέπει να δηλώνεται με τύπο `unsigned int`. Όπως θα δεις όμως, συνήθως, θα τη δηλώνουμε με τύπο `int`. Από αμέλεια; Και από αμέλεια, αλλά όπως θα δεις σε κάποια από τα παραδείγματα, μερικές φορές δίνουμε σε τέτοιες μεταβλητές κάποια αρνητική τιμή=φρουρό.

9.2.1 Εισαγωγή Στοιχείων

Στο παράδειγμα της προηγούμενης παραγράφου είδαμε πώς διαβάζουμε τις τιμές των στοιχείων ενός πίνακα από το πληκτρολόγιο. Το ξαναγράφουμε λίγο αλλαγμένο και χωρίς την άθροιση:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   cout << "Δώσε το " << m << "ο στοιχείο: ";
   cin >> x[m];
} // for
```

Με αυτόν τον τρόπο το πρόγραμμα παίρνει τις τιμές όλων των στοιχείων με τη σειρά, καθοδηγώντας το χρήστη. Παράδειγμα εκτέλεσης:

```
Δώσε το 0ο στοιχείο: 5.5
Δώσε το 1ο στοιχείο: 6.3
Δώσε το 2ο στοιχείο: 4
```

. . .

Ας πούμε τώρα ότι ο χρήστης δεν θέλει να δώσει όλα τα στοιχεία αλλά θέλει να σταματήσει με φρουρό (9999). Τι κάνουμε στην περίπτωση αυτή;

```
m = 0;  cin >> x[m];
while ( x[m] != 9999 && m < N-1 )
{
   m = m + 1;  cin >> x[m];
} // while
```

Όπως βλέπεις, η *m* ξεκινάει από 0 και αυξάνεται κατά 1 όσο έχουμε `m < N-1`. Άρα αποκλείεται να πάρει τιμή μεγαλύτερη από *N*–1. Όταν τελειώσει η εκτέλεση της **while** θα έχουμε: `x[m] == 9999 || m >= N-1`.

- Αν έχουμε `x[m] == 9999` τότε στο x[*m*] έχουμε τον φρουρό, που δεν είναι τιμή προς επεξεργασία. Άρα θα έχουν διαβαστεί *m* τιμές στα x[0] .. x[*m*–1].
- Αν έχουμε *m* `>=` *N*–1 τότε θα έχουμε για την ακρίβεια *m* `==` *N*–1 (αφού η *m* παίρνει πρώτη τιμή 0 και αυξάνεται κατά 1 κάθε φορά). Στην περίπτωση αυτή θα έχουν διαβαστεί ήδη *N* (= *m* + 1) τιμές στα x[0] .. x[*N*–1].

Έτσι, αν θέλουμε να έχουμε το πλήθος των τιμών που διαβάστηκαν στη μεταβλητή *count*, θα πρέπει να βάλουμε:

```
if ( x[m] == 9999 )  count = m;
                else  count = m + 1;
```

Αν θέλουμε να δώσουμε στον χρήστη το δικαίωμα να δίνει και τον δείκτη του στοιχείου θα πρέπει να σκεφτούμε διαφορετικά:

```
cout << "Δώσε τη θέση του στοιχείου: ";  cin >> m;
cout << "Δώσε το " << m << "ο στοιχείο: ";  cin >> x[m];
```

Τώρα όμως πρέπει να σκεφτούμε και άλλα δύο πράγματα:

- Μπορεί να πάρουμε παράνομη τιμή του δείκτη –στην περίπτωσή μας μικρότερη από 0 ή μεγαλύτερη από *N*–1. Θα πρέπει λοιπόν να ελέγχουμε την τιμή του *m* πριν διαβάσουμε το x[*m*].
- Δεν είναι καθόλου σίγουρο ότι ο χρήστης θα δώσει τιμές για όλα τα στοιχεία. Δεν μπορούμε λοιπόν να δουλεύουμε με τη **for**. Θα πρέπει να δουλεύουμε μάλλον με φρουρό (π.χ. -1 ή κάποιον άλλον αρνητικό στο *m*).

```
cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
     << ") (-1 για ΤΕΛΟΣ): ";
cin >> m;
while ( m != -1 )
{
   if ( 0<= m && m <= N-1 )
   {
      cout << "Δώσε το " << m << "ο στοιχείο: ";
      cin >> x[m];
   }
   else
      cout << "*** Λάθος θέση ***" << endl;
   cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
        << ") (-1 για ΤΕΛΟΣ): ";
   cin >> m;
} // while
```

Παράδειγμα εκτέλεσης:

```
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 7
Δώσε το 7ο στοιχείο: -15.7
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 3
Δώσε το 3ο στοιχείο: -3
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 11
*** Λάθος θέση ***
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 1
Δώσε το 1ο στοιχείο: 6.3
```

. . .

Παρατήρηση: ▶

Αν γράφεις ένα «πραγματικό» πρόγραμμα θα πρέπει να προσέχεις αυτά τα «7ο στοιχείο» ή «στοιχείο στη θέση 7» που είναι σίγουρο ότι θα προκαλέσουν σύγχυση και λάθη (το πρώτο στοιχείο βρίσκεται στη θέση 0;!). Αν ο χρήστης είναι κάποιος που ξέρει προγραμματισμό μπορείς να αναφερθείς στο «*στοιχείο με δείκτη* 2»· αν δεν ξέρει, θα πρέπει να βρεις τη γλώσσα που καταλαβαίνει και να του μιλήσεις με αυτήν.◀

Μερικές φορές, όταν ο πίνακας είναι μικρός, μπορεί να θελήσεις να διαβάσεις όλα τα στοιχεία του από μια γραμμή. Αυτό είναι απλό: από τον 1ο τρόπο αφαιρούμε τα μηνύματα:

```
for ( m = 0; m <= N-1; m = m+1 )  cin >> x[m];
```

Σε αυτό μπορείς να απαντήσεις –αν π.χ. $N = 5$– με τη γραμμή:

```
5.5   6.3  4.0   -3  5.1<Enter>
```

9.2.2 Εισαγωγή Στοιχείων από Αρχείο

Τα παραπάνω ισχύουν και για εισαγωγή τιμών από αρχείο. Αλλά:

- Πρέπει να προσθέσουμε ελέγχους για τέλος αρχείου.
- Δεν έχουν νόημα τα μηνύματα προς το χρήστη.

Έστω ότι έχουμε $N = 9$, το ρεύμα:

```
ifstream t( "text1.dta" );
```

και το αρχείο text1.dta με περιεχόμενο:

```
-7   -15      8
    14       33
-8  16   114
    375
```

Αν είναι σίγουρο ότι στο αρχείο υπάρχουν N τιμές, μπορείς να διαβάσεις έτσι:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   t >> x[m];
} // for
```

Αλλά, επειδή οι τιμές μπορεί να είναι λιγότερες ή περισσότερες, πιο σίγουρος είναι ο παρακάτω τρόπος (αντικαταστήσαμε τον έλεγχο του φρουρού με τη `!t.eof()`):

```
m = 0;  t >> x[m];
while ( !t.eof() && m < N-1 )
{
   m = m + 1;  t >> x[m];
} // while
if ( t.eof() )  count = m;
          else  count = m+1;
```

Ας πούμε τώρα ότι $N = 9$ και το αρχείο arrval.txt έχει τα εξής:

```
7   -15.7
3    -3
8     3.75
2     4.0
6     0
11    6.3
4     5.1
5   -13
0     5.5
```

Σε κάθε γραμμή, ο πρώτος αριθμός δείχνει τον δείκτη στον πίνακα και η δεύτερη την τιμή του αντίστοιχου στοιχείου. Διαβάζουμε τις τιμές των στοιχείων, με το 2ο τρόπο, ως εξής:

```
ifstream t( "arrval.txt" );
double y;
int    rNum;
```

```
rNum = 0;
t >> m;
while ( !t.eof() )
{
   rNum = rNum + 1;
   if ( 0<= m && m <= N-1 )
      t >> x[m];
   else
   {
      cout << "Λάθος τιμή δείκτη στη γραμμή " << rNum << endl;
      t >> y;
   }
   t >> m;
} // while
t.close();
```

Αυτές οι εντολές θα δώσουν τιμές σε όλα τα στοιχεία του πίνακα εκτός από το x[1]. Και θα μας δώσουν και ένα μήνυμα λάθους:

```
Λάθος τιμή δείκτη στη γραμμή 6
```

διότι εκεί δίνεται τιμή δείκτη 11.

9.2.3 Γράψιμο Στοιχείων

Το γράψιμο είναι πιο απλό. Ας πούμε ότι θέλουμε να γράψουμε τα στοιχεία του x σε μια γραμμή της οθόνης. Δίνουμε:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << "  ";
cout << endl;
```

και παίρνουμε:

```
5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
```

Καταλαβαίνεις βέβαια ότι το `<< "  "` είναι απαραίτητο για να διαχωρίζονται οι τιμές. Να τι θα βγει αν το παραλείψεις:

```
5.56.34-35.1-130-15.73.75
```

Αν έχεις το *t* (*ofstream*) συνδεμένο με κάποιο αρχείο-κείμενο τότε οι:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << "  ";
t << endl;
```

γράφουν τις τιμές σε μια γραμμή του αρχείου.

Αν θέλεις να βάζεις μια τιμή σε κάθε γραμμή θα πρέπει να δίνεις `endl` για κάθε στοιχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << endl;
```

και για αρχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << endl;
```

Δες ακόμη πώς γράψαμε τις τιμές των στοιχείων του x στο παράδ. της §9.2.

Ας πούμε όμως ότι έχουμε:

```
const int N( 50 );
int z[N];
```

και θέλεις να τυπώσεις τον πίνακα z όπως φαίνεται στο Σχ. 9-2. Πώς γίνεται αυτό;

Όπως φαίνεται έχουμε να τυπώσουμε 10 γραμμές, αριθμημένες (από την 1η στήλη) από 0 μέχρι 9. Αυτό γίνεται ως εξής:

```
for ( r = 0; r <= 9; r =r + 1 )
{
   Γράψε τη γραμμή r
} // for (r = ...
```

Τώρα ας δούμε πώς θα γράψουμε τη γραμμή *r*. Ας πάρουμε για παράδειγμα τις γραμμές 0 και 1· τι περιέχουν; Τα:

```
 0   35     10    9     20  872     30  232     40  347
 1   18     11   34     21    0     31  667     41   61
 2   15     12   21     22   23     32  139     42  753
 3   80     13   57     23  -34     33  -45     43   73
 4   10     14  239     24  -32     34   -9     44    6
 5   40     15  909     25   56     35  -89     45  -37
 6   23     16  213     26  787     36   34     46   43
 7  789     17  576     27  146     37  576     47  -99
 8  563     18  903     28  589     38  122     48  344
 9    1     19  239     29  568     39   99     49  572
```

Σχ. 9-2 Εκτύπωση των τιμών των στοιχείων του πίνακα z. Πριν από κάθε τιμή γράφεται ο δείκτης. Π.χ. 0 35 σημαίνει ότι το στοιχείο z[0] έχει τιμη 35.

0, z[0], 10, z[10], 20, z[20], 30, z[30], 40, z[40] (γραμμή 0)
1, z[1], 11, z[11], 21, z[21], 31, z[31], 41, z[41] (γραμμή 1)
Καταλαβαίνεις λοιπόν ότι μπορούμε να γράψουμε τη γραμμή r ως εξής:

```
    cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
         << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
```

(φυσικά θα πρέπει να βάλουμε και κενά για να διαχωρίζονται οι τιμές.)

Να λοιπόν η λύση στο πρόβλημά μας:

```
for ( r = 0; r <= 9; r = r + 1 )
{
   cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
        << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
} // for (r =…
```

Αλλά μπορούμε να τη γράψουμε πιο κομψά: Η γραμμή δεν βγαίνει με **for**; Για δες αυτήν[1]:

```
    for ( c = 0; c <= 40; c = c + 10 )
        cout << (r + c) << z[r+c];
```

Να ολόκληρο το κομμάτι μαζί με τον καθορισμό πλάτους πεδίου για να ξεχωρίζουν οι τιμές μεταξύ τους:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 40; c = c + 10)
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << z[r+c];
   } // for (c =...
   cout << endl;
} // for (r =...
```

9.2.4 Απλοί Υπολογισμοί

Οι υπολογισμοί αθροίσματος και γινομένου στοιχείων πίνακα γίνονται όπως ξέρουμε:

```
sum = 0;
for ( m = 0; m <= N-1; m = m + 1 )
   sum = sum + x[m];// άθροισμα
```

[1] Άλλοι προτιμούν να γράψουν το εξής:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 4; c = c + 1)
   {
      cout.width(7);  cout << (r + c*10);
      cout.width(5);  cout << z[r+c*10];
   } // for (c =...
   cout << endl;
} // for (r =...
```

Έτσι έχεις το πλεονέκτημα το *c* να έχει πάντοτε ως τιμή τον αριθμό της στήλης (0 .. 4).

και

```
product = 1;
for ( m = 0; m <= N-1; m = m + 1 )
   product = product * x[m];// γινόμενο
```

Για να βρούμε το μέγιστο (ελάχιστο) δεν χρειάζεται να καταφύγουμε στην τεχνική της απίθανα μικρής (μεγάλης) αρχικής τιμής: Θεωρούμε αρχικώς ότι μέγιστη (ελάχιστη) είναι η τιμή του στοιχείου με δείκτη 0:

```
maxNdx = 0;  xMax = x[maxNdx];
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > xMax )
   {  maxNdx = m;  xMax = x[m];  }
} // for (m …
```

Στη *xMax* κρατάμε τη μέγιστη από τις τιμές των στοιχείων του πίνακα και στη *maxNdx* τον δείκτη του στοιχείου στον πίνακα. Πρόσεξε όμως το εξής: το βασικό είναι να βρούμε τον *δείκτη* του μεγίστου (*maxNdx*). Αν τον ξέρουμε, αφού έχουμε όλες τις τιμές στη μνήμη, παίρνουμε και τη μέγιστη τιμή:

```
maxNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > x[maxNdx] )  maxNdx = m;
} // for (m …
xMax = x[maxNdx];
```

Παρομοίως δουλεύουμε και για το ελάχιστο:

```
minNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] < x[minNdx] )  minNdx = m;
} // for (m …
xmin = x[minNdx];
```

9.3 Παράμετρος – Πίνακας

Κάτι που θα σου είναι πολύ χρήσιμο είναι το να γράφεις συναρτήσεις με παραμέτρους - πίνακες.

Ας δούμε πώς γίνεται αυτό με ένα παράδειγμα. Η:

```
double vectorSum( double x[], int n )
{
   int m;
   double sum( 0 );

   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

υπολογίζει και επιστρέφει το άθροισμα των τιμών των στοιχείων ενός πίνακα με στοιχεία τύπου `double`.

Θέλήσαμε να κάνουμε τη συνάρτησή μας γενική, ώστε να δουλεύει με οποιονδήποτε πίνακα με στοιχεία τύπου `double`. Βάλαμε τον x ως παράμετρο στη συνάρτηση *αλλά δεν βάλαμε πλήθος στοιχείων*. Το πλήθος *n* των στοιχείων του πίνακα το περνάμε με άλλη παράμετρο.

Με το ίδιο τρόπο μπορούμε να γράψουμε και μια συνάρτηση που θα επιστρέφει τον δείκτη του στοιχείου με τη μέγιστη από τις τιμές ενός πίνακα.

```
int maxNdx( int x[], int n )
{
   int m;
```

```
    int mxp( 0 );

    for ( m = 1; m <= n-1; m = m + 1 )
    {
       if (x[m] > x[mxp])  mxp = m;
    } // for (m ...
    return mxp;
} // maxNdx
```

Ας πούμε ότι σε ένα πρόγραμμα έχουμε:

```
const int N( 9 ), Nd2( N/2 );

double x[N] = { 5.5, 6.3, 4, -3, 5.1, -13, 0, -15.7, 3.75 },
       u[Nd2] = { 5.5, 4, 5.1, 0 };
int ix[N] = { 5, 6, 4, -3, 1, -13, 0, -15, 3 },
    iu[Nd2] = { 5, 4, 1, 0 };
```

και δίνουμε:

```
cout << " Σx: " << vectorSum( x, N ) << endl;
cout << " Σu: " << vectorSum( u, Nd2 ) << endl;

cout << " max(ix): " << ix[maxNdx(ix, N)]
     << " στη θεση: " << maxNdx( ix, N ) << endl;
cout << " max(iu): " << iu[maxNdx(iu, Nd2)]
     << " στη θεση: " << maxNdx( iu, Nd2 ) << endl;
```

Αποτέλεσμα:

```
Σx: -7.05
Σu: 14.6
max(ix): 6 στη θεση: 1
max(iu): 5 στη θεση: 0
```

Πρόσεξε τις κλήσεις των συναρτήσεων: Παίρνουμε το άθροισμα των τιμών των στοιχείων

- του x με τη `vectorSum(x, N)` και
- του u με τη `vectorSum(u, Nd2)`.

Ακόμη:

- Η `maxNdx(ix, N)` μας δίνει τον *δείκτη* του μέγιστου στοιχείου του ix ενώ παίρνουμε την *τιμή* αυτού του στοιχείου με την `ix[maxNdx(ix, N)]`.
- Για τον iu δείκτης μέγιστου στοιχείου: `maxNdx(iu, Nd2)` και *τιμή* μέγιστου στοιχείου: `iu[maxNdx(iu, Nd2)]`.

Αλλά, δυστυχώς, η *vectorSum* δέχεται μόνο πίνακες με στοιχεία τύπου `double`. Αν θέλουμε το άθροισμα των στοιχείων του ix θα πρέπει να γράψουμε άλλη συνάρτηση. Παρομοίως, η *maxNdx* δέχεται πίνακα τύπου `int` μόνον. Αργότερα θα δούμε ότι μπορούμε να βάλουμε τον μεταγλωττιστή να γράφει τις συναρτήσεις που μας ενδιαφέρουν.

Και γιατί βάλαμε τον τύπο της *maxNdx* `int` και όχι `unsigned int`; Αυτό θα ήταν το σωστό, αλλά διάβασε αυτά που ακολουθούν.

Μερικές φορές μας ενδιαφέρει να κάνουμε μια επεξεργασία σε ένα κομμάτι του πίνακα μόνον· ας πούμε να βρούμε το άθροισμα των πρώτων πέντε στοιχείων του x. Θα μπορούσαμε να καλέσουμε τη *vectorSum* ως εξής:

```
q = vectorSum( x, 5 );
```

Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη *vectorSum* και για την περίπτωση αυτή.

Πάντως αυτές οι χρήσεις έχουν ένα πρόβλημα: μπερδεύουν τη δομή του πίνακα με την περιοχή επεξεργασίας, Αυτό μπορεί να οδηγήσει σε μερικά πολύ «δύσκολα» προγραμματιστικά λάθη. Μέχρι να γίνεις μεγάλος/η προγραμματιστής/τρια καλύτερα να διαχωρίζεις αυτά τα στοιχεία στη συνάρτηση. Π.χ.:

```
double vectorSum( double x[], int n, int from, int upto )
```

```
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Φυσικά, θα πρέπει να έχουμε:

$$0 \le from \le upto \le n-1$$

Και αν δεν ισχύει η συνθήκη τι κάνουμε; Κατ' αρχήν θα πρέπει να καλέσουμε την *exit()*. Επειδή όμως δεν είναι σπάνιο να καλούμε μια συνάρτηση σαν αυτήν με $n == 0$ ή με $upto < from$ και να περιμένουμε από τη συνάρτηση τιμή 0, θα γράψουμε:

```
   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 ) sum = sum + x[m];
   }
```

Παρομοίως γράφουμε:

```
int maxNdx( int x[], int n, int from, int upto )
{
   int m;
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      mxp = from;
      for ( m = from+1; m <= upto; m = m + 1 )
      {
         if (x[m] > x[mxp])  mxp = m;
      } // for (m ...
   }
   return mxp;
} // maxNdx
```

Όπως βλέπεις και εδώ αποφεύγουμε να καλέσουμε την *exit* αλλά εδώ έχουμε τη δυνατότητα –όταν υπάρχει πρόβλημα με την περιοχή επεξεργασίας– να επιστρέψουμε μια τιμή ("`-1`") που δεν μπορεί να είναι δείκτης στοιχείου πίνακα.[2]

Οι παράμετροι πίνακες έχουν μια σημαντική διαφορά από τις παραμέτρους τιμής: *Η τυπική παράμετρος-πίνακας δεν είναι αντίγραφο της πραγματικής παραμέτρου· είναι το ίδιο αντικείμενο*. Γι' αυτό, προσοχή!

- ***Αν αλλάξεις τιμές στοιχείων παράμετρου-πίνακα μέσα στη συνάρτηση η αλλαγή περναει στη συνάρτηση που έκανε την κλήση.***

Παράδειγμα

Αλλάζουμε τη *vectorSum* ως εξής:

```
double vectorSum( double x[], int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )  sum = sum + x[m];

   for ( m = 0; m <= n-1; m = m + 1 )  x[m] = 0;

   return sum;
```

[2] Καταλαβαίνεις τώρα γιατί προτιμήσαμε να βάλουμε τύπο αποτελέσματος της συνάρτησης `int` και όχι `unsigned int`.

```
} // vectorSum
```

Δηλαδή, αφού υπολογίσουμε το άθροισμα των στοιχείων, αλλάζουμε τις τιμές τους σε "**0**". Καλούμε τη συνάρτηση και γράφουμε τις τιμές των στοιχείων πριν και μετά την κλήση:

```
cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;

cout << " Σx: " << vectorSum( x, N, 0, N-1 ) << endl;

cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;
```

Αποτέλεσμα:

```
x:  5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
Σx: -7.05
x:  0  0  0  0  0  0  0  0  0
```

Η C++ σου δίνει ένα εργαλείο για να προστατευθείς από αλλαγές στοιχείων του πίνακα κατά λάθος. Μπορείς να δηλώνεις τις παραμέτρους σου **const** (σταθερές). Αν στο παραπάνω παράδειγμα δηλώσεις:

```
double vectorSum( const double x[], int n, int from, int upto )
{ . . . }
int maxNdx( const int x[], int n, int from, int upto )
{ . . . }
```

τότε αν ο *μεταγλωττιστής* βρει μέσα στο σώμα της συνάρτησης εντολή που να αλλάζει τιμή στοιχείου του x θα βγάλει λάθος: «Cannot modify a const object» (δεν μπορείς να αλλάξεις ένα αντικείμενο **const**).

Παρατήρηση: ▶

Όταν γράφουμε μια συνάρτηση αναδρομικώς μας *είναι απαραίτητη η περιοχή επεξεργασίας* η οποία περιορίζεται σε κάθε αναδρομική κλήση. Δίνουμε στη συνέχεια τις δύο συναρτήσεις σε αναδρομική μορφή:

```
double rVectorSum( const double x[], int n, int from, int upto)
{
   double sum( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      sum = x[upto] + rVectorSum( x, n, from, upto-1 );
   }
   return sum;
} // rVectorSum
```

Σε κάθε αναδρομική κλήση μειώνεται κατά 1 η τιμή της *upto* μέχρι που να γίνει κλήση με *upto* < *from* και η *sum* μένει με το 0 παίρνει αρχικώς.

```
int rMaxNdx( const int x[], int n, int from, int upto )
{
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      if ( from == upto )
         mxp = from;
      else
      {
         mxp = rMaxNdx( x, n, from+1, upto );
         if ( x[from] > x[mxp])  mxp = from;
      }
   }
```

```
      return mxp;
   } // rMaxNdx
```

Εδώ ο περιορισμός της περιοχής επεξεργασίας γίνεται με αύξηση της τιμής της *from*. ◀

9.4 Δύο Παραδείγματα με Αριθμούς

Στην παράγραφο αυτή θα δούμε δυο αριθμητικά παραδείγματα, από τα πιο χαρακτηριστικά για πίνακες. Είναι πολύ πιθανό να σου φανούν χρήσιμα σε προγράμματα που θα γράψεις για άλλα ενδιαφέροντα ή άλλες υποχρεώσεις σου.

Πριν προχωρήσουμε όμως να σου τονίσουμε κάτι, αν δεν το έχεις καταλάβει ήδη. Στα προγράμματα με πίνακες τα στοιχεία εισόδου είναι συνήθως πολλά. Μέχρι να κάνεις ένα πρόγραμμα να δουλέψει –μέχρι να διορθώσεις τα διάφορα σημαντικά λάθη– θα χρειαστεί συνήθως να πληκτρολογήσεις αρκετές φορές τα ίδια στοιχεία, πράγμα ενοχλητικό και αντιπαραγωγικό! Μια καλή λύση είναι η εξής: Γράψε τα στοιχεία με τα οποία δοκιμάζεις το πρόγραμμά σου σε ένα αρχείο text. Γράψε το πρόγραμμά σου έτσι που να μην διαβάζει από το πληκτρολόγιο, αλλά από το αρχείο.

Έτσι είναι γραμμένα τα παραδείγματα που ακολουθούν.

Παράδειγμα 1 - Τιμή Πολυωνύμου (αλγόριθμος του Horner)

Ένα πολύ συνηθισμένο πρόβλημα, σε αριθμητικά προγράμματα, είναι ο υπολογισμός της τιμής πολυωνύμου:

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m = \sum_{k=0}^{m} a_k x^k$$

όταν δίνονται οι συντελεστές (a_k, $k = 0..m$) και η τιμή της x.

Θέλουμε μια συνάρτηση που θα παίρνει ως παραμέτρους, τους συντελεστές και το x και θα επιστρέφει ως τιμή το $p(x)$.

Οι συντελεστές θα πρέπει να αποθηκευτούν σε ένα μονοδιάστατο πίνακα τύπου `double` με $m+1$ στοιχεία ($0 .. m$). Η συνάρτηση που γράφουμε είναι αρκετά γενική:

```
   double p1( const double a[], int m, double x )
```

Για να δοκιμάσουμε τη συνάρτηση που θα γράψουμε, ετοιμάζουμε το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

   double p1( const double a[], int m, double x );

int main()
{
   const int N( 20 );

   double   a[N];    // συντελεστές πολυωνύμου
   int      b;       // βαθμός πολυωνύμου
   double   x, y;
   int      k, pa;
   ifstream syntFl;  // από το αρχείο Text με τους συντελεστές

   syntFl.open( "syntFl.txt" );
   syntFl >> b;
   if ( syntFl.eof() )
   {
      syntFl.close();
      cout << "αρχειο συντελεστών άδειο" << endl;
   }
   else if ( b < 0 || b > N-1 )
```

```
   {
      syntFl.close();
      cout << "λάθος βαθμός πολυωνύμου" << endl;
   }
   else   // θέλουμε να διαβάσουμε b+1 συντελεστές
   {      // στις θεσεις a[0]..a[b]
      k = 0;  syntFl >> a[k];
      while ( !syntFl.eof() && k <= b-1 )
      {
         k = k + 1;  syntFl >> a[k];
      } // while
      if ( syntFl.eof() )  pa = k;
                     else  pa = k + 1;
      syntFl.close();
   // διαβάσαμε pa συνετελεστές
      if ( pa < b + 1 )
         cout << "Διάβασα " << pa << " τιμές" << endl;
      else    // όλα εντάξει
      {
         cout << " ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :" << endl;
         for ( k = 0; k <= b; k = k+1 )  cout << a[k] << "  ";
         cout << endl;

         cout << "ΔΩΣΕ  x = ";   cin >> x;
         while ( x != 0 )
         {
            cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
                 << p1( a, b, x ) << endl;
            cout << "ΔΩΣΕ  x = ";   cin >> x;
         } // while
         cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
              << p1( a, b, x ) << endl;
      } // if (pa ...
   } // !syntFl.eof ...
} // main
```

Το syntfl.txt είναι ένα αρχείο text, όπου γράφουμε, με τον κειμενογράφο μας, στην πρώτη γραμμή το βαθμό του πολυωνύμου και στις επόμενες τους συντελεστές. Βλέπεις ότι διαβάζουμε τα στοιχεία του a όπως είδαμε στην §9.1.1 και τους γράφουμε όπως είδαμε στην §9.1.2. Πριν από αυτό, διαβάζουμε και ελέγχουμε την τιμή του βαθμού του πολυωνύμου· θα πρέπει να είναι: $0 \le b \le N - 1$.

Ας πάρουμε για παράδειγμα το παρακάτω πολυώνυμο, 10ου βαθμού, που αποτελείται από 11 όρους (έχει 11 συντελεστές):

$p(x) = -10 + 9.5x + 7.2x^2 + 6.8x^3 - 6.1x^4 + 5.3x^5 - 4.6x^6 + 3.5x^7 + 2.7x^8 - x^9 + 0.8x^{10}$

Στο αρχείο μας θα γράψουμε:

```
10
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
```

Ελέγχουμε τη **while** με φρουρό το "**0**" στη x. Πάντως δεν ξεχνούμε να γράψουμε και το $p(0)$ στο τέλος.

Και τώρα, να γράψουμε τη συνάρτηση, δηλαδή εντολές που υπολογίζουν το άθροισμα που γράψαμε πιο πάνω. Εύκολο:

```
      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
      {
         Υπολόγισε το x^k
         px = px + a[k]*x^k
      }
```

Πως υπολογίζουμε το x^k; Ε, αυτό το ξέρουμε: **pow(x, k)**. Η συνάρτηση που θέλαμε, γράφτηκε εύκολα:

```
   double p1( const double a[], int m, double x )
   {
```

```
      double px;
      int k;

      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
         px = px + a[k]*pow(x, k);
      return px;
   }  // p1
```

Εισάγουμε τη συνάρτηση στο πρόγραμμά μας και να ένα παράδειγμα εκτέλεσης:

```
 ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
ΔΩΣΕ  x = 0.3562
 ΑΠΟΤΕΛΕΣΜΑ:  p(0.3562)= -5.46928
ΔΩΣΕ  x = -0.45
 ΑΠΟΤΕΛΕΣΜΑ:  p(-0.45)= -13.8303
ΔΩΣΕ  x = 0
 ΑΠΟΤΕΛΕΣΜΑ:  p(0)= -10
```

Και τώρα να *μετρήσουμε*. Ο αλγόριθμός μας κάνει:

m υψώσεις σε δύναμη (pow),

m πολλαπλασιασμούς *a*[*k*] * ...

m προσθέσεις *px* + *a*[*k*] * ...

Δεν μετράμε εκχωρήσεις και τις πράξεις για τον έλεγχο της **for**.

Τι έγινε όμως εδώ; Αγνοήσαμε τελείως το γεγονός ότι: $x^k = x^{k-1} \cdot x$ και για κάθε όρο υπολογίζαμε το x^k από την αρχή. Ασυγχώρητα σπάταλο πρόγραμμα. Η *p2()* διορθώνει αυτήν την αβλεψία:

```
   double p2( const double a[], int m, double x )
   {
      double px, xPow;
      int    k;

      px = a[0];  xPow = 1;
      for ( k = 1; k <= m; k = k+1 ) // εδώ έχουμε xPow = x^(k-1)
      {
         xPow = xPow * x;            // xPow = x^k
         px = px + a[k]*xPow;
      }
      return  px;
   }  // p2
```

Εδώ υπολογίζουμε το x^k με ένα πολλαπλασιασμό από το x^{k-1} και αποφεύγουμε την χρονοβόρα ύψωση σε δύναμη.

Βάλε την *p2()* στο πρόγραμμά σου και, όπου υπάρχει "`p1`" άλλαξε το σε "`p2`". Δοκίμασέ το και θα πάρεις τα ίδια αποτελέσματα που είδαμε παραπάνω. Αλλά τώρα έχουμε:

$2m$ πολλαπλασιασμούς (`xPow * x` και `a[k] * xPow`),

m προσθέσεις (`px + a[k] ...`),

Εδώ δεν έχουμε υψώσεις σε δύναμη και αυτό είναι σημαντικό κέρδος.

Όμως μπορούμε να έχουμε ένα καλύτερο πρόγραμμα! Πώς; Ας δώσουμε ένα παράδειγμα, με το πολυώνυμο τρίτου βαθμού:

$$p(x) = ax^3 + bx^2 + cx + d$$

Υπολογίζοντας την τιμή του με την *p2()*, θα κάνουμε 6 πολλαπλασιασμούς και 3 προσθέσεις.

Ένας άλλος τρόπος να γράψουμε το $p(x)$ είναι ο εξής:

$$p(x) = ((ax + b)x + c)x + d$$

που μας υποδεικνύει έναν άλλο τρόπο υπολογισμού:

$p(x) \leftarrow a;$

$p(x) \leftarrow p(x)x + b;$

$p(x) \leftarrow p(x)x + c;$
$p(x) \leftarrow p(x)x + d;$

Μετράμε: 3 πολλαπλασιασμοί και 3 προσθέσεις. Θρίαμβος! Αυτός ο νέος τρόπος υπολογισμού, που λέγεται **αλγόριθμος του Horner** ή μέθοδος των **φωλιασμένων πολλαπλασιασμών** (nested multiplication) δίνεται στη συνάρτηση ph:

```
// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x )
{
   double px;
   int    k;

   px = a[m];
   for ( k = m-1; k >= 0; k = k-1 )  px = px*x + a[k];
   return  px;
}  // ph
```

αλλά τώρα, κάνοντας μόνον:

m πολλαπλασιασμούς (`px*x`),
m προσθέσεις (`px*x + a[k]`),

Τι καταφέραμε λοιπόν; Η περιπλοκότητα αυτού του αλγόριθμου είναι τάξης *N*, όπως και του πρώτου. Αλλά, το ότι ελαττώσαμε τον αριθμό των πολλαπλασιασμών –αφού πρώτα-πρώτα διώξαμε αυτούς που χρειάζονται για την ύψωση σε δύναμη– μας δίνει διπλό κέρδος:

- *λιγότερα λάθη στρογγύλευσης*, δηλ. μεγαλύτερη ακρίβεια του αποτελέσματος· τα λάθη στρογγύλευσης είναι από τα σοβαρότερα προβλήματα με τον τύπο `double`,
- εξοικονόμηση χρόνου, γιατί οι πράξεις στον τύπο `double` είναι πιο αργές.

Συνεπώς ο αλγόριθμος του Horner είναι πολύ καλύτερος από τους προηγούμενους. Για υπολογισμό 500 τιμών του παραπάνω πολυωνύμου οι λόγοι των χρόνων των τριών προγραμμάτων είναι: 210:71:42. Όπως βλέπεις, η μεγάλη διαφορά προέρχεται από τον υπολογισμό της δύναμης, περίπου 3:1. Από τον υποδιπλασιασμό του πλήθους των πολλαπλασιασμών είχαμε περίπου υποδιπλασιασμό του χρόνου, περίπου 7:4.

Ποιό είναι το δίδαγμα από τα παραπάνω; Κατ' αρχήν:

♦ ***Υπάρχει πάντα μια καλύτερη λύση!***

Πάντα; Ναι, εκτός αν μπορείς να αποδείξεις το αντίθετο. Και θα ψάχνουμε πάντα για την καλύτερη λύση; Δεν θα πάρουμε υπ' όψη μας ότι την *p1()* την είχαμε έτοιμη στο κεφάλι μας πριν μας τη ζητήσουν; Και βέβαια!

Ας πούμε τα πράγματα με οικονομικούς όρους: Η *p1()* είχε πολύ μικρό **κόστος ανάπτυξης**, ενώ η *ph()* είχε μικρό **κόστος εκμετάλλευσης**[3]. Δεν θα πρέπει να αγνοήσουμε την *p2()*, όπου επιτύχαμε ικανοποιητικό κόστος εκμετάλλευσης χωρίς μεγάλο κόστος ανάπτυξης.

Αν η συνάρτηση, που έχουμε να γράψουμε, πρόκειται να χρησιμοποιηθεί σε κάποιο πρόγραμμα, που υπολογίζει χιλιάδες τιμές και θα χρησιμοποιείται πολύ καιρό, θα πρέπει να κατεβάσουμε το κόστος εκμετάλλευσης με κάθε θυσία (στην περίπτωσή μας: υψηλό κόστος ανάπτυξης). Αν πρόκειται να χρησιμοποιηθεί μια φορά, τότε δεν είναι και τρομερό να χρησιμοποιήσουμε κάτι που το γράφουμε πολύ γρήγορα, ακόμα κι αν δεν έχει την "ταχύτητα του φωτός".

Ο καλός προγραμματιστής ξέρει να σταθμίσει αυτούς τους παράγοντες και να δώσει τη *συνολικώς* καλύτερη λύση. Βέβαια, ο καλός προγραμματιστής έχει τις απαραίτητες γνώσεις για να γράψει κατ' ευθείαν την *ph()!*[4]

---

[3] Ένας άλλος παράγοντας, το **κόστος συντήρησης**, δεν παίζει σημαντικό ρόλο στο παράδειγμά μας.

[4] Στην πραγματικότητα την έχει ήδη έτοιμη σε κάποια από τις βιβλιοθήκες προγραμμάτων που έχει.

Παράδειγμα 2 - Στατιστικές

Αν μας δοθούν N αριθμοί x_k, $k = 0,\ldots N-1$, έχουμε τη μέση τιμή τους:

$$\langle \mathbf{x} \rangle = \frac{1}{N}\sum_{k=0}^{N-1} x_k$$

και την τυπική απόκλισή τους: $\sigma = \sqrt{\mathrm{Var}(\mathbf{x})}$, όπου:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2 \qquad (A)$$

που υπολογίζεται και ως εξής:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1} x_k^2 - \langle \mathbf{x} \rangle^2 \qquad (B)$$

Θέλουμε δυο προγράμματα τα οποία θα διαβάζουν (το καθένα) από το αρχείο text stat.dta τους N αριθμούς x_k τύπου `double` και θα βρίσκουν και θα τυπώνουν το $\langle \mathbf{x} \rangle$ και σ. Το πρώτο από τα προγράμματα θα χρησιμοποιεί τον τύπο (A) για το σ, ενώ το δεύτερο τον τύπο (B). Και τα δύο προγράμματα θα πρέπει να κάνουν τη μέγιστη δυνατή οικονομία σε μνήμη και υπολογιστικό χρόνο. Ποιό από τα δυό προγράμματα είναι καλύτερο και γιατί;

1η λύση: Έχουμε να γράψουμε δύο συναρτήσεις μια για τη μέση τιμή, ας την πούμε *vectorAvrg*, και μια για την τυπική απόκλιση, ας την πούμε stdDev. Και οι δύο θα βγουν από αλλαγές που θα κάνουμε στην *vectorSum* που γράψαμε πιο πάνω. Πρώτα η:

```
double vectorAvrg( const double x[], int n, int from, int upto)
{
   int m;
   double fv( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 )  fv = fv + x[m];
      fv = fv/(upto-from+1);
   }
   return fv;
} // vectorAvrg
```

και μετά η:

```
double stdDev( const double x[], int n, int from, int upto )
{
   int m;
   double fv( 0 ), vAvg;

   if ( 0 <= from && from <= upto && upto < n )
   {
      vAvg = vectorAvrg( x, n, from, upto );
      for ( m = from; m <= upto; m = m + 1 )
         fv = fv + pow( x[m]-vAvg, 2 );
      fv = sqrt( fv/(upto-from+1) );
   }
   return fv;
} // stdDev
```

Στον υπολογισμό του $\frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2$ θα μπορούσαμε να είχαμε γράψει:

```
   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + pow( x[m]-vectorAvrg(x, n, from, upto), 2 );
```

Αλλά έτσι, ζητούμε να κληθεί n φορές η *vectorAvrg* για να υπολογίσει τη μέση τιμή. Όπως τη γράψαμε τώρα, γίνεται μόνο μια κλήση, όταν δίνουμε αρχική τιμή στη *vAvg*. Βέβαια, ένας καλός μεταγλωττιστής θα κάνει αυτήν τη μετατροπή αυτομάτως.

Η `main` δεν έχει δυσκολίες:

```
#include <iostream>
```

```
#include <fstream>
#include <cmath>
using namespace std;

   double vectorAvrg(const double x[], int n, int from, int upto);
   double stdDev( const double x[], int n, int from, int upto );

int main()
{
   const int Nmax = 100;

   double x[Nmax], y;
   int n, k;
   ifstream  t("stat.dta");

   k = 0;  t >> x[k];
   while ( !t.eof() && k <= Nmax-2 )
   {  k = k + 1;  t >> x[k];  }  // while
   if ( t.eof() )  n = k;
             else  n = k + 1;
   t.close();

   cout << " <x> = " << vectorAvrg( x, Nmax, 0, n-1 ) << endl;
   cout << " σ = " << stdDev( x, Nmax, 0, n-1 ) << endl;
} // main
```

2η λύση: Η πρώτη σκέψη είναι να ξαναγράψουμε την *stdDev* και με αυτόν τον τρόπο να κάνουμε $N - 1$ λιγότερες αφαιρέσεις. Αλλά ας το ξανασκεφτούμε. Το πρόβλημά μας λύνεται αν υπολογίσουμε τα $\sum_{k=0}^{N-1} x_k$ και $\sum_{k=0}^{N-1} x_k^2$. Αυτά όμως μπορούμε να τα υπολογίζουμε όταν διαβάζουμε το αρχείο και *δεν χρειαζόμαστε πίνακα*! Βέβαια στην περίπτωση αυτή χάνουμε τις ωραίες μας συναρτήσεις.

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main()
{
   double x, vAvg, sigma, sx, sx2;
   int n, k;
   ifstream  t( "stat.dta" );

   sx = 0;  sx2 = 0;
   k = 0;  t >> x;
   while ( !t.eof() )
   {
      k = k + 1;
      sx = sx + x;   sx2 = sx2 + x*x;
      t >> x;
   }  // while
   n = k;
   t.close();
   cout << "Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      vAvg = sx/n;
      sigma = sqrt( sx2/n - vAvg*vAvg );
      cout << " <x> = " << vAvg << endl;
      cout << " σ = " << sigma << endl;
   } // if
} // main
```

Πρόσεξε ότι εδώ διαβάζουμε όσους αριθμούς υπάρχουν, αφού δεν έχουμε πίνακα.

k	v[k]	k	v[k]	k	v[k]	k	v[k]	k	v[k]
1	35	11	9	21	872	31	232	41	347
2	18	12	34	22	0	32	667	42	61
3	15	13	21	23	23	33	139	43	753
4	80	14	57	24	-34	34	-45	44	73
5	10	15	239	25	-32	35	-9	45	6
6	40	16	909	26	56	36	-89	46	-37
7	23	17	213	27	787	37	34	47	43
8	789	18	576	28	146	38	576	48	-99
9	563	19	903	29	589	39	122	49	344
10	1	20	239	30	568	40	99	50	572

Σχ. 9-3 Ο πίνακας που θα χρησιμοποιούμε στις δοκιμές των προγραμμάτων μας. Είναι ίδιος με αυτόν του Σχ. 9-2 με τη διαφορά ότι `v[0] == INT_MIN` και `v[51] == INT_MAX`.

Η 2η λύση είναι και ταχύτερη και οικονομικότερη σε χρήση μνήμης (αφού δεν χρησιμοποιεί πίνακα).

Πάντως οι δύο συναρτήσεις που γράψαμε για την πρώτη λύση είναι χρήσιμες γενικότερα.

9.5 Και Άλλες Συνηθισμένες Δουλειές με Πίνακες

Στην επεξεργασία στοιχείων με τη βοήθεια του ΗΥ συχνά εφαρμόζουμε τις παρακάτω επεξεργασίες πινάκων:

- **Αναζήτηση** ή **ψάξιμο** (searching) για τον εντοπισμό μιας τιμής μέσα σε έναν πίνακα.
- **Ταξινόμηση** (sorting), σε αύξουσα ή φθίνουσα διάταξη, των τιμών που είναι αποθηκευμένες στον πίνακα.
- **Συγχώνευση** (merging) δυό ταξινομημένων πινάκων σε έναν.

Για τις επεξεργασίες αυτές έχουν επινοηθεί αρκετοί αλγόριθμοι. Μερικοί από αυτούς είναι εξαιρετικά ενδιαφέροντες είτε διότι είναι γρήγοροι είτε διότι είναι οικονομικοί σε μνήμη είτε διότι μπορούν να εφαρμοστούν και σε (σειριακά) αρχεία είτε τέλος διότι είναι *όμορφοι*(!)[5]. Στις επόμενες παραγράφους θα δούμε μερικούς αλγορίθμους και τα αντίστοιχα προγράμματά τους για τέτοιες επεξεργασίες. Σίγουρα οι αλγόριθμοι αυτοί δεν είναι οι καλύτεροι, αλλά είναι αρκετά *απλοί* και *κατανοητοί*.

Για τα παραδείγματά μας θα χρησιμοποιήσουμε έναν πίνακα με 50 στοιχεία τύπου `int`. Επειδή σε ορισμένες περιπτώσεις θα μας χρειαστούν και φρουροί θα δηλώσουμε:

```
const int N( 50 );

int v[N+2];
```

και θα αποθηκεύουμε τις τιμές που θέλουμε στις θέσεις από `v[1]` μέχρι `v[N]`. Στη θέση `v[0]` θα έχουμε το −∞ (*INT_MIN*) και στη θέση `v[N+1]` το +∞ (*INT_MAX*). Οι τιμές των στοιχείων (Σχ. 9-3) βρίσκονται στο αρχείο text, με όνομα στο δίσκο pin.txt και θα τις διαβάζουμε, χωρίς ελέγχους, ως εξής:

```
ifstream a;
   :
a.open( "pin.txt" );
for ( k = 1; k <= N; k = k+1 )       // Διάβασε τον πίνακα
   a >> v[k];
a.close();
v[0] = INT_MIN;  v[N+1] = INT_MAX;   // βάλε τους φρουρούς
```

[5] Αισθητική των αλγορίθμων; Μα σίγουρα θα έχει τύχει να δεις μερικές όμορφες μαθηματικές αποδείξεις. Παρόμοια αισθητική υπάρχει κι' εδώ.

Μπορείς να δεις το περιεχόμενο του πίνακα όπως φαίνεται στο Σχ. 9-3 αν στις εντολές που δώσαμε στο τέλος της §9.2.3 αλλάξεις τη **for** που διατρέχει τις γραμμές:

```
for ( c = 0; c <= 4; c = c+1 )  cout << "     k  v[k]";
cout << endl;
for ( r = 1; r <= 10; r = r+1 )   // τύπωσε τη r-οστή γραμμή
{
   for ( c = 0; c <= 40; c = c+10 )
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << v[r+c];
   } // for ( c = ...
   cout << endl;                 // ...και πήγαινε στη επόμενη
} // for ( r = ...
```

9.5.1 Αναζήτηση στα Στοιχεία Πίνακα

Όταν λέμε «αναζήτηση» εννοούμε τη διαδικασία που δίνει απάντηση στο εξής πρόβλημα:

Έχουμε έναν πίνακα `T v[N]` *και μια τιμή x (τύπου T). Υπάρχει στοιχείο του* `v` *που να έχει τιμή ίση με x και αν ναι ποια η τιμή του δείκτη για το στοιχείο αυτό;*

Παρατήρηση: ▶

Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου στο πρόγραμμά μας. Στην περίπτωση αυτή η αναζήτηση δίνει απάντηση στο ερώτημα «*ανήκει η τιμή x στο σύνολο (που υλοποιείται με τον πίνακα) v;*»◀

Εμείς θα δουλέψουμε με τον πίνακα τύπου `int` που είδαμε πιο πριν. Θέλουμε λοιπόν μια:

```
int linSearch( int v[], int n, int from, int upto, int x )
```

που θα ψάχνει στα στοιχεία v`[`*from*`]`, v`[`*from*+1`]`,..., v`[`*upto*`]` να βρει την τιμή *x*. Αν τη βρει, θα επιστρέφει τον δείκτη του στοιχείου ως τιμή της συνάρτησης, αλλιώς θα επιστρέφει τιμή –1. Αν δηλαδή δώσουμε:

```
thesi = linSearch( v, N+2, n1, n2, x );
```

και αν $0 < n1 \le n2 < N + 2$ θα πρέπει:

$$(n1 \le \text{linSearch}(v, N{+}2, n1, n2, x) \le n2 \;\&\&\; v[\text{linSearch}(v, N{+}2, n1, n2, x)] == x)$$
$$|| \;(\text{linSearch}(v, N{+}2, n1, n2, x) == -1 \;\&\&\; (\forall j: n1..n2 \bullet v[j] \;!= x))$$

Τα πράγματα είναι πολύ απλά:

Ξεκίνα από την αρχή (*from*)
Όσο (δεν τελείωσε ο πίνακας) και (δεν το βρήκες) κάνε τα εξής:
{ Προχώρησε στο επόμενο στοιχείο του πίνακα

Μεταφράζουμε:

Ξεκίνα από την αρχή (*from*)	→ `k = from`
δεν τελείωσε ο πίνακας	→ `k <= upto`
δεν το βρήκες	→ `v[k] != x`
Προχώρησε στο επόμενο στοιχείο του πίνακα	→ `k = k+1`

Δηλαδή:

```
k = from;
while ( k < upto && v[k] != x )  k = k+1;
```

Δεν τελειώσαμε όμως: Η εκτέλεση της **while** θα τελειώσει:

- είτε διότι `!(k < upto)` ή αλλιώς `k >= upto` (για την ακρίβεια `k == upto` αφού η τιμή της *k* αυξάνεται κατά 1), φτάσαμε δηλαδή στο τέλος της περιοχής αναζήτησης,
- είτε διότι `!(v[k] != x)` ή αλλιώς `v[k] == x`, δηλαδή βρήκαμε την τιμή που ψάχνουμε στο στοιχείο v`[`*k*`]`.

Χρειάζεται λοιπόν και άλλος ένας έλεγχος, ακριβώς μετά τη **while**:

```
if ( v[k] == x )  fv = k;
```

```
               else  fv = -1;
```

Να λοιπόν ολόκληρη η *linSearch*:

```
   // linSearch -- Ψάχνει στα στοιχεία v[from],... v[upto] να
   //              βρει την τιμή x. Αν τη βρει
   //                 επιστρέφει ως τιμή τον δείκτη του στοιχείου
   //              αλλιώς
   //                 επιστρέφει τιμή -1
   int linSearch( const int v[], int n, int from, int upto, int x)
   {
      int k, fv( -1 );

      if ( 0 <= from && from <= upto && upto < n )
      {
         k = from;
         while ( k < upto && v[k] != x )  k = k+1;
         if ( v[k] == x )  fv = k;
                     else  fv = -1;
      // (from <= fv <= upto && v[fv] == x) ||
      //                (fv == -1 && ( j:from..upto " v[j] != x))
      }
      return fv;
   } // linSearch
```

Αυτή είναι η μέθοδος **γραμμικής αναζήτησης** (linear search). Στη συνέχεια βλέπεις και ένα πρόγραμμα που τη δοκιμάζει:

```
#include <iostream>
#include <fstream>
#include <climits>
#include <cstdlib>
using namespace std;

 int linSearch( const int v[], int n, int from, int upto, int x );

int main()
{
   const int N( 50 );

   int  v[N+2];      // ο πίνακας όπου ψάχνουμε
   int  x;           // για την τιμή της ψάχνουμε
   int  ndx;         // ο δείκτης της, αν τη βρούμε
   int  k;
   ifstream a;

// Διάβασε τον πίνακα και βάλε τους φρουρούς
// όπως παραπάνω

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = linSearch( v, N+2, 1, N, x );
      if ( ndx > 0 )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
         cout << " ΔΕΝ ΥΠΑΡΧΕΙ" << endl;
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Παράδειγμα εκτέλεσης:

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 23
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 7
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): -121
 ΔΕΝ ΥΠΑΡΧΕΙ
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 6
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 45
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατηρήσεις ▶

1. Όταν δώσαμε τιμή της $x = 23$ πήραμε απάντηση 7. Αλλά, αν δεις τον πίνακα στο Σχ. 9-3, το 23 υπάρχει και στη θέση 23. Η διαδικασία αυτή θα σου δώσει λοιπόν *μόνο την πρώτη εμφάνιση της τιμής*. Ο πίνακας του παραδείγματος υλοποιεί ένα **πολυσύνολο** (multiset, bag). Για ένα πολυσύνολο B η ερώτηση "$x \in B$;" δεν έχει και τόσο νόημα· η πιο σωστή ερώτηση είναι «*πόσες φορές υπάρχει η τιμή x στο πολυσύνολο B;*» (άσκ. 9-10)

2. Ειδικώς για τη συνάρτηση αυτή, η βίαιη αντίδραση (`exit(1)`) όταν είμαστε εκτός προδιαγραφών δεν είναι καλή. Πολλές φορές καλούμε μια τέτοια συνάρτηση με *μηδενική περιοχή αναζήτησης* (*from* > *upto*) και αυτό που θέλουμε είναι η απάντηση: *δεν υπάρχει η τιμή*. Γι' αυτό, όταν δεν ισχύει η συνθήκη της **if** η *fv* μένει με την τιμή "`-1`" που σημαίνει ακριβώς αυτό. ◀

Που γίνεται η πολλή δουλειά στην *linSearch()*; Προφανώς στη **while**. Για κάθε επανάληψη γίνονται 2 συγκρίσεις, 1 λογική "`&&`", 1 πρόσθεση και 1 εκχώρηση (`k = k+1`). Αν δεν υπάρχει η τιμή που ψάχνουμε –η χειρότερη περίπτωση για τον αλγοριθμό μας– όλα πολλαπλασιάζονται επί N (για την ακρίβεια επί *upto*−*from*+1).

Αν καταφέρουμε να διώξουμε τη μια σύγκριση διώχνουμε αυτόματα και τη λογική πράξη και ο αλγόριθμός μας επιταχύνεται σημαντικά. Ας δοκιμάσουμε. Και πρώτα-πρώτα: Ποια θα διώξουμε; Μάλλον δεν μπορούμε να διώξουμε την "`v[k] != x`". Γι' αυτήν γράφτηκε ολόκληρη η διαδικασία! Αν διώξουμε την "`k < upto`" πώς θα σταματήσουμε στο τέλος του πίνακα; Θα μπορούσαμε να σταματήσουμε με την άλλη συνθήκη. Αλλά αν η x δεν υπάρχει στον πίνακα; Θα τη βάλουμε εμείς με το ζόρι!

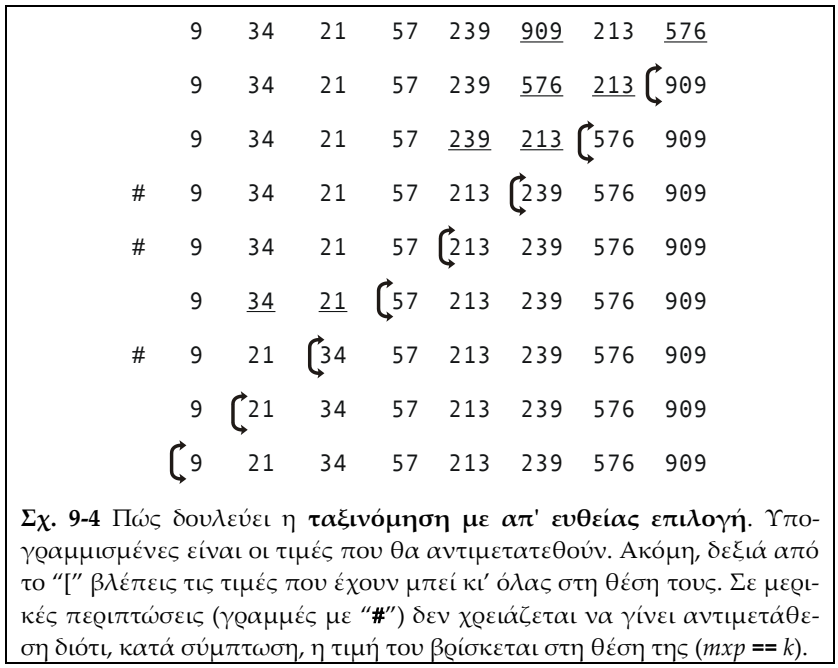
Τώρα μπερδεύτηκες, ε; Λοιπόν, πριν αρχίσουμε την αναζήτηση, θα βάλουμε την x στη θέση `v[`*upto*`+1]`. Όταν η "`while (v[k] != x) k = k+1`" σταματήσει, ελέγχουμε την τιμή της k: Αν $k \le upto$ τότε πραγματικά η x υπάρχει στον πίνακα, αλλιώς, αν $k > upto$ τότε δεν υπάρχει· σταμάτησε από τη x που βάλαμε εμείς στην `v[`*upto*`+1]`. Έξυπνο, έτσι; Αλλά κάτι σου θυμίζει! Τι άλλο; Την **τιμή - φρουρό**. Η τεχνική του φρουρού λύνει πολλά προβλήματα και απλουστεύει τα προγράμματά μας:

```
        save = v[upto+1];    // φύλαξε το v[upto+1]
        v[upto+1] = x;       // φρουρός
        k = from;
        while ( v[k] != x )  k = k+1;
        if ( k <= upto )  fv = k;
                    else  fv = -1;
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Πρόσεξε ότι πριν βάλουμε το φρουρό (x) στη θέση `v[`*upto* + 1`]` φυλάγουμε την τιμή που υπάρχει εκεί σε μια μεταβλητή (*save*) και αποκαθιστούμε την αρχική κατάσταση μετά το τέλος της αναζήτησης. Αν η *upto* έχει τιμή N δεν υπάρχει πρόβλημα αφού έχουμε προβλέψει μια ακόμη θέση (όπου έχουμε τον φρουρό "+∞").

Αλλά, δυστυχώς, ο μεταγλωττιστής έχει αντιρρήσεις: "`Cannot modify a const object in function linSearch`". Τι συμβαίνει; Φταίει το "`const`" που βάλαμε στην παράμετρο-πίνακα. Αργότερα θα δούμε πώς μπορούμε να το διορθώσουμε. Προς το παρόν θα καταφύγουμε σε «ριζικές θεραπείες»: θα βγάλουμε το "`const`"!

```
int linSearch( int v[], int n, int from, int upto, int x )
{
   int save;
   int k, fv( -1 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      save = v[upto+1];    // φύλαξε το v[upto+1]
      v[upto+1] = x;       // φρουρός
      k = from;
      while ( v[k] != x )  k = k+1;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
```



Σχ. 9-4 Πώς δουλεύει η **ταξινόμηση με απ' ευθείας επιλογή**. Υπογραμμισμένες είναι οι τιμές που θα αντιμετατεθούν. Ακόμη, δεξιά από το "[" βλέπεις τις τιμές που έχουν μπεί κι' όλας στη θέση τους. Σε μερικές περιπτώσεις (γραμμές με "**#**") δεν χρειάζεται να γίνει αντιμετάθεση διότι, κατά σύμπτωση, η τιμή του βρίσκεται στη θέση της (*mxp* == *k*).

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
    // (from <= fv <= upto && v[fv] == x) ||
    //         (fv == -1 && (για κάθε j:from..upto " v[j] != x))
    }
    return fv;
} // linSearch
```

9.5.2 Ταξινόμηση Στοιχείων Πίνακα

Τώρα θέλουμε να ταξινομήσουμε[6] τα στοιχεία του σε αύξουσα διάταξη, δηλ. να έχουμε (στα v[0] και v[*N*+1] έχουμε τα "−∞" και "+∞"):

$$\forall j: 1..n \bullet v[j] \le v[j+1]$$

Ενας απλός τρόπος είναι ο εξής:

Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*]
Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*−1] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*−1]
:
Βρες το μέγιστο από τα στοιχεία v[1]...v[2] και
αντιμετάθεσε την τιμή του με αυτήν του v[2]

Αυτή η διαδικασία περιγράφεται και ως εξής:

```
for (k = N; k >= 2; k = k-1)
{
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k] και
   αντιμετάθεσε την τιμή του με αυτήν του v[k]
}
```

Τελειώσαμε! Η "εντολή":

```
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k]
```

μας είναι γνωστή. Χρησιμοποιώντας τη *maxNdx* που είδαμε στην §9.3 μπορούμε να τη γράψουμε ως εξής:

---

[6] Όπως θα δούμε στη συνέχεια, ο πιο συνηθισμένος λόγος για να ταξινομήσουμε τα στοιχεία ενός πίνακα είναι η ευκολία στο ψάξιμό τους είτε με υπολογιστή είτε με το χέρι. Σκέψου, πώς θα έψαχνες τον τηλεφωνικό κατάλογο αν δεν ήταν ταξινομημένος!

```
        mxp = maxNdx( v, N+2, 1, k );
```

Όσο για την "εντολή":

αντιμετάθεσε την τιμή του με αυτήν του v[*k*]

ξέρουμε να τη γράψουμε:

```
        sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
```

Αυτή η μέθοδος λέγεται **ταξινόμηση με απ' ευθείας επιλογή** (straight selection sort). Στο Σχ. 9-4 βλέπεις πως ταξινομείται με τη διαδικασία αυτή ένας πίνακας με 8 στοιχεία.

Το παρακάτω πρόγραμμα:

- διαβάζει τα στοιχεία του v από το pin.txt,
- τα ταξινομεί με τη μέθοδο που είδαμε,
- γράφει τον ταξινομημένο πίνακα στην οθόνη και
- τον φυλάγει στο αρχείο pinsrt.txt.

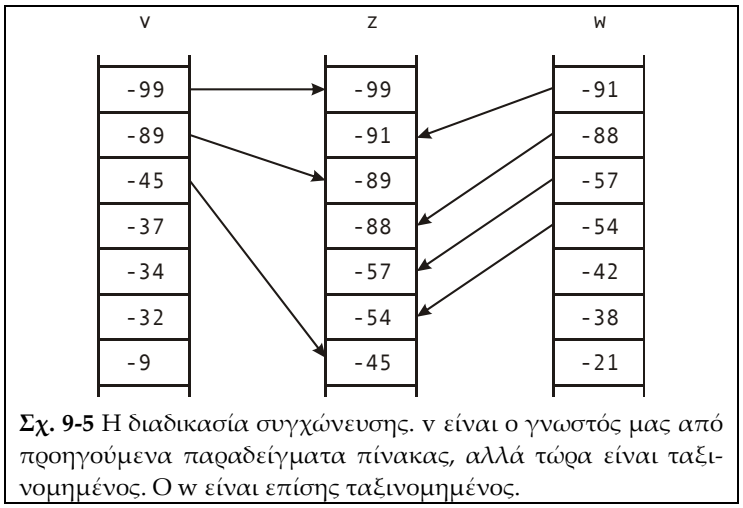
Φυσικά, χρησιμοποιεί τη *maxNdx* που ξέρουμε.

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

   int maxNdx( int x[], int n, int from, int upto );

int main()
{
   const int N( 50 );

   int  v[N+2], sv;
   int  k, mxp, r, c;
   fstream a;

   a.open( "pin.txt", ios_base::in );
// Διάβασε τον πίνακα και βάλε τους φρουρούς όπως παραπάνω

// ταξινόμηση
   for ( k = N; k >= 2; k = k-1 )
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
   } // for

// Δείξε τα στοιχεία του ταξινομημένου πίνακα όπως παραπάνω
// φύλαξε τον πίνακα
   a.open( "pinsrt.txt", ios_base::out );
   for ( k = 1; k <= N; k = k+1 )  a << v[k] << endl;
   a.close();
} // main
```

Να τι θα δούμε στην οθόνη:

```
ΣΤΟΙΧΕΙΑ ΤΟΥ ΠΙΝΑΚΑ v ΜΕΤΑ ΤΗΝ ΤΑΞΙΝΟΜΗΣΗ
     k  v[k]     k  v[k]     k  v[k]     k  v[k]     k  v[k]
     1  -99     11    9     21   40     31  146     41  576
     2  -89     12   10     22   43     32  213     42  576
     3  -45     13   15     23   56     33  232     43  589
     4  -37     14   18     24   57     34  239     44  667
     5  -34     15   21     25   61     35  239     45  753
     6  -32     16   23     26   73     36  344     46  787
     7   -9     17   23     27   80     37  347     47  789
     8    0     18   34     28   99     38  563     48  872
     9    1     19   34     29  122     39  568     49  903
    10    6     20   35     30  139     40  572     50  909
```

Ας «μετρήσουμε» λιγάκι αυτή τη μέθοδο. Όπως φαίνεται από τη **for**, θα έχουμε *N*−1 κλήσεις της *maxNdx* και άλλες τόσες αντιμεταθέσεις τιμών του v. Στη *maxNdx* (*from* = 1, *upto*



Σχ. 9-5 Η διαδικασία συγχώνευσης. v είναι ο γνωστός μας από προηγούμενα παραδείγματα πίνακας, αλλά τώρα είναι ταξινομημένος. Ο w είναι επίσης ταξινομημένος.

= *k*), υπάρχει μια άλλη **for** που, για κάθε τιμή της *k*, εκτελεί, *k* – 1 φορές τη σύγκριση v**[***m***]** > v**[***mxp***]**. Αυτή η σύγκριση θα εκτελεσθεί συνολικώς:

$$(N-1) + (N-2) + \ldots + 2 + 1 = \tfrac{1}{2}N(N-1) = \tfrac{1}{2}N^2 - \tfrac{1}{2}N$$

φορές. Για μεγάλα *N*, ο όρος ½*N* γίνεται αμελητέος και ο όρος που κυριαρχεί είναι ο ½N^2. Λέμε ότι ο χρόνος εκτέλεσης του αλγόριθμου αυξάνεται σαν το N^2. Πάντως υπάρχουν και άλλες πράξεις, που πρέπει να εξετασθούν πιο προσεκτικά. Π.χ. πόσες φορές εκτελείται η `mxp = m`; Για πιο πολύπλοκες δομές στοιχείων θα πρέπει να μετρήσουμε τον υπολογιστικό χρόνο για την εκτέλεση της v**[***m***]** > v**[***mxp***]** και ακόμη της αντιμετάθεσης (εκτελείται *N*–1 φορές).

Αυτή η μέθοδος είναι πολύ απλή στη σύλληψη και για μικρές τιμές της *N* είναι ικανοποιητική.

9.5.3 Συγχώνευση Πινάκων

Έστω ότι έχουμε δυό μονοδιάστατους πίνακες v και w, με στοιχεία του ίδιου τύπου, ας πούμε `int`, με *NV* και *NW* στοιχεία αντιστοίχως. Οι πίνακες αυτοί είναι ταξινομημένοι σε αύξουσα τάξη. Θέλουμε να τους συγχωνεύσουμε σε έναν άλλο πίνακα z, με *NZ* (≥ *NV* + *NW*) στοιχεία του ίδιου τύπου, έτσι ώστε ο z να είναι ταξινομημένος επίσης κατ' αύξουσα διάταξη.

Μια πρώτη σκέψη (και η χειρότερη) που θα μπορούσαμε να κάνουμε είναι η εξής: Να βάλουμε στις πρώτες *NV* θέσεις του πίνακα z τα στοιχεία του v, στις επόμενες *NW* θέσεις τα στοιχεία του w και στη συνέχεια να ταξινομήσουμε τον z όπως είδαμε στην προηγούμενη παράγραφο. Με αυτόν τον τρόπο, δεν εκμεταλλευόμαστε το γεγονός ότι οι δύο πίνακες είναι ήδη ταξινομημένοι. Ο υπολογιστικός χρόνος θα είναι ανάλογος του $(NV+NW)^2$, ενώ ήδη έχει καταναλωθεί χρόνος για να ταξινομηθούν οι v και w. Θα πρέπει να ψάξουμε για κάτι καλύτερο.

Για δες την παρακάτω απλή ιδέα:

```
Πάρε την πρώτη τιμή από τον v
Πάρε την πρώτη τιμή από τον w
Ετοιμάσου να γράψεις στην πρώτη θέση του z
while ( δεν έφτασες στο τέλος ούτε του v ούτε του w )
{
  if ( τιμή από τον v < τιμή από τον w )
  {
    Γράψε στον z την τιμή από τον v
    Αντικατάστησε την τιμή από τον v με την επόμενή της
  }
  else   // if τιμή από τον v >= τιμή από τον w
  {
```

```
    Γράψε στον z την τιμή από τον w
    Αντικατάστησε την τιμή από τον wμε την επόμενή της
  }
  Ετοιμάσου να γράψεις στην επόμενη θέση του z
}
```

Στο Σχ. 9-5, βλέπεις πως δουλεύει αυτή η μέθοδος.

Αλλά δεν τελειώσαμε ακόμη: Η εκτέλεση της **while** θα τελειώσει όταν φτάσουμε στο τέλος είτε του v είτε του w. Στη συνέχεια θα πρέπει να αντιγράψουμε στον z τα στοιχεία του άλλου πίνακα, που δεν τελείωσε. Και αν τελειώσουν και οι δυο μαζί; Δεν γίνεται! *Σε κάθε εκτέλεση της περιοχής επανάληψης παίρνουμε ένα στοιχείο από έναν πίνακα*. Θα πρέπει να συμπληρώσουμε τον αλγόριθμό μας με το κομμάτι:

```
Πρόσθεσε στον z τα στοιχεία που περίσσεψαν από τον πίνακα που δεν εξαντλήθηκε
```

Θα γράψουμε ένα πρόγραμμα που θα διαβάζει, όπως ξέρουμε, τις τιμές των στοιχείων του πίνακα v, από το αρχείο pinsrt.txt, όπου τα γράψαμε με το πρόγραμμα της προηγούμενης παραγράφου, μετά την ταξινόμηση. Θα διαβάζει ακόμη, με τον ίδιο τρόπο, τις τιμές των (20) στοιχείων του πίνακα w που βρίσκονται στο αρχείο pinw.dta, ήδη ταξινομημένα. Στη συνέχεια θα συγχωνεύει τα στοιχεία των δύο πινάκων στον z.

Τρεις μεταβλητές *dv*, *dw*, *dz* θα παίζουν ρόλο δεικτών στους v, w, z αντιστοίχως. Οι *dv*, *dw* θα δείχνουν τα στοιχεία των v και w που συγκρίνουμε, ενώ ο *dz* δείχνει το στοιχείο του z όπου θα γίνει η αντιγραφή. Και οι τρεις θα αρχίζουν από 1:

```
   dv = 1;  dw = 1;  dz = 1;
```

Η while που δώσαμε παραπάνω γράφεται:

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
```

Αν, στο τέλος, τελείωσαν τα στοιχεία του v (`dv > NV`) και έχουν περισσέψει στοιχεία από τον w, τα αντιγράφουμε στον z ως εξής:

```
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
```

Πρόσεξε ότι οι αρχικές τιμές των δεικτών *dz* και *dw* γι' αυτήν τη while είναι ακριβώς οι τελικές τιμές τους από την προηγούμενη. Παρόμοια προσθέτουμε στον z και τα στοιχεία του v, αν έχει εξαντληθεί ο w. Μετά από τις πιό πάνω παρατηρήσεις μπορούμε να δώσουμε ολόκληρο το πρόγραμμα.

```
#include <fstream>
using namespace std;

int main()
{
   const int N( 50 ), NV( N ), NW( 20 ), NZ( 70 );

   int  v[N+2], w[NW+2], z[NZ+2];
   int  k;
   int  dv, dw, dz;
   fstream a;

   a.open( "pinsrt.txt", ios_base::in );
   for (k = 1; k <= NV; k = k+1)  a >> v[k];
   a.close();
   a.open( "pinw.txt", ios_base::in);
   for (k = 1; k <= NW; k = k+1)  a >> w[k];
   a.close();

// ΣΥΓΧΩΝΕΥΣΗ
   dv = 1;  dw = 1;  dz = 1;
```

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
   if ( dv > NV )
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
   else
      while ( dv <= NV )
      {  z[dz] = v[dv];  dv = dv + 1;  dz = dz + 1;  }

   a.open( "pinz.txt", ios_base::out );
   for ( k = 1; k <= NZ; k = k+1 )  a << z[k] << endl;
   a.close();
} // main
```

Μπορείς εύκολα να δείς ότι ο χρόνος εκτέλεσης αυτού του αλγόριθμου είναι ανάλογος του *NV*+*NW*.

9.6 Ταχύτερα – Οικονομικότερα – Καλύτερα

Τώρα ας γυρίσουμε για να ξαναδούμε τα προηγούμενα προβλήματά μας.

Πρώτα η αναζήτηση. Αλλάζουμε λίγο το πρόβλημά μας: Έστω π.χ. ότι έχουμε έναν μονοδιάστατο πίνακα, v, με *N* στοιχεία τύπου **int**, *ταξινομημένα κατ' αύξουσα τάξη* και μια τιμή *x* (επίσης τύπου **int**). Θέλουμε έναν αλγόριθμο που θα ψάχνει στα στοιχεία v[*from*], v[*from*+1], ..., v[*upto*] να βρει την τιμή *x*.

Οι δυο παραλλαγές γραμμικής αναζήτησης που είδαμε δουλεύουν μια χαρά. Αλλά παίρνοντας υπόψη μας την ταξινόμηση μπορούμε να τα καταφέρουμε κάπως καλύτερα. Και μάλιστα στη χειρότερη περίπτωση: όταν η τιμή που ψάχνουμε δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν, οι αλγόριθμοι που είδαμε σαρώνουν ολόκληρον τον πίνακα πριν αποφανθούν.

Αν ο πίνακας είναι ταξινομημένος, δεν χρειάζεται να τον ψάξουμε ολόκληρον. Ας ξεκινήσουμε από τον τηλεφωνικό κατάλογο. Ψάχνουμε το τηλέφωνο του "**ΠΑΠΑΔΗΜΑ**", αλλά μετά το όνομα "**ΠΑΠΑΔΑΝΙΗΛ**" βρίσκουμε το όνομα "**ΠΑΠΑΔΙΑΚΟΣ**". Φυσικά, σταματάμε το ψάξιμο!

Έστω, λοιπόν, ότι v[*from*] ≤ *x* ≤ v[*upto*]. Θα ψάχνουμε μέχρι να βρούμε κάποιο v[*k*] ≥ *x*:

- Αν έχουμε v[*k*] == *x* τότε βρήκαμε την τιμή.
- Αλλιώς, αν v[*k*] > *x*, δεν έχει νόημα να συνεχίσουμε την αναζήτηση αφού όλα τα επόμενα στοιχεία έχουν τιμές μεγαλύτερες από το v[*k*], άρα και από το *x*.

```
      k = from;
      while ( v[k] < x )  // I: ∀j:from-1..k-1 • v[j] < x
         k = k+1;
   // (∀j:0..k-1 • v[j] < x) && (x ≤ v[k])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Για να τερματίζεται η **while** και στην περίπτωση που *x* > v[*upto*] θα πρέπει να έχουμε φρουρό (+∞) στο v[*upto*+1]. Ο φρουρός (−∞) στο v[*from*−1] χρειάζεται για να έχει νόημα η αναλλοίωτή μας όταν *m* == *from*, αλλά δεν είναι απαραίτητη η φυσική του παρουσία.

Αν θέλεις, μπορείς να ψάξεις και από το τέλος προς την αρχή:

```
      k = upto;
      while ( x < v[k] )  // I: ∀j:k+1..upto+1 • x < v[j]
         k = k-1;
   // (v[k] ≤ x) && (∀j:k+1..upto+1 • x < v[j])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Στην περίπτωση αυτή, θα πρέπει να έχουμε φρουρό (−∞) στο v[*from*−1] για να τερματιζεται η **while** και στην περίπτωση που *x* < v[*from*]. Εδώ, δεν είναι απαραίτητη η φυσική του παρουσία φρουρού (+∞) στο v[*upto*+1], αλλά πρέπει να υποθέσουμε ότι υπάρχει για να έχει νόημα η αναλλοίωτή μας όταν *m* == *upto*.

Τώρα ας δούμε κάτι καλύτερο.

Έψαξες ποτέ σου τον τηλεφωνικό κατάλογο γραμμικώς; Μάλλον απίθανο. Συνήθως, αν ψάχνεις για τον ΠΑΠΑΔΗΜΑ, ανοίγεις τον κατάλογο εκεί που μαντεύεις ότι είναι το 'Π'. Αν πέσεις στο 'Σ', γυρνάς πιο πίσω. Αν τώρα πέσεις στο 'Ο' ξαναδοκιμάζεις πιο μπροστά κ.ο.κ. Ας προσπαθήσουμε να κάνουμε κάτι παρόμοιο και στον πίνακά μας.

Αρχίζουμε ελέγχοντας το στοιχείο που βρίσκεται στο μέσο του πίνακα, έχει δηλαδή δείκτη *middle* = (*from*+*upto*)/2. Αν v[*middle*] == *x*, βρήκαμε την τιμή που ψάχνουμε, τελειώσαμε. Αν v[*middle*] < *x* τότε, αφού ο πίνακας είναι ταξινομημένος, όλα τα στοιχεία v[*from*],... v[*middle*], είναι σίγουρα < *x*. Θα πρέπει λοιπόν να ψάξουμε στα: v[*middle*+1],... v[*upto*]. Αν βρίσκαμε v[*middle*] > *x* θα έπρεπε να συνεχίζαμε το ψάξιμο στην περιοχή v[*from*],... v[*middle*−1]. Και πώς θα συνεχιστεί το ψάξιμο; Μα με τον ίδιο τρόπο! Αλλά στον μισό πίνακα.

Δες το παράδειγμα στο Σχ. 9-4, όπου σε ένα ταξινομημένο πίνακα 8 στοιχείων ψάχνουμε για την τιμή 100.

Βάζουμε αρχικά, "`from = 1; upto = 8;`" και υπολογίζουμε το *middle* = 4: v[*middle*] = 57. Το 100 –αν υπάρχει– θα βρίσκεται μετά το 57. Συνεχίζουμε με τον ίδιο τρόπο, αφού αλλάξουμε το "`from = middle + 1;`" (= 5). Υπολογίζουμε και πάλι το *middle* = 6. Τώρα έχουμε: v[*middle*] = 239. Το 100 αν υπάρχει, θα υπάρχει πριν από το 239. Τώρα, πριν συνεχίσουμε, αλλάζουμε το "`upto = middle - 1;`" (= 5). Η περιοχή που ψάχνουμε περιορίστηκε σε ένα στοιχείο. Αλλά ας συνεχίσουμε: *middle* = 5, v[*middle*] = 213. Αν υπάρχει το 100, θα υπάρχει πριν από αυτό. Αλλάζουμε λοιπόν το "`upto = middle - 1;`" (= 4). Και εδώ έχουμε το «περίεργο φαινόμενο»: το άνω όριο της περιοχής αναζήτησης (*upto* = 4) να είναι μικρότερη από το κάτω όριο (*from* = 5). Δεν υπάρχει πια περιοχή αναζήτησης. Καιρός να σταματησουμε διότι το 100 δεν υπάρχει στον πίνακά μας.

Τελειώνουμε λοιπόν το ψάξιμο:

- όταν βρούμε το *x* (v[*middle*] == *x*) ή
- όταν η περιοχή που ψάχνουμε δεν υπάρχει πια (*from* > *upto*).

```
middle = (from + upto) / 2;
while ( from <= upto && v[middle] != x )
{
   if (v[middle] < x)  from = middle + 1;
   else if (v[middle] > x)  upto = middle - 1;
   middle = (from + upto) / 2;
}  // while
```

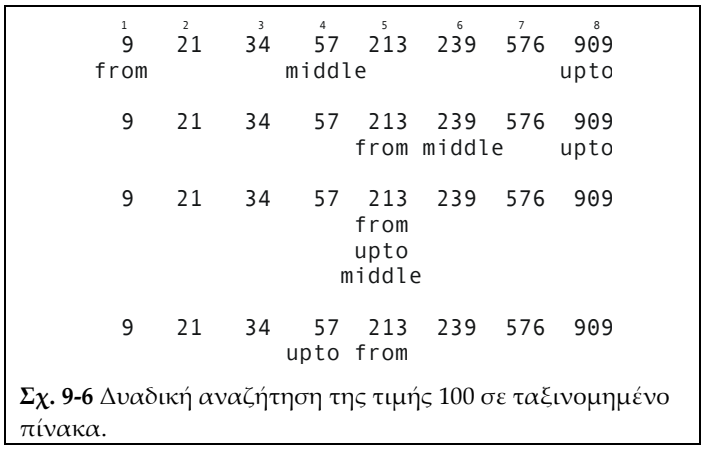
Μετά το τέλος της επαναληπτικής διαδικασίας, θα πρέπει να ελέγξουμε για ποιο λόγο σταμάτησε:

```
if (v[middle] == x)  η x υπάρχει στο Meso
```

Αυτός είναι ο αλγόριθμος **δυαδικής αναζήτησης** (binary search).

Όπως είδες και από το παράδειγμα, οι περισσότερες επαναλήψεις θα γίνουν στην περίπτωση που η τιμή *x* δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν το ψάξιμο πρέπει να συνεχιστεί ωσότου συναντηθούν οι δείκτες *from* και *upto* (που τους λέμε αντιστοίχως *κάτω δείκτη* και *άνω δείκτη* της περιοχής που ψάχνουμε). Ας πούμε ότι, αρχικά, η περιοχή που ψάχνουμε έχει *N* στοιχεία (*from* = 1 και *upto* = *N*). Όταν οι δείκτες συναντηθούν η περιοχή θα έχει λιγότερα από 1 στοιχεία (*upto* < *from*). Το μήκος της περιοχής μικραίνει με υποδιπλασιασμό (περίπου, γιατί έχουμε ακέραιη διαίρεση). Μετά από *l* επαναλήψεις, η περιοχή που ψάχνουμε θα έχει περίπου:

$$N / 2^l \text{ στοιχεία.}$$



Σχ. 9-6 Δυαδική αναζήτηση της τιμής 100 σε ταξινομημένο πίνακα.

Αυτός ο αριθμός γίνεται μικρότερος από 1 όταν το 2^l γίνει μεγαλύτερο από N, ή:

$$l > log_2 N$$

Έχουμε λοιπόν έναν αλγόριθμο με *λογαριθμική* περιπλοκότητα χρόνου, αντί για την γραμμική (ανάλογη του N) που έχουν οι προηγούμενοι. Τι σημαίνει αυτό; Αν π.χ. πάρουμε $N = 1024$, τότε στη χειρότερη περίπτωση θα έχουμε 10 επαναλήψεις ($log_2 1024 = 10$) της **while**, ενώ με το γραμμική αναζήτηση θα έχουμε 1024!

Η συνάρτηση *binSearch()*, που υλοποιεί τη μέθοδο δυαδικής αναζήτησης, θα είναι διαφορετική από τη *linSearch()* ως προς το εξής: Θα επιστρέφει πάντοτε μια τιμή δείκτη!

Έστω ότι έχουμε:

- πίνακα που δεν είναι «γεμάτος» αλλά έχει τιμές, ταξινομημένες κατ' αύξουσα τάξη, στις θέσεις από `v[1]` μέχρι `v[`*last*`]`, όπου *last* < *N* και
- μια τιμή *x* που πρέπει να εισαχθεί στον πίνακα ώστε να παραμείνει ταξινομημένος.

Η *binSearch()* θα μας δίνει τη θέση *k* στην οποία θα πρέπει να εισαχθεί η *x* διότι όταν σταματησει η **while** θα έχουμε $v[k-1] < x \le v[k]$.

Δες ένα χαρακτηριστικό παράδειγμα χρήσης της *binSearch*:

```
   ndx = binSearch( v, last, x );
   if ( v[ndx] == x )
      cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   else
   {
      for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
      v[ndx] = x;
      last = last + 1;
      cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   }
```

Από την τιμή που μας επιστρέφει η συνάρτηση –και φυλάγουμε στην *ndx*– δεν ξέρουμε αν βρέθηκε η *x* στιον πίνακα· χρειάζεται και ο έλεγχος

```
   if ( v[ndx] == x )
```

Πρόσεξε πώς γίνεται η εισαγωγή της νέας τιμής στον πίνακα: Με τη **for** «ανοίγουμε χώρο» για να τη εισαγάγουμε.

Να πώς θα είναι η *binSearch*:

```
// binSearch -- Δυαδική αναζήτηση στα στοιχεία v[1],...v[last]
//              για να βρεθεί θέση fv τέτοια ώστε:
//                       v[fv-1] < x <= v[fv]
//              Επιστρέφει την fv.
//              Ο v πρέπει να είναι ταξινομημένος κατ' αύξουσα
//              τάξη με φρουρούς στα άκρα
//              v[0] == -inf,  v[last+1] == +inf
unsigned int binSearch( const int v[], int last, int x )
{
   unsigned int l( 1 ), r( last+1 );
   unsigned int middle;
```

```
   while ( l < r ) // I: (∀k:[0..l) • v[k] < x) &&
   {               //    (∀k:[r..last+1) • v[k] >= x)
      middle = (l + r) / 2;
      if ( v[middle] < x )  l = middle + 1;
                    else  r = middle;
   }  // while
// v[l-1] < x <= v[l]
   return l;
}  // binSearch
```

Στη συνέχεια δίνουμε μια (όχι και πολύ αυστηρή) απόδειξη ορθότητας της *binSearch*.

Γιατί δεν βάλαμε και εδώ παραμέτρους (*from*, *upto*) που θα μας δίνουν δυνατότητα αναζήτησης σε τμήμα του πίνακα; Αυτή η δυνατότητα έρχεται σε σύγκρουση με τη δυνατότητα να μας δίνεται η θέση εισαγωγής. Δες ένα παράδειγμα. Ας πούμε ότι έχουμε:

0	1	2	3	4	6	7	8	9	10	11	12	13	14	15	16	17
-∞	2	7	11	15	17	19	23	29	31	37	41	44	53	+∞		

και αναζητούμε το 12 με *from* = 7 και *upto* = 12. Αφού δεν υπάρχει θα μας υποδειχθεί η εισαγωγή στη θέση *from* = 7 αλλά αυτό είναι λάθος. Παρομοίως, θα μας υποδειχθεί η εισαγωγή του 61 στη θέση *upto*+1 = 13. Οι υποδείξεις για τη θέση εισαγωγής είναι σωστές μόνον αν v[*from*-1] < *x* < v[*upto*+1].

Δοκιμάζουμε τη *binSearch* με ένα πρόγραμμα που έχει τις εντολές αναζήτησης – εισαγωγής (που είδαμε παραπάνω) σε έναν ταξινομημένο πίνακα που διαβάζουμε από το pinsrt.txt:

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

unsigned int binSearch( const int v[], int n, int x );

int main()
{
   const unsigned int N( 60 );

   int  v[N];               // ο πίνακας που ψάχνουμε
   unsigned int last(N-10); // με τιμές στις θέσεις v[1]...v[last]
   int  x;                  // για την τιμή της ψάχνουμε
   unsigned int ndx;        // εδώ θα είναι αν τη βρούμε
   int  k;
   ifstream a( "pinsrt.txt" );

   for ( k = 1; k <= last; k = k+1 )
      a >> v[k];                              // Διάβασε τον πίνακα
   a.close();
   v[0] = INT_MIN;  v[last+1] = INT_MAX;      // βάλε τους φρουρούς

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = binSearch( v, last, x );
      if ( v[ndx] == x )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
      {
         for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
         v[ndx] = x;
         last = last + 1;
         cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      }
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Τώρα, κρατούμε για τον πίνακα 60 θέσεις αλλά τον φορτώνουμε στις θέσεις 1..50. Στις θέσεις 0 και 51 μπαίνουν οι φρουροί.

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 23
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατήρηση: ▶

Εδώ θα πρέπει να σταματήσουμε για λίγο και να σκεφτούμε αυτά που είδαμε. Η δυαδική αναζήτηση είναι πολύ εντυπωσιακή στην ταχύτητά της αλλά δεν είναι θαύμα. Είναι γρήγορη, διότι ο πίνακας μας έχει υποστεί μια προεργασία, *έχει ταξινομηθεί*. Ας πούμε λοιπόν ότι θέλεις να κάνεις ένα σύστημα, όπου θα πρέπει θα ψάχνεις έναν πίνακα και να δίνεις την απάντησή σου *γρήγορα*. Θα πρέπει να χρησιμοποιήσεις μια γρήγορη μέθοδο, όπως είναι η δυαδική αναζήτηση. Ενώ όμως απαιτείται ταχύτητα στην αναζήτηση, πολύ συχνά, έχεις αρκετή άνεση χρόνου για να προετοιμάσεις κατάλληλα το σύστημά σου: να ταξινομήσεις τον πίνακά σου ή να ενημερώνεις τον πίνακά σου ώστε να είναι ταξινομημένος – διαδικασίες σχετικά χρονοβόρες. Με αυτήν τη φιλοσοφία σχεδιάζονται πολλά πληροφοριακά συστήματα σήμερα και ένα πολύ κοινό παράδειγμα είναι ο τηλεφωνικός κατάλογος. Όταν τον ψάχνεις, η αλφαβητική ταξινόμηση σου είναι απαραίτητη και την έχεις. Έτσι, έχεις τη δυνατότητα να ψάχνεις γρήγορα. Η ταξινόμηση γίνεται πριν τυπωθεί, όταν η υπηρεσία που τον ετοιμάζει έχει όλη τη χρονική άνεση να κάνει κάτι τέτοιο. ◀

Και τώρα, ας ξανάρθουμε στην ταξινόμηση. Ο αλγόριθμος που είδαμε απαιτεί, για μεγάλα N, χρόνο λN^2. Πρόσεξε τώρα το εξής:

- Κόβουμε τον πίνακά μας στη μέση. Έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε έναν από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικώς: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

Με τις διαδικασίες που γράψαμε αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα από αυτό, δεν μπορούμε να μη σκεφτούμε: «γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log N$, που φυσικά είναι καλύτερος από N^2. Αυτός ο αλγόριθμος χρειάζεται διπλή μνήμη από αυτήν που απαιτείται για την αποθήκευση του πίνακα. Αργότερα θα μάθεις ότι υπάρχουν αλγόριθμοι ταξινόμησης με χρόνο $N\log N$, χωρίς υπερβολικές απαιτήσεις μνήμης.

Αυτή είναι μια περίπτωση εφαρμογής της αρχής «διαίρει και βασίλευε» (divide and conquer) στον αλγόριθμό μας: Κόβουμε τον πίνακα στα δυο και λύνουμε το αρχικό πρόβλημα (ταξινόμηση) στα δυο κομμάτια.

9.6.1 Απόδειξη Ορθότητας της *binSearch*

Η απόδειξη ορθότητας θα γίνει με τη μέθοδο της επαγωγής.

Αρχικώς, στην [0..*l*) υπάρχει μόνον ο φρουρός "`-∞`" και ισχύει η "`-∞ <` x". Η δεξιά περιοχή είναι κενή: [*last*+1..*last*+1) και η ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x είναι (τετριμμένως) αληθής.

Αν v`[`*middle*`] <` x βάζουμε `l = middle + 1`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`] <` x θα ισχύει για κάθε *k* στο [0..*middle*] ή στο [0..*middle*+1) που είναι το [0..*l*). Για την ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x δεν έχουμε αλλαγή.

Αν v`[`*middle*`]` ≥ x βάζουμε `r = middle`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`]` ≥ x θα ισχύει για κάθε *k* στο [*middle*..*last*] ή στο [*middle*..*last*+1) που είναι το [*r*..*last*+1). Για την ∀*k*: [0..*l*) • v`[`*k*`] <` x δεν έχουμε αλλαγή.

Μετά τη **while** θα έχουμε:

$$(l >= r)\ \&\&\ (\forall k:\ [0..l) \bullet v[k] < x)\ \&\&\ (\forall k:\ [r..last+1) \bullet v[k] \geq x)$$

Αφού $l \in [r..last+1)$ θα έχουμε: v`[`l`]` ≥ x. Παρομοίως, αφού l-1 ∈ [0..l) θα έχουμε: v`[`l-1`] <` x. Δηλαδή θα ισχύει η v`[`l-1`] <` $x \leq$ v`[`l`]`.

Για την απόδειξη του τερματισμού εξετάζουμε τη διαφορά $d = r - l$ που έχει θετική τιμή (από τη συνθήκη της **while**).

Από την $l <$ `r` παίρνουμε $l \leq$ *middle* `<` r (το "`=`" αριστερά χρειάζεται διότι η (*l*+*r*)`/`2 είναι ακέραιη διαίρεση.)

- Αν εκτελεσθεί η `l = middle + 1` η νέα τιμή διαφοράς $d' = r - (l+r)/2 - 1 < d$.
- Αν εκτελεσθεί η `r = middle` η νέα τιμή διαφοράς $d' = (l+r)/2 - l < d$.

Επομένως η $r - l$ παράγει –όσο εκτελείται η **while**– μια γνησίως φθίνουσα ακολουθία θετικών ακεραίων που δεν είναι δυνατόν να έχει άπειρο πλήθος όρων.

9.7 Ανακεφαλαίωση

Ένας πίνακας είναι μια ακολουθία τιμών ίδιου τύπου, όπως και ένα σειριακό αρχείο, αλλά:

- Ο πίνακας υλοποιείται στην κύρια μνήμη ενώ το αρχείο στη βοηθητική.
- Μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε στοιχείο του πίνακα με την ίδια ευκολία, ενώ στο σειριακό αρχείο για να πάμε στο ν-οστό στοιχείο πρέπει να περάσουμε από τα προηγούμενα $\nu - 1$.[7]
- *Μπορούμε να χειριστούμε οποιοδήποτε στοιχείο ενός πίνακα όπως μια μεταβλητή* ενώ για να χειριστούμε μια τιμή από ένα αρχείο θα πρέπει να τη φέρουμε στην κύρια μνήμη.

Τα ν στοιχεία ενός πίνακα αριθμούνται από 0..ν–1. Όλα τα στοιχεία του πίνακα έχουν το ίδιο όνομα και διακρίνονται μεταξύ τους από τον *δείκτη* που είναι ο αριθμός του στοιχείου μέσα σε αγκύλες.

Οι πίνακες χρησιμοποιούνται φυσικά για την επίλυση με ΗΥ μαθηματικών προβλημάτων όπου χρησιμοποιούνται πίνακες. Πέρα από αυτό όμως οι πίνακες διευκολύνουν τον προγραμματισμό σε περιπτώσεις που έχουμε πολλά ομοειδή (ίδιου τύπου) αντικείμενα που υφίστανται την ίδια επεξεργασία.

Η αναζήτηση τιμών σε έναν πίνακα είναι ένα πολύ συνηθισμένο πρόβλημα. Η ταχύτητα της αναζήτησης αυξάνεται σημαντικά αν τα δεδομένα μας είναι ταξινομημένα. Έτσι, έχουν επινοηθεί πολλοί αλγόριθμοι για ταξινόμηση.

---

[7] Αργότερα θα δούμε ότι μπορούμε να έχουμε και αρχεία *τυχαίας πρόσβασης*.

Ασκήσεις

Α Ομάδα

9-1 Έστω ένας πίνακας

```
double a[10];
```

Γράψε εντολές που θα τον «αναποδογυρίσουν». Δηλαδή να αντιμεταθέσουν τις τιμές των στοιχείων του:

(a[0] ↔ a[9]) (a[1] ↔ a[8]) (a[2] ↔ a[7]) (a[3] ↔ a[6]) (a[4] ↔ a[5])

9-2 Ένας τρόπος να δούμε «πόσο μεγάλος» είναι ένας μονοδιάστατος πίνακας A, με στοιχεία από 0 μέχρι $N - 1$, είναι να υπολογίσουμε το:

$$\|A\| = |A_0| + |A_1| + \ldots + |A_{N-1}|$$

Γράψε πρόγραμμα που θα διαβάζει τις τιμές των στοιχείων του A ($N = 5$) και θα υπολογίζει και θα τυπώνει το $\|A\|$.

Στη συνέχεια, αν $\|A\| \neq 0$, θα κάνει τον A *μοναδιαίο*, διαιρώντας όλα τα στοιχεία του δια $\|A\|$. Τέλος, θα τυπώνει τα στοιχεία του μοναδιαίου πίνακα.

Επανάλαβε τα παραπάνω αν: $\|A\| = \sqrt{A_0^2 + A_1^2 + \ldots + A_{N-1}^2}$.

Το ίδιο αν: $\|A\| = \max_{k=0..N-1} (|A_k|)$.

Τα μαθηματικά ονομάζουν τα τρία αποτελέσματα νόρμα-1, νόρμα-2 και νόρμα-∞ αντίστοιχα.

9-3 Λύσε ξανά την προηγούμενη άσκηση αφού γράψεις τρεις συναρτήσεις *norm*1, *norm*2 και *normInf* που υπολογίζουν τις τρεις νόρμες.

9-4 Τροποποίησε το πρόγραμμα συγχώνευσης για την περίπτωση που ο πίνακας w δίνεται ταξινομημένος σε *φθίνουσα* τάξη. **Προσοχή!** *Μόνον* ο w.

9-5 Γράψε μια

```
double vectorSumIf( const double x[], int n,
                    double lb, double ub )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία: $lb \le x[k] \le ub$.

Β Ομάδα

9-6 Έστω ότι έχουμε δύο πίνακες x, y με n στοιχεία. Λέμε ότι ο x *προηγείται λεξικογραφικά* του y αν υπάρχει κάποιο k, από 0 μέχρι $n-1$, τέτοιο ώστε να έχουμε: $x_k < y_k$ και $x_j == y_j$ για κάθε j από 0 μέχρι $k-1$.

Γράψε μια:

```
bool lt( int x[], int y[], int n)
```

που θα επιστρέφει τιμή:

- `true` αν ο x προηγείται λεξικογραφικά του y και
- `false` αν όχι.

9-7 *Μέθοδος των ελαχίστων τετραγώνων*. Παραλλαγή της Άσκ. 6-11. Γράψε πρόγραμμα που θα διαβάζει από ένα αρχείο text, με όνομα στο δίσκο lsq.txt, τα (x_k, y_k), $k = 0 .. N-1$ και θα υπολογίζει τα α και β. Σε κάθε γραμμή του αρχείου έχουμε ένα ζεύγος x_k, y_k. Υποθέτουμε το N δεν υπερβαίνει το 100. Οι τιμές των x_k, y_k που διαβάζονται θα αποθηκεύονται σε δύο πίνακες x, y με χώρο για 100 στοιχεία το πολύ.

Ένα μέτρο του πόσο καλή είναι η προσαρμογή ευθείας στα δεδομένα δίνεται από τον **συντελεστή πολλαπλής συσχέτισης** (multiple correlation coefficient) R^2 που ορίζεται ως εξής:

$$R^2 = \frac{\sum_{k=0}^{N-1}(\alpha x_k + \beta - < y >)^2}{\sum_{k=0}^{N-1}(y_k - < y >)^2} \quad \text{όπου: } <y> = \frac{1}{N}\sum_{k=0}^{N-1} Y_k$$

Η τιμή του R^2 είναι από 0 (τέλεια προσαρμογή) μέχρι 1. Συμπλήρωσε λοιπόν το πρόγραμμά σου ώστε μετά τον υπολογισμό των α και β να υπολογίζει και να γράφει το R^2.

***9-8** Απόδειξε ότι η *maxNdx* είναι σωστή, δηλαδή

```
// 0 <= from <= upto <= n-1
   mxp = from;
   m = from+1;
   while ( m <= upto ) )
   {
      if ( x[m] > x[mxp] )  mxp = m;
      m = m + 1;
   }
// from ≤ mxp ≤ upto && ∀j: from..upto • x[j] ≤ x[mxp]
```

Υπόδ.: Απόδειξε ότι η

$$from \le mxp \le m - 1 \;\&\&\; \forall j: from\,..\,m-1 \bullet \mathtt{x[}j\mathtt{]} \le \mathtt{x[}mxp\mathtt{]}$$

είναι αναλλοίωτη της **while**.

9-9 Γράψε μια:

```
int linSearchSrt( const int v, int n, int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον *ταξινομημένο*, σε αύξουσα τάξη, πίνακα v για την τιμή *x*. Απόδειξε ότι είναι σωστή.

9-10 Γράψε μια:

```
unsigned int linSearchMult( const int v, int n,
                            int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον πίνακα v –που υλοποιεί ένα *πολυσύνολο*– για τη *x* και θα επιστρέφει το *πλήθος* των στοιχείων του v που είναι ίσα με τη *x*.

9-11 Απόδειξε ότι το πρόγραμμα ταξινόμησης είναι σωστό, δηλαδή (στα `v[0]` και `v[`N`+1]` έχουμε τα $-\infty$ και $+\infty$)[8]:

```
// N > 0
   k = N;
   while ( k >= 2 ) // ∀j: k..N • v[j] ≤ v[j+1]
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
      k = k - 1;
   } // for
// ∀j: 1..N • v[j] ≤ v[j+1]
```

Γ Ομάδα

9-12 Έλυσες την Άσκ. 7-15; Λύσε τώρα και το πραγματικό πρόβλημα: Γράψε συνάρτηση

```
double mp( double a[], int N, double x )
```

που να υλοποιεί την παρακάτω συνάρτηση:

[8] Για να είναι σωστός ο αλγόριθμος θα πρέπει να αποδείξουμε ότι δεν κάνουμε εισαγωγή ή διαγραφή τιμών. Αφού όμως ο αλγόριθμός μας κάνει αντιμεταθέσεις τιμών στοιχείων μπορούμε να θεωρήσουμε ότι αυτό είναι εξασφαλισμένο.

$$m_p(x) = \prod_{k=0}^{N-1} \frac{1}{x-a_k} = \frac{1}{x-a_0} \times \frac{1}{x-a_1} \times \ldots \times \frac{1}{x-a_{N-1}}, \text{ όπου } N \geq 1$$

9-13 Έστω ότι έχουμε έναν πίνακα:

```
const int N = 50;

int v[N+2];
```

Στα v[0] και v[N+2] έχουμε τα $-\infty$ και $+\infty$ αντιστοίχως.

Στον πίνακα έχουμε ήδη τιμές στις θέσεις από v[1] μέχρι v[l] ($l < N$), ταξινομημένες κατ' αύξουσα τάξη. Θέλουμε να εισαγάγουμε μία ακόμη τιμή –της μεταβλητής x– στη θέση v[m] όπου $m \leq l + 1$, έτσι ώστε:

- να μη χάσουμε κάποια από αυτές που ήδη υπάρχουν,
- ο πίνακας να συνεχίσει να είναι ταξινομημένος κατ' αύξουσα τάξη.

9-14 (Συνέχεια της προηγούμενης) Με βάση τα παραπάνω μπορούμε να δούμε μιαν άλλη μέθοδο ταξινόμησης ενός πίνακα: την μέθοδο **κατ' ευθείαν εισαγωγής** (straight insertion sort):

```
for ( i = 2; i <= N; i = i+1 ) //I: το v[1]…v[i-1] ταξινομημένο
{
   Βάλε το v[i] στη σωστή θέση στο κομμάτι v[1]…v[i-1]
} // for
```

Άλλαξε το πρόγραμμα της ταξινόμησης ώστε να υλοποιεί αυτήν τη μέθοδο.

9-15 Μας δίνονται δύο αρχεία text, int1.txt και int2.txt, που περιέχουν ακέραιους αριθμούς ταξινομημένους κατ' αύξουσα τάξη. Γράψε πρόγραμμα που θα συγχωνεύσει τα περιεχόμενα των δύο αρχείων σε ένα (int3.txt) ταξινομημένο κατ' αύξουσα τάξη.

κεφάλαιο

10

Προγράμματα με Κείμενα

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να γράφεις προγράμματα που διαχειρίζονται μη-αριθμητικά δεδομένα. Να χρησιμοποιείς τις τεχνικές αυτές και σε αριθμητικά δεδομένα.

Προσδοκώμενα αποτελέσματα:

Είναι μάλλον απίθανο να γράφεις «πραγματικό» πρόγραμμα χωρίς να χρειαστεί να διαχειριστείς κείμενο, έστω και αν πρόκειται για κείμενο με αριθμούς. Θα μάθεις να γράφεις πιο «επαγγελματικά» προγράμματα.

Έννοιες κλειδιά:

- *ορμαθός χαρακτήρων*
- *τύπος (κλάση) string*
- *σύνδεση ορμαθών*
- *αναζήτηση υποορμαθού*
- *πίνακες χαρακτήρων*
- *μετατροπή ορμαθού σε αριθμό*
- *μετατροπή αριθμού σε ορμαθό*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Αν γράψεις στο πρόγραμμά σου:

```
cout << "πάν μέτρον εκατό πόντοι" << endl;
```

θα περιμένεις ως αποτέλεσμα, σύμφωνα με όσα έχουμε μάθει μέχρι τώρα, να δεις στην οθόνη

```
πάν μέτρον εκατό πόντοι
```

Ξέρουμε λοιπόν πώς να στείλουμε ένα κείμενο στην οθόνη μας μέσω του ρεύματος *cout*. Δεν ξέρουμε όμως πώς να αποθηκεύσουμε ένα κείμενο στη μνήμη ούτε πώς να το επεξεργαστούμε.

Η πιο απλή ιδέα, σύμφωνα με αυτά που ξέρουμε ήδη, είναι να δούμε το κείμενο ως πίνακα χαρακτήρων:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ','π','ό','ν','τ','ο','ι' };
```

Σωστό! Και αν δώσεις:

```
cout << a << endl;
```

θα δεις:

```
πάν μέτρον εκατό πόντοι
```

(είναι όμως πιθανό να δεις το κείμενό σου να ακολουθείται από μερικά «σκουπίδια».)

Η C++ σου επιτρέπει να κάνεις τη δήλωση πολύ πιο απλά:

```
char b[] = "πάν μέτρον εκατό πόντοι";
```

και αν δώσεις:

```
cout <<  b << endl;
```

θα δεις σίγουρα:

```
πάν μέτρον εκατό πόντοι
```

Έχουν διαφορά οι δύο δηλώσεις; Ναι. Αν ζητήσεις:

```
cout << (sizeof a) << "  " << (sizeof b) << endl;
```

θα πάρεις:

```
            23  24
```

Η διαφορά οφείλεται σε ένα `char(0)` (ή αλλιώς: `'\0'`), που μπαίνει ως τελευταίος χαρακτήρας στον b. Αυτό είναι κληρονομιά από τη C, που χρησιμοποιεί αυτόν τον χαρακτήρα ως *φρουρό* στους αλγόριθμους επεξεργασίας κειμένων της βιβλιοθήκης της. Το ότι στην πρώτη περίπτωση δεν έχουμε αυτόν τον `char(0)` έχει ως αποτέλεσμα να παίρνουμε τα «σκουπίδια» σε ορισμένες περιπτώσεις.

Η βασική ιδέα λοιπόν είναι αυτή: *Το κείμενο παριστάνεται ως πίνακας με στοιχεία τύπου* `char`. Η C έμεινε σε αυτό και απλώς περιλαμβάνει στις βιβλιοθήκες της πολλές συναρτήσεις που βοηθούν στην επεξεργασία τέτοιων πινάκων. Αυτές οι συναρτήσεις έχουν κληρονομηθεί και στη C++. Αλλά, στη C++ σχεδιάστηκε ένας ειδικός τύπος (κλάση) με το όνομα `string` (ορμαθός) που έχει ενσωματωμένες πολλές μεθόδους που μας χρειάζονται για να διαχειριστούμε ένα κείμενο.

Για να δουλέψεις με τον τύπο `string` θα πρέπει να περιλάβεις στο πρόγραμμά σου το αρχείο string ("`#include <string>`").

10.1 Δήλωση Μεταβλητών Τύπου *string*

Η πιο απλή περίπτωση δήλωσης είναι η εξής[1]:

[1] Αν δεν βάλεις το "`using namespace std`" θα πρέπει να δηλώνεις "`std::string`".

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s0;
```

Η *s0* έχει ως τιμή έναν ορμαθό *μηδενικού μήκους*, χωρίς περιεχόμενο. Στη συνέχεια μπορείς να του δώσεις την τιμή που θέλεις, όπως θα δούμε παρακάτω.

Πάντως, μπορείς να του δώσεις τιμή από την αρχή, με τη δήλωση:

```
   string s1( "ένας ορμαθός της C++" );
```

Να σου υπενθυμίσουμε εδώ, ότι αν ο ορμαθός που θα εκχωρήσεις ως τιμή έχει κάποιους ειδικούς χαρακτήρες θα πρέπει να χρησιμοποιήσεις τον χαρακτήρα "\" (δες τον Πίν. 4-3). Π.χ.:

```
   string path( "f:\\cwork\\prog7_1.cpp" );
   string quest( "what\'s this\?" );
```

Αν μετά τα παραπάνω δώσεις τις εντολές:

```
   cout << "    s0: " << s0 << endl;
   cout << "    s1: " << s1 << endl;
   cout << "  path: " << path << endl;
   cout << " quest: " << quest << endl;
```

θα πάρεις:

```
    s0:
    s1: ένας ορμαθός της C++
  path: f:\cwork\prog7_1.cpp
 quest: what's this?
```

Δεν μπορείς να δώσεις ως αρχική τιμή μια σταθερά τύπου **char**. Έτσι, η δήλωση:

```
   string a( 'a' );
```

είναι λάθος!

Μπορείς όμως να δώσεις:

```
   string a( 10,'a' );
```

οπότε η:

```
   cout << "     a: " << a << endl;
```

θα δώσει:

```
     a: aaaaaaaaaa
```

Δηλαδή η a πήρε ως τιμή έναν ορμαθό με 10 **'a'**. Αν λοιπόν θέλεις να δώσεις ως τιμή ένα **'a'** μπορείς να δώσεις ένα από τα παρακάτω:

```
   string a( 1,'a' );
   string a( "a" );
```

Μπορείς να δώσεις ως αρχική τιμή έναν πίνακα τύπου **char**:

```
   char b[] = "πάν μέτρον εκατό πόντοι";
   string s2( b );
```

Μπορείς όμως να δώσεις ως αρχική τιμή και την τιμή μιας άλλης μεταβλητής τύπου **string**:

```
   string s2( a );
```

ή έναν *υποορμαθό* μιας μεταβλητής τύπου **string**:

```
   string s3( s1, 5, 7 );
```

Με αυτήν τη δήλωση δίνουμε ως αρχική τιμή στην *s3* έναν υποορμαθό της *s1* που ξεκινάει από τον χαρακτήρα 5 (αρχίζουμε το μέτρημα από 0) και έχει μήκος 7 χαρακτήρες. Αν *s1* είναι αυτή που δηλώσαμε παραπάνω, τότε, αν ζητήσουμε να τυπωθεί η τιμή της *s3*, θα πάρουμε:

```
ορμαθός
```

10.2 Μετατροπή σε Απλό Πίνακα

Πριν προχωρήσουμε, ας δούμε πώς μπορείς να πάρεις το «περιεχόμενο» ενός αντικειμένου τύπου *string*, σε έναν πίνακα με στοιχεία τύπου `char`. Υπάρχουν δύο *μέθοδοι* γι' αυτό: η *c_str* και η *data* που κάνουν την ίδια –περίπου– δουλειά.

Ας πούμε ότι έχουμε:

```
const int sz = 50;
string s1( "ένας ορμαθός της C++" );
char ac[sz];
```

και παίρνουμε το `s1.c_str()`. Αυτό έχει `char(0)` στο τέλος. Έτσι, μπορείς να το χειριστείς με τις συναρτήσεις της C, π.χ. να τα αντιγράψεις σε έναν πίνακα σαν τον ac:

```
strcpy( ac, s1.c_str() );
```

Για τη *strcpy()* θα τα πούμε στη συνέχεια.

Ποια η διαφορά του `s1.data()`; *Δεν υπάρχει εγγύηση ότι θα έχει* `char(0)` *στο τέλος*. Έτσι, η

```
strcpy( ac, s1.data() );
```

δεν είναι σίγουρο ότι θα δουλέψει. Πάντως σε πολλές περιπτώσεις θα δεις να συμπεριφέρεται ακριβώς σαν το `s1.c_str()`.

Και για τις δύο συναρτήσεις υπάρχει ένας περιορισμός: *Μην προσπαθήσεις να τροποποιήσεις τους πίνακες που βγάζουν*. Για παράδειγμα μην προσπαθήσεις να κάνεις κάτι σαν:

```
(c1.c_str())[3] = 'Σ';
```

10.3 Εκχώρηση Τιμής και Αντιμετάθεση

Χρησιμοποιώντας τον τελεστή "`=`" της εκχώρησης, μπορείς να εκχωρήσεις σε μια μεταβλητή τύπου *string*:

- μια τιμή τύπου *string*,
- μια τιμή τύπου `char`,
- έναν ορμαθό χαρακτήρων

Έτσι, αν έχεις δηλώσει:

```
string s0;
string s1( "ένας ορμαθός της C++" );
char b[] = "πάν μέτρον εκατό πόντοι";
```

μπορείς να δώσεις:

```
s0 = s1;                cout << "    s0: " << s0 << endl;
s0 = 'a';               cout << "    s0: " << s0 << endl;
s0 = b[4];              cout << "    s0: " << s0 << endl;
s0 = "what\'s this\?";  cout << "    s0: " << s0 << endl;
```

και θα πάρεις από τις εντολές εξόδου:

```
 s0: ένας ορμαθός της C++
 s0: a
 s0: μ
 s0: what's this?
```

Εκτός όμως από το "`=`", μπορείς να εκχωρήσεις τιμή και με κάποια από τις μορφές της μεθόδου *assign()*: Ας πούμε ότι έχουμε δηλώσει:

```
string s1( "ένας ορμαθός της C++" );
char b[] = "0123456789";
string s2;
```

1. Μπορείς να δώσεις: "`s2.assign(s1)`" που είναι ίδιο με το "`s2 = s1`". Οι

```
s2.assign( s1 );              cout << s2 << endl;
```

θα δώσουν:

```
ένας ορμαθός της C++
```

2. Μπορείς να δώσεις: "`s2.assign(s1, p, n)`" που σημαίνει: Δώσε ως τιμή στην *s2* το κομμάτι της *s1* που έχει *n* (πλήθος) χαρακτήρες και ξεκινάει από τον χαρακτήρα *p*. Π.χ. οι

```
   s2.assign( s1, 5, 7 );          cout << s2 << endl;
```

θα δώσουν:

```
ορμαθός
```

3. Μπορείς να δώσεις: "`s2.assign(cs)`" όπου *cs* ορμαθός ή πίνακας με στοιχεία τύπου `char`. Π.χ. οι:

```
   s2.assign( b );                 cout << s2 << endl;
   s2.assign( "0123456789" );      cout << s2 << endl;
```

θα δώσουν:

```
0123456789
0123456789
```

3. Μπορείς να δώσεις: "`s2.assign(cs, n)`" όπου *cs* ορμαθός ή πίνακας με στοιχεία τύπου `char` και *n* ακέραιος, από 0 μέχρι το μήκος του *cs*. Στην *s2* εκχωρούνται οι *n* πρώτοι χαρακτήρες του *cs*. Π.χ. οι:

```
   s2.assign( b, 5 );              cout << s2 << endl;
   s2.assign( "0123456789", 5 );   cout << s2 << endl;
```

θα δώσουν:

```
01234
01234
```

4. Τέλος, μπορείς να δώσεις "`s2.assign(n, c)`" όπου *n* φυσικός και *c* τιμή τύπου `char`. Τιμή της *s2* είναι ένας ορμαθός με *n* φορές τον *c*. Π.χ. η:

```
   s2.assign( 5, 'a' );            cout << s2 << endl;
```

θα δώσει:

```
aaaaa
```

Αντιμετάθεση τιμών ξέρεις να κάνεις! Αν πρόκειται για τιμές τύπου *string* τα πράγματα είναι πιο εύκολα: υπάρχει σχετική μέθοδος που λέγεται *swap()*. Δες τις παρακάτω εντολές:

```
   cout << " s1: " << s1 << "  s2: " << s2 << endl;
   s2.swap( s1 );
   cout << " s1: " << s1 << "  s2: " << s2 << endl;
```

που δίνουν:

```
 s1: ένας ορμαθός της C++  s2: aaaaa
 s1: aaaaa  s2: ένας ορμαθός της C++
```

Δηλαδή: η "`s2.swap(s1)`" έχει ως αποτέλεσμα την αντιμετάθεση τιμών των μεταβλητών *s1* και *s2*. Αλλά, όπως θα δούμε αργότερα, η αντιμετάθεση με τη *swap()* είναι ασφαλέστερη από την αντιμετάθεση που ξέρουμε.

10.4 Ανάγνωση και Γραφή

Ας ξεκινήσουμε με το γράψιμο, για το οποίο έχουμε ήδη μιλήσει. Απλώς θα συμπληρώσουμε ότι ο τελεστής "`<<`" στέλνει τιμές μεταβλητών τύπου `string`, όχι μόνον μέσω του *cout* στην οθόνη, αλλά και μέσω οποιουδήποτε ρεύματος *ofstream* σε αρχείο text.

Πώς μπορούμε να διαβάσουμε την τιμή μιας μεταβλητής *a* τύπου `string` από το πληκτρολόγιο ή από κάποιο αρχείο; Δηλαδή, δεν μπορούμε να διαβάσουμε την τιμή της *a* από το *cin* με τον "`>>`"; Μπορούμε, αλλά... Ας κάνουμε μια δοκιμή. Δίνουμε:

```
   cout << "Τι εχεις να πεις;" << endl;
   cin >> a;
   cout << a << endl;
```

και έχουμε την εξής εκτέλεση:

```
Τι εχεις να πεις;
των φρονίμων τα παιδιά τα κάνει κρεμαστάρια<enter>
των
```

Δηλαδή, μόλις βρήκε κενό στάματησε το διάβασμα· το ίδιο θα γινόταν αν έβρισκε τέλος γραμμής (`'\n'`) ή στηλοθέτη (`'\t'`).

Η ανάγνωση γίνεται με την[2]:

```
getline( cin, a, '\n' );
```

που λέει τα εξής:

- διάβασε από το ρεύμα *cin*,
- αποθηκεύοντας αυτά που διαβάζεις στην *a*,
- μέχρι να βρεις τέλος γραμμής (`'\n'`).

Ως πρώτο όρισμα μπορείς να βάλεις και ρεύμα τύπου *ifstream*, δηλαδή μπορείς να διαβάζεις και από αρχείο.

Αν, ως απάντηση στην `getline(cin, a, '\n')`, πιέσεις απλώς το πλήκτρο <enter>, χωρίς να γράψεις κείμενο, η *a* γίνεται *κενή*.

Η *getline()* είναι ένα πολύ καλό εργαλείο και για την ανάγνωση αρχείων text. Ας πούμε ότι στο αρχείο `inpData.txt` έχουμε γραμμές της μορφής:

```
873\t537\tΑνδρικόπουλος\t2510 997 799\n
```

(με τα '`\t`' και '`\n`' συμβολίζουμε τους χαρακτήρες "tab" και "newline" αντιστοίχως.) Αν έχουμε δηλώσει:

```
ifstream tin( "inpData.txt" );
string s1, s2, s3, s4;
```

μπορούμε να διαβάσουμε μια γραμμή ως εξής:

```
getline( tin, s1, '\t' );
getline( tin, s2, '\t' );
getline( tin, s3, '\t' );
getline( tin, s4, '\n' );
```

Βέβαια, οι δύο ακέραιοι, στην αρχή, έχουν διαβαστεί στις *s1* και *s2* ως *string*. Στη συνέχεια θα δεις πώς μπορούμε να τις μετατρέψουμε σε `int`. Παρομοίως, αν έχουμε μια γραμμή με μια ημερομηνία

```
19.11.2008
```

μπορούμε να τη διαβάσουμε ως εξής:

```
getline( tin, s1, '.' );
getline( tin, s2, '.' );
getline( tin, s3, '\n' );
```

Αντί για την (*std*`::`)*getline()* μπορείς να αποθηκεύσεις αυτό που διαβάζεις σε ορμαθό της C –δηλαδή πίνακα χαρακτήρων– με τη *getline* της *cin*. Αν, ας πούμε, έχεις δηλώσει:

```
char q[100];
```

μπορείς να διαβάσεις κείμενο από το πληκτρολόγιο στον q και από εκεί να το αντιγράψεις στην *a*, ως εξής:

```
cin.getline( q, 100 );
a.assign( q );
```

Εδώ ζητούμε να διαβαστεί μια γραμμή από το πληκτρολόγιο και να αποθηκευτεί στον *q*· αλλά να διαβαστούν το πολύ 99 (= 100 – 1) χαρακτήρες. Στο μήκος αυτό (100) περιλαμβάνεται και μια θέση για το `char(0)` που θα σημειώνει το τέλος του κειμένου στον *q*. Στη συνέχεια, με την `a.assign(q)`, αντιγράφουμε το περιεχόμενο του *q* (χωρίς το `char(0)`) στην *a*.

[2] `std::getline` για την ακρίβεια...

Η απάντηση <enter> χωρίς κείμενο στη `cin.getline()` έχει παρόμοιο αποτέλεσμα με αυτό που είδαμε στην `std::getline`: ο *q* παραμένει «κενός» (με `char(0)` στη θέση q[0]).

Ας δούμε και ένα

Παράδειγμα

Θα κάνουμε πιο ευέλικτο το πρόγραμμα αντιγραφής αρχείων που είδαμε στο Κεφ. 8. Θα βάλουμε σε μεταβλητές τα ονόματα των αρχείων:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
   ifstream s;
   ofstream t;
   string sFlNm, tFlNm;
   char ch;

   cout << "όνομα αρχείου: ";  getline( cin, sFlNm, '\n' );
   s.open( sFlNm.c_str() );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο " << sFlNm << endl;
   else
   {
      cout << "όνομα αντιγράφου: ";  getline( cin, tFlNm, '\n' );
      t.open( tFlNm.c_str() );
      if ( t.fail() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το " << tFlNm << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
      {
         s.get( ch );
         while ( !s.fail() && !t.fail() )
         {  t.put( ch );  s.get( ch );  } // while
         if ( !s.eof() )
            cerr << "πρόβλημα κατά την αντιγραφή" << endl;
         t.close();
         s.close();
      } // if ( t.fail
   } // if ( s.fail
} // main
```

Όπως βλέπεις, βάλαμε μεν τα ονόματα των αρχείων σε μεταβλητές τύπου `string`, αλλά δεν μπορούμε να περάσουμε τα ονόματα των μεταβλητών στις *open* που περιμένουν ορμαθό χαρακτήρων ή πίνακα με στοιχεία τύπου `char`. Έτσι, περάσαμε τα `sFlNm.c_str()` και `tFlNm.c_str()`.

10.5 Συγκρίσεις

Μπορείς να συγκρίνεις δύο ορμαθούς μεταξύ τους είτε με τη μέθοδο *compare()* είτε με τους γνωστούς τελεστές (`<`, `>`, `==` κλπ).

Η "`s1.compare(s2)`" δίνει ακέραιη

- *αρνητική τιμή* αν η τιμή της *s1* προηγείται λεξικογραφικώς της τιμής της *s2*,
- *θετική* τιμή αν *η τιμή της s1 έπεται λεξικογραφικώς της τιμής της s2*,
- *μηδέν* (0) αν *οι τιμές των s1 και s2 είναι ίσες*.

Αν λοιπόν δηλώσουμε:

```
string s1( "abc" );
string s2( "bce" );
string s3( "abcd" );
string s4( "abc" );
```

οι:

```
cout << s1.compare(s2) << "  " << s2.compare(s1) << endl;
cout << s1.compare(s3) << "  " << s3.compare(s1) << endl;
cout << s1.compare(s4) << "  " << s4.compare(s1) << endl;
```

θα δώσουν:

```
-1  1
-1  1
0  0
```

που σημαίνουν:

- Η τιμή της *s1* ("**abc**") λεξικογραφικώς προηγείται της τιμής της *s2* ("**bce**"), ενώ
- η τιμή της *s2* λεξικογραφικώς έπεται της τιμής της *s1*. Παρομοίως,
- η τιμή της *s1* λεξικογραφικώς προηγείται της τιμής της *s3* ("**abcd**"), ενώ
- η τιμή της *s3* λεξικογραφικώς έπεται της τιμής της *s1*. Τέλος,
- οι τιμές των *s1* και *s4* είναι ίσες.

Μπορείς να συγκρίνεις και την τιμή μιας μεταβλητής τύπου **string** με έναν ορμαθό χαρακτήρων ή έναν πίνακα με στοιχεία τύπου **char**. Αν, για παράδειγμα

```
char a[] = "abd";
```

οι:

```
cout << s1.compare( "cdef" ) << endl;
cout << s2.compare( a ) << endl;
```

θα δώσουν:

```
-2
1
```

Μπορείς λοιπόν να γράφεις στο πρόγραμμά σου:

```
if ( s1.compare(s2) < 0 )
   cout << "το " << s1 << " προηγείται λεξικογραφικώς του "
        << s2 << endl;
else if ( s1.compare(s2) > 0 )
   cout << "το " << s1 << " έπεται λεξικογραφικώς του "
        << s2 << endl;
else
   cout << "τα " << s1 << " και " << s2
        << " είναι ίσα" << endl;
```

Πάντως μπορείς να χρησιμοποιήσεις και τους γνωστούς μας τελεστές σύγκρισης:

αντί για **s1.compare(s2)** θ **0** μπορείς να γράφεις **s1** θ **s2**

όπου θ κάποιος από τους "**>**", "**>=**", "**<**", "**<=**", "**==**", "**!=**".

Έτσι, μπορείς να γράψεις την παραπάνω διπλή **if** και ως εξής:

```
if ( s1 < s2 )
   cout << "το " << s1 << " προηγείται λεξικογραφικώς του "
        << s2 << endl;
else if ( s1 > s2 )
   cout << "το " << s1 << " έπεται λεξικογραφικώς του "
        << s2 << endl;
else
   cout << "τα " << s1 << " και " << s2
        << " είναι ίσα" << endl;
```

όπως και:

```
if ( s1 == "abc" )  cout << "eq" << endl;
```

Ας δούμε ένα παράδειγμα χρήσης της δυνατότητας σύγκρισης.

Παράδειγμα

Έχουμε έναν πίνακα με στοιχεία τύπου `string`:

```
   string v[N] = { "PASCAL", "BASIC", "FORTRAN", "COBOL", "RPG",
                   "ALGOL", "LISP", "PROLOG", "LOGO", "C++",
                   "PL/I", "ADA", "BCPL", "SNOBOL", "APL", "C" };
```

και θέλουμε να ταξινομήσουμε τα στοιχεία κατ' αλφαβητική σειρά.

Μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο που αντιγράφουμε από την §9.5.2:

```
// ταξινόμηση
   for ( k = N; k >= 2; k = k-1 )
   {
      meg = maxNdx( v, 1, k );
      sv = v[meg];  v[meg] = v[k];  v[k] = sv;
   } // for
```

μόνο που εδώ τα στοιχεία μας είναι στις θέσεις[3] v[0] μέχρι v[*N*-1]. Ακόμη η αντιμετάθεση τιμών των v[*mxp*] και v[*k*] μπορεί να γίνει με τη *swap*:

```
// ταξινόμηση
   for ( k = N-1; k >= 1; k = k-1 )
   {
      mxp = maxNdx( v, N, 0, k );
      v[mxp].swap( v[k] );
   } // for
```

Και με τη *maxNdx* τι γίνεται; Την αντιγράφουμε όπως είναι αλλάζοντας μόνον τον τύπο των στοιχείων του πίνακα από `int` σε *string*:

```
   int maxNdx( const string x[], int n, int from, int upto )
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <string>
using namespace std;

   int maxNdx( const string x[], int n, int from, int upto );

int main()
{
   const int N = 16;

   string v[N] = { "PASCAL", "BASIC", "FORTRAN", "COBOL", "RPG",
                   "ALGOL", "LISP", "PROLOG", "LOGO", "C++",
                   "PL/I", "ADA", "BCPL", "SNOBOL", "APL", "C" };
   int  k, mxp;
// ταξινόμηση
   for ( k = N-1; k >= 1; k = k-1 )
   {
      mxp = maxNdx( v, N, 0, k );
      v[mxp].swap( v[k] );
   } // for
// αποτέλεσμα
   for ( k = 0; k <= N-1; k=k+1 )  cout << v[k] << endl;
} // main
```

Αποτέλεσμα:

```
ADA
ALGOL
APL
```

---

[3] Φυσικά δεν υπάρχει πρόβλημα αν θέλεις να βάλεις φρουρούς στην αρχή και στο τέλος. Άλλαξε τη δήλωση σε:

```
   string v[N+2] = { " ", "PASCAL",... "C", " " };
```

και μετά δώσε:

```
   v[0].assign(5, char(0));  v[N+1].assign(5,char(255));
```

```
BASIC
BCPL
C
C++
COBOL
FORTRAN
LISP
LOGO
PASCAL
PL/I
PROLOG
RPG
SNOBOL
```

10.6 Μήκος Ορμαθού – Κενός Ορμαθός

Αφού τιμή μιας μεταβλητής τύπου **string** είναι ένας ορμαθός χαρακτήρων, έχει νόημα το μήκος της, που είναι ακριβώς το μήκος της τιμής της. Το μήκος μας δίνεται από τη μέθοδο **length** (μήκος).

Ας πούμε ότι έχουμε δηλώσει:

```
string s0, s1( "ένας ορμαθός της C++" );
string path( "f:\\cwork\\prog7_1.cpp" ),
       quest( "what\'s this\?" );
string a( 10,'a' ), c( "a" );
```

και δίνουμε:

```
cout << s0.length() << "  " << s1.length() << "  "
     << path.length() << "  " << quest.length() << "  "
     << a.length() << "  " << c.length() << endl;
```

Αποτέλεσμα:

```
0  20  20  12  10  1
```

Πρόσεξε ότι στην *path* το "\\" μετράει ως *ένας* χαρακτήρας. Ως ένας χαρακτήρας μετράνε και τα "\'" και "\?" στην *quest*.

Εκτός από τη *length()* υπάρχει και η *size()* που κάνει ακριβώς τα ίδια. Θα πάρεις τα παραπάνω αποτελέσματα και με τη:

```
cout << s0.size() << "  " << s1.size() << "  "
     << path.size() << "  " << quest.size() << "  "
     << a.size() << "  " << c.size() << endl;
```

Πρόσεξε ακόμη ότι το μήκος της *s0* είναι μηδέν. Η τιμή της *s0* είναι ο κενός ορμαθός, χωρίς κάποιον χαρακτήρα. Αυτό ελέγχεται με τη μέθοδο *empty()*, που μας δίνει τιμή **true** αν η τιμή της μεταβλητής είναι ο κενός ορμαθός (μηδενικό μήκος) και **false** αν έχει έστω και έναν χαρακτήρα. Π.χ. οι:

```
if ( s0.empty() )  cout << " το s0 είναι άδειο!" << endl;
             else  cout << " το s0 κάτι έχει μέσα" << endl;
if ( s1.empty() )  cout << " το s1 είναι άδειο!" << endl;
             else  cout << " το s1 κάτι έχει μέσα" << endl;
```

θα δώσουν:

```
το s0 είναι άδειο!
το s1 κάτι έχει μέσα
```

10.6.1 Το Μέγιστο Μήκος Ορμαθού

Και ποιο είναι το μέγιστο μήκος ορμαθού που μπορώ να αποθηκεύσω σε μεταβλητή *string*; Το βρίσκεις με τη *max_size()*:

```
string s1;
```

```
   cout << "max_size = " << s1.max_size() << endl;
```

Αποτέλεσμα (BC++ v.5.5):

```
max_size = 4294967281
```

10.6.2 Ο Τύπος "size_type"

Ο τύπος του αποτελέσματος της *length()*, της *size()* και της *max_size()* είναι *string*::*size_type*, που είναι ο τύπος της C++ για τα μεγέθη των αντικειμένων που παριστάνονται στον τύπο *string*.

Για παρόμοιες δουλειές η C μας κληρονομεί τον **size_t** που είναι ο τύπος του αποτελέσματος που επιστρέφει ο τελεστής "**sizeof**". Αν ψάξεις το **cstddef** (ή το **stddef.h**) θα βρεις τον ορισμό:

```
typedef unsigned int size_t;
```

ή κάτι παρόμοιο.

Μπορείς να θεωρείς ότι οι δυο τύποι είναι ταυτόσημοι.[4]

10.7 Σύνδεση – Επισύναψη

Πολύ συχνά έχουμε ανάγκη να **συνδέσουμε** (concatenate) δυο ή περισσότερους ορμαθούς. Αυτό μπορούμε να το κάνουμε «προσθέτοντας» ορμαθούς με το "**+**". Αν, για παράδειγμα, έχουμε δηλώσει:

```
   string s1( "πέντε" ), s2( "στο χέρι παρά δέκα" ), s3;
```

τότε οι εντολές:

```
   cout << ("κάλλιο " + s1 + " και " + s2 + " και καρτέρει")
        << endl;
   s3 = "κάλλιο " + s1 + " και " + s2 + " και καρτέρει";
   cout << s3 << endl;
```

θα μας δώσουν:

```
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

Με τον τελεστή "**+**" μπορούμε να συνδέσουμε:

- δύο μεταβλητές τύπου **string**,
- μεταβλητή τύπου **string** και ορμαθό χαρακτήρων,
- μεταβλητή τύπου **string** και πίνακα με στοιχεία τύπου **char** (το περιεχόμενό του θα πρέπει να τελειώνει με '**\0**'.)

Το αποτέλεσμα της πράξης μπορεί να εκχωρηθεί σε μια μεταβλητή τύπου **string**.

Πολύ συχνά υπάρχει ανάγκη να **επισυνάψουμε** (append) έναν ορμαθό στην τιμή μιας μεταβλητής τύπου **string**. Π.χ. αυτά που κάναμε στο παράδειγμα θα μπορούσαν να γίνουν και ως εξής:

```
   s3 = "κάλλιο ";
   s3 = s3 + s1;  s3 = s3 + " και ";  s3 = s3 + s2;
   s3 = s3 + " και καρτέρει";
```

Για την επισύναψη ο **string** έχει ειδική μέθοδο, την *append()*. Τα ίδια πράγματα θα μπορούσαν να γραφούν και ως εξής[5]:

---

[4] Πάντως, για να μην χρησιμοποιεί η C++ τον "**size_t**" διατηρεί το δικαίωμα για αλλαγές.

[5] Αλλά και ως εξής:

```
   s3 = "κάλλιο ";   s3 += s1;  s3 += " και ";
                     s3 += s2;  s3 += " και καρτέρει";
```

```
s3 = "κάλλιο ";
s3.append( s1 );  s3.append( " και " );  s3.append( s2 );
s3.append( " και καρτέρει" );
```

10.8 Αναζήτηση

Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο. Αυτό το ξέρεις καλά, αφού σίγουρα έχεις χρησιμοποιήσει πολλές φορές την εντολή **find** (ή **αναζήτησε**) του κειμενογράφου σου. Η κλάση `string` είναι εφοδιασμένη με αρκετές μεθόδους για τη δουλειά αυτή.

Ας ξεκινήσουμε με τη `find` (βρες). Αν έχεις δηλώσει "`string s`" και αν *t* είναι

- μια μεταβλητή τύπου `string` ή
- ένας ορμαθός χαρακτήρων ή
- πίνακας χαρακτήρων ή
- τιμή τύπου `char`

τότε

- Η `s.find(`*t*`)` ψάχνει την τιμή της *s*, *από αριστερά προς τα δεξιά*, για να εντοπίσει τον «υποορμαθό» *t*. Αν τον βρει επιστρέφει τον δείκτη του χαρακτήρα στην *s*, που αρχίζει ο *t*.
- Η `s.find(`*t*, *i*`)`, όπου *i* τιμή τύπου *string*`::`*size_type*, ψάχνει την τιμή της *s*, *από αριστερά προς τα δεξιά*, ξεκινώντας από τιμή δείκτη *i*, για να εντοπίσει τον «υποορμαθό» *t*. Αν τον βρει επιστρέφει τον δείκτη, στην *s*, που αρχίζει ο *t*.

Αν δεν βρεθεί η *t* μέσα στην *s* η *find* επιστρέφει μια απίθανη τιμή. Αυτήν την τιμή μπορείς να την ελέγχεις ως *string*`::`*npos*. Ο τύπος του αποτελέσματος είναι *string*`::`*size_type*.

Παράδειγμα

Γράφει ο Ποιητής:

ΑΞΙΟΝ ΕΣΤΙ στὸ πέτρινο πεζούλι
ἀντικρὺ τοῦ πελάγους ἡ Μυρτὼ νὰ στέκει
σὰν ὡραῖο ὀκτὼ ἢ σὰν κανάτι
μὲ τὴν ψάθα τοῦ ἥλιου στὸ ἕνα χέρι

Μπορούμε να αποθηκεύσουμε αυτό το κείμενο σε μια μεταβλητή

```
string text1;
```

ως εξής: [6]

```
text1 = "ΑΞΙΟΝ ΕΣΤΙ στο πέτρινο πεζούλι\n"
        "αντικρύ του πελάγους η Μυρτώ να στέκει\n"
        "σαν ωραίο οκτώ ή σαν κανάτι\n"
        "με την ψάθα του ήλιου στο ένα χέρι";
```

Με τα `'\n'` αποθηκεύουμε και τις αλλαγές γραμμής.

Δηλώνουμε ακόμη:

```
char a[] = "Μυρτώ";
```

Οι εντολές:

```
if ( text1.find("Μυρτώ") == string::npos )
```

Λέγαμε στο Κεφ. 2 ότι, για το αριθμητικό νόημα του "`+`", "`v += a`" σημαίνει: "`v = v + a`". Το ίδιο ισχύει και όταν ο τελεστής "`+`" σημειώνει τη σύνδεση.

[6] Μπορείς να το γράψεις και έτσι:

```
text1 = "ΑΞΙΟΝ ΕΣΤΙ στο πέτρινο πεζούλι\n";
text1.append( "αντικρύ του πελάγους η Μυρτώ να στέκει\n" );
text1.append( "σαν ωραίο οκτώ ή σαν κανάτι\n" );
text1.append( "με την ψάθα του ήλιου στο ένα χέρι" );
```

```
      cout << "Μυρτώ δεν βρέθηκε" << endl;
   else
      cout << text1.find("Μυρτώ") << endl;
   if ( text1.find("Μυρσίνη") == string::npos )
      cout << "Μυρσίνη δεν βρέθηκε" << endl;
   else
      cout << text1.find( "Μυρσίνη" ) << endl;
   cout << text1.find(a) << endl;
   cout << text1.find('M') << endl;
   cout << text1.find(a[0]) << endl;
```

θα δώσουν:

```
54
Μυρσίνη δεν βρέθηκε
54
54
54
```

Δηλαδή:

- Η λέξη `"Μυρτώ"` υπάρχει στη θέση με δείκτη 54 (γραμμές 1 και 3).
- Το γράμμα `'M'` υπάρχει στη 54 (γραμμές 4 και 5).
- Η λέξη `"Μυρσίνη"` *δεν* υπάρχει στην *text1* (γραμμή 2).

Παρατήρηση:▶

Για όσους δεν το πρόσεξαν (και δεν θύμωσαν) ήδη: Ο σωστός τρόπος να γράψουμε το πρόγραμμα είναι:

```
   size_t pos;
// . . .
   pos = text1.find("Μυρτώ");
   if ( pos == string::npos ) cout << "Μυρτώ δεν βρέθηκε" << endl;
                         else cout << pos << endl;
   pos = text1.find("Μυρσίνη");
   if ( pos == string::npos ) cout << "Μυρσίνη δεν βρέθηκε" << endl;
                         else cout << pos << endl;
   cout << text1.find(a) << endl;
   cout << text1.find('M') << endl;
   cout << text1.find(a[0]) << endl;
```

Η αναζήτηση κειμένου είναι χρονοβόρα διαδικασία και δεν αφήνουμε τέτοια πράγματα στην «καλή θέληση» (και ευφυΐα) του μεταγλωττιστή. Στο παράδειγμά μας το κείμενο δεν έχει ούτε 100 χαρακτήρες· άρα μικρό το κακό. Σε «πραγματικές συνθήκες» όμως…◀

Οι εντολές:

```
   cout << text1.find( "του" ) << endl;
   cout << text1.find( "του", 40 ) << endl;
```

θα δώσουν:

```
39
110
```

Δηλαδή: η λέξη "`του`" υπάρχει για πρώτη φορά στη θέση με δείκτη 39. *Ξεκινώντας την αναζήτηση από τη θέση* με δείκτη 40, η λέξη "`του`" *βρίσκεται για πρώτη φορά* με δείκτη 110.

Ας πάρουμε το κείμενο:

Νῦν ἡ ταπείνωση τῶν Θεῶν Νῦν ἡ σποδὸς τοῦ Ἀνθρώπου
Νῦν Νῦν τὸ μηδὲν

καὶ Αἰὲν ὁ κόσμος ὁ μικρὸς, ὁ Μέγας!

που το αποθηκεύουμε στη μεταβλητή:

```
   string text2;
// . . .
   text2 = "Νυν η ταπείνωση των θεών Νυν η σποδός του Ανθρώπου\n"
           "Νυν Νυν το μηδέν\n\n"
           "και Αιέν ο κόσμος ο μικρός, ο Μέγας!";
```

Αν έχουμε δηλώσει:

```
int ndx;
```

οι παρακάτω εντολές μας δίνουν όλους τους δείκτες όλων των θέσεων που αρχίζει η λέξη "Νυν":

```
ndx = text2.find( "Νυν" );
while ( 0 <= ndx && ndx < text2.length() )
{
   cout << ndx << "  ";
   ndx = text2.find( "Νυν", ndx+1 );
}
cout << endl;
```

Αποτέλεσμα:

```
0  25  51  55
```

Μια άλλη μέθοδος, η *rfind()*, είναι όμοια με τη *find()*, με μόνη διαφορά ότι ψάχνει *από δεξιά προς τα αριστερά* (από το τέλος προς την αρχή). Έτσι, για τα στοιχεία του παραπάνω παραδείγματος, οι εντολές:

```
cout << text1.rfind("του") << endl;
cout << text1.rfind("του", 108) << endl;
```

θα δώσουν:

```
110
39
```

Δες τώρα ένα άλλο πρόβλημα αναζήτησης: Ας πούμε ότι έχουμε την *text2* με την τιμή που έχει στο παράδειγμα και θέλουμε να βρούμε το πρώτο τονούμενο πεζό φωνήεν στο κείμενο. Μπορούμε να δώσουμε το εξής:

```
cout << text2.find_first_of("άέήίΐόύΰώ") << endl;
```

που θα μας δώσει:

```
10
```

Πράγματι, στη θέση με δείκτη 10 του κειμένου μας υπάρχει το 'ΐ' (ταπείνωση). Μπορούμε να συνεχίσουμε την αναζήτηση:

```
cout << text2.find_first_of("άέήίΐόύΰώ", 11) << endl;
```

Αποτέλεσμα:

```
22
```

Στη θέση με δείκτη 22 του κειμένου μας υπάρχει το 'ώ' (Θεών).

Δηλαδή, η μέθοδος *find_first_of()* μας δίνει *τον δείκτη της πρώτης θέσης όπου υπάρχει χαρακτήρας από κάποιο σύνολο*, που το δίνουμε *ως πρώτο όρισμα σε μορφήν ορμαθού*. Αν θέλουμε η αναζήτηση να μην ξεκινήσει από την αρχή, δίνουμε και δεύτερο όρισμα με τον δείκτη της επιθυμητή θέσης εκκίνησης.

Ό,τι είναι η *rfind()* για τη *find()* είναι η *find_last_of()* για τη *find_first_of()*: ψάχνει για τον τελευταίο χαρακτήρα που να ανήκει σε κάποιο σύνολο ή, αλλιώς, ψάχνει από δεξιά προς τα αριστερά.

Παρόμοια δουλειά κάνει και η *find_first_not_of()* η οποία ψάχνει για τον πρώτο χαρακτήρα που *δεν ανήκει* στο σύνολο-όρισμα. Η αντίστοιχη μέθοδος που ψάχνει από το τέλος είναι η *find_last_not_of()*.

Παρατήρηση: ▶

Θα πεις: «Καλά αυτά, αλλά ο Ποιητής γράφει πολυτονικό και εδώ κατακρεουργήθηκε!» και θα έχεις δίκιο! Αυτό όμως έγινε για να δείξουμε αυτά που θέλουμε με απλό και γρήγορο τρόπο. Υπάρχει ένας άλλος τύπος, ο *wstring*, ακριβώς σαν τον *string* αλλά με βάση τον `wchar_t`. Με αυτόν τον τύπο μπορείς να χειριστείς όλα τα σύμβολα του Unicode, επομένως και το πολυτονικό μας.◀

10.9 Αντικατάσταση – Διαγραφή – Εισαγωγή

Μια άλλη δουλειά, πολύ χρήσιμη, που την ξέρεις και αυτήν από τον κειμενογράφο σου (**replace** ή **αντικατάστησε**), είναι η αντικατάσταση ενός ορμαθού από έναν άλλον. Για μεταβλητές τύπου `string` μπορούσε να την επιτύχουμε με τη μέθοδο *replace()*. Δες ένα παράδειγμα.

Ας πούμε ότι έχουμε:

```
string target( "ενός" ), text1;
text1 ="Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου\n"
       "είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή\n")
       "ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.";
```

και δίνουμε τις εντολές:

```
cout << text1 << endl << endl;
text1.replace( text1.find(target), target.length(), "ΚΑΠΟΙΟΥ");
cout << text1 << endl;
```

Αποτέλεσμα:

```
Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή
ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.

Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ΚΑΠΟΙΟΥ χαρακτήρα ή μιας λέξης ή
ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.
```

Οι παράμετροι της `replace` στο παράδειγμά μας είναι οι εξής:

- Η πρώτη είναι ένας φυσικός που μας δίνει τον δείκτη της θέσης από την οποίαν ξεκινάει το προς αντικατάσταση κείμενο. Στην περίπτωσή μας βάλαμε τη θέση της τιμής του *text*1 που αρχίζει η τιμή της μεταβλητής *target* (δηλαδή τη λέξη "`ενός`") για πρώτη φορά (`text1.find(target)`).
- Η δεύτερη είναι και πάλι ένας φυσικός που δίνει το *μήκος* του κειμένου που θα αντικατασταθεί. Στην περίπτωσή μας βάλαμε το μήκος της τιμής της μεταβλητής *target* (δηλαδή της λέξης "`ενός`" που είναι 4).
- Η τρίτη παράμετρος είναι ο ορμαθός με τον οποίον θα γίνει η αντικατάσταση. Μπορεί να είναι ακόμη: τιμή τύπου `char` ή πίνακας χαρακτήρων ή μεταβλητή τύπου *string*.

Όπως βλέπεις τα μήκη των δύο ορμαθών ("`ενός`" και "`ΚΑΠΟΙΟΥ`") δεν είναι απαραίτητο να είναι ίσα. Αυτό φυσικά έχει ως συνέπεια να αλλάξει και το μήκος του κειμένου μας.

Αν θέλουμε να αντικαταστήσουμε όλες τις λέξεις "`ενός`" που θα βρούμε, δουλεύουμε όπως στην πολλαπλή *find*:[7]

```
ndx = text1.find( target );
while ( 0 <= ndx && ndx < text1.length() )
{
   text1.replace( ndx, target.length(), "ΚΑΠΟΙΟΥ" );
   ndx = text1.find( target, ndx+1 );
}
```

και παίρνουμε:

```
Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ΚΑΠΟΙΟΥ χαρακτήρα ή μιας λέξης ή
ΚΑΠΟΙΟΥ κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.
```

Με τη *replace()* μπορείς να κάνεις και *διαγραφή*. Αν δώσεις, π.χ.:

```
text1.replace( text1.find(target), target.length(), "" );
```

ζητάς να αντικατασταθεί η τιμή της *target*, την πρώτη φορά που θα βρεθεί στην τιμή της *text*1, με τον κενό ορμαθό, δηλ. να διαγραφεί.

---

[7] Το "`ndx+1`" θα δουλέψει συνήθως. Γενικώς, είναι λάθος. Το σωστό είναι "`ndx+μήκος("ΚΑΠΟΙΟΥ")`". Σκέψου το λιγάκι!

Υπάρχει όμως και μέθοδος *erase()* που κάνει διαγραφές. Στη συνήθη της μορφή παίρνει δύο ορίσματα: το πρώτο είναι ο δείκτης της θέσης από την οποίαν αρχίζει η διαγραφή και το δεύτερο είναι το πλήθος των χαρακτήρων που θα διαγραφούν. Μπορούμε λοιπόν να γράψουμε:

```
text1.erase( text1.find(target), target.length() );
```

Φυσικά, εκτός από αντικατάσταση και διαγραφή μπορείς να κάνεις και εισαγωγή με τη μέθοδο *insert()*. Είναι σαν την *append()* με τη διαφορά ότι:

- με την *append()* η επισύναψη γίνεται πάντοτε στο τέλος, ενώ
- με την *insert()* η εισαγωγή γίνεται όπου θέλουμε· το καθορίζουμε με την τιμή που δίνουμε στην πρώτη παράμετρο.

Ας ξαναδούμε το παράδειγμα που δώσαμε για την *append()*, αλλά κάπως πιο «ανακατεμένο». Είχαμε:

```
string s1( "πέντε" ), s2( "στο χέρι παρά δέκα" ), s3;
```

και δίνουμε:

```
s3 = " και καρτέρει";      cout << s3 << endl;
s3.insert( 0, s1 );        cout << s3 << endl;
s3.insert( 5, s2 );        cout << s3 << endl;
s3.insert( 0, "κάλλιο " ); cout << s3 << endl;
s3.insert( 12, " και " );  cout << s3 << endl;
```

Αποτέλεσμα:

```
 και καρτέρει
πέντε και καρτέρει
πέντεστο χέρι παρά δέκα και καρτέρει
κάλλιο πέντεστο χέρι παρά δέκα και καρτέρει
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

10.10 Διαχείριση Χαρακτήρων

Όπως στους πίνακες, έτσι και στις μεταβλητές τύπου *string*, μπορείς να διαχειρίζεσαι κάθε χαρακτήρα βάζοντας μετά από το όνομα της μεταβλητής τον κατάλληλο δείκτη. Αν έχεις δηλώσει:

```
string s3;
```

μπορείς να γράψεις για παράδειγμα:

```
if ( s3[0] == 'κ' )  s3[0] = 'K';
```

Βέβαια, χρειάζεται προσοχή: είναι δική σου ευθύνη να διασφαλίσεις ότι ο δείκτης θα παίρνει τιμές από 0 μέχρι `s3.length()-1`.

Αυτά τα πράγματα σου θυμίζουν πολύ τους πίνακες· αλλά η ομοιότητα σταματάει εδώ: Μια τιμή τύπου *string* κρύβει σίγουρα κάποιον πίνακα με τιμές τύπου `char` αλλά δεν είναι μόνον αυτό.

Εκτός από τη χρήση δείκτη, υπάρχει και άλλος τρόπος διαχείρισης των χαρακτήρων μιας μεταβλητής τύπου *string*: η μέθοδος `at`. Αυτό που γράψαμε παραπάνω μπορεί να γραφεί και ως εξής:

```
if ( s3.at(0) == 'κ' )  s3.at( 0 ) = 'K';
```

Αυτός ο τρόπος είναι ασφαλέστερος από αυτόν με τον δείκτη, διότι αν κάνεις λάθος και βγεις έξω από τα όρια της μεταβλητής σου θα ειδοποιηθείς, όπως θα μάθουμε αργότερα.

Παράδειγμα

Το παρακάτω πρόγραμμα, αντιστρέφει ένα κείμενο, που δίνεται ως στοιχείο εισόδου. Αν η αντίστροφη γραφή συμπίπτει με την ορθή, το πρόγραμμα εκφράζει τον ενθουσιασμό του! Στο πρόγραμμα χρησιμοποιούνται οι μέθοδοι:

- *at()*, για τη διαχείριση τιμής μιας μεταβλητής *string*, χαρακτήρα προς χαρακτήρα (`strIn.at(k)`)· για την ίδια δουλειά χρησιμοποιείται και η
- *length()*, διότι θέλουμε να επεξεργαστούμε όλους τους χαρακτήρες, από το τέλος προς την αρχή (`for (k = strIn.length()-1; k>=0; k=k-1)`),
- *compare()*, για τη σύγκριση τιμών (`strIn.compare(strOut) == 0`) δύο μεταβλητών τύπου `string`· σύγκριση όμως γίνεται και με τον τελεστή `!=` (`strIn != "T" && strIn != "t" && ...`),
- *append()*, για να κτίσουμε την τιμή μιας μεταβλητής τύπου *string*, χαρακτήρα προς χαρακτήρα. Για αυτήν συζητούμε και παρακάτω.

Πώς δουλεύει το πρόγραμμα; Αφού πάρουμε στη *strIn* το κείμενο που θα χειριστούμε αντιγράφουμε στη *strOut* έναν προς ένα τους χαρακτήρες της *strIn* αλλά από το τέλος προς την αρχή. Αυτό γίνεται με τη

```
      for ( k = strIn.length()-1; k >= 0; k = k-1 )
```

Έτσι αποκλείεται να βγούμε έξω από όρια και μπορούμε να χρησιμοποιήσουμε είτε τον δείκτη (`strIn[k]`) είτε την *at* (`strIn.at(k)`). Ποιο είναι το πρόβλημα με την *append*; Και οι δύο τρόποι μας δίνουν έναν χαρακτήρα και η *append* δεν δέχεται τέτοια παράμετρο. Και ποια λύση δίνουμε; Δηλώνουμε μια μεταβλητή

```
   string str1( "x" );
```

Το 'x' »κρατάει τη θέση» (place holder) για να έχουμε ορμαθό με μήκος 1. Στη συνέχεια το αντικαθιστούμε με κάθε χαρακτήρα που παίρνουμε από τη *strIn*:

```
         str1[0] = strIn.at(k);
```

Φυσικά αυτή δεν είναι η μοναδική λύση. Στην επόμενη παράγραφο θα δούμε μια καλύτερη.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   const string msg( "ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): " );

   int k;
   string strIn, strOut, str1( "x" );

   cout << msg;  getline( cin, strIn, '\n' );
   while ( strIn != "T" && strIn != "t" &&
           strIn != "Τ" && strIn != "τ" )
   {
      strOut = "";    // κενό
      for ( k = strIn.length()-1; k >= 0; k=k-1 )
      {
         str1[0] = strIn.at( k );
         strOut.append( str1 );
      }
      cout << " Η ανάποδη γραφή της: " << strIn
           << " είναι: " << endl;
      cout << strOut << endl;
      if ( strIn == strOut )
         cout << " Καλό αυτό!" << endl;
      cout << endl;
      cout << msg;  getline( cin, strIn, '\n' );
   } // while
   cout << " ΤΕΛΟΣ" << endl;
} // main
```

Να ένα παράδειγμα διαλόγου με το πρόγραμμα αυτό. Με κεκλιμένα οι απαντήσεις του χρήστη:

```
ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ABCDEFGHJIK
 Η ανάποδη γραφή της: ABCDEFGHJIK είναι:
```

```
KIJHGFEDCBA

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ΝΙΨΟΝ ΑΝΟΜΗΜΑΤΑ ΜΗ ΜΟΝΑΝ ΟΨΙΝ
 Η ανάποδη γραφή της: ΝΙΨΟΝ ΑΝΟΜΗΜΑΤΑ ΝΗ ΜΟΝΑΝ ΟΨΙΝ είναι:
ΝΙΨΟ ΝΑΝΟΜ ΗΝ ΑΤΑΜΗΜΟΝΑ ΝΟΨΙΝ

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ
 Η ανάποδη γραφή της: ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ είναι:
ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ
 Καλό αυτό!

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): MADAM IN EDEN I'M ADAM
 Η ανάποδη γραφή της: MADAM IN EDEN I'M ADAM είναι:
MADA M'I NEDE NI MADAM

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): MADAMINEDENIMADAM
 Η ανάποδη γραφή της: MADAMINEDENIMADAM είναι:
MADAMINEDENIMADAM
 Καλό αυτό!

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): Τ
 ΤΕΛΟΣ
```

10.11 Υποορμαθοί

Μερικές φορές θέλουμε να πάρουμε έναν **υποορμαθό** (substring), ένα κομμάτι ενός ορμαθού, για να το χειριστούμε χωριστά. Η μέθοδος *substr()* μας δίνει αυτήν τη δυνατότητα: μας δίνει μια νέα τιμή τύπου *string* με περιεχόμενο έναν υποορμαθό κάποιας άλλης τιμής του ίδιου τύπου.

Αν έχουμε μια μεταβλητή *s1* τύπου *string*, μπορούμε να πούμε ότι ένας υποορμαθός της ορίζεται από δύο φυσικούς αριθμούς:

- τον *δείκτη της θέσης* που αρχίζει και
- το *μήκος* που έχει.

Αυτές ακριβώς τις παραμέτρους περιμένει και η *substr()*. Αν:

```
string s1 = "ένας ορμαθός της C++", s2;

s2 = s1.substr( 5, 7 );
cout << s2 << endl;
```

θα πάρουμε:

```
ορμαθός
```

Μπορούμε να βάλουμε μόνο μία παράμετρο, τη θέση, οπότε το τέλος του υποορμαθού είναι το τέλος του αρχικού:

```
s2 = s1.substr( 5 );
cout << s2 << endl;
```

Αποτέλεσμα:

```
ορμαθός της C++
```

Παράδειγμα 1

Στο παράδειγμα της προηγούμενης παραγράφου, η αντιγραφή από τη *strIn* στη *strOut* μπορεί να γίνει ως εξής:

```
     strOut = "";    // κενό
     for ( k = strIn.length()-1; k >= 0; k = k-1 )
        strOut.append( strIn.substr(k,1) );
```

Παράδειγμα 2 ✍

Έχοντας δηλώσει:

```
string s1, s2, s3;
```

μετά την εκτέλεση των παρακάτω εντολών:

```
s1 = "ΦΑΣΟΥΛΙ ΦΑΣΟΥΛΙ ΓΕΜΙΖΕΙ ΤΟ ΣΑΚΟΥΛΙ";
s2 = s1.substr( 8, 8 );
s3 = s1.substr( s1.find('Τ'), 10 );
cout << s2 << s3 << endl;
s2 = s1.substr( 20, 20 );
cout << s2 << endl;
```

θα τυπωθούν τα ακόλουθα:

```
ΦΑΣΟΥΛΙ ΤΟ ΣΑΚΟΥΛΙ
ΖΕΙ ΤΟ ΣΑΚΟΥΛΙ
```

Από τη δεύτερη γραμμή βλέπεις ότι εσύ μπορεί να παραβείς τους περιορισμούς μήκους, αλλά η `substr` δεν θα υπερβεί τα όρια της αρχικής τιμής.
✍✍✍

10.12 Ρεύματα Από και Προς *string*

Μέσα στις πολλές δυνατότητες διαχείρισης εισόδου/εξόδου στοιχείων η C++ δίνει και τύπους **ρευμάτων από και προς *string*** (string streams). Έτσι, έχουμε τη δυνατότητα:

- Να παίρνουμε τα στοιχεία εισόδου –όπως και αν είναι αυτά– να κάνουμε ελέγχους εγκυρότητας και να γνωστοποιήσουμε στον χρήστη τυχόν προβλήματα ώστε να τα διορθώσει.
- Να ετοιμάζουμε (μορφοποιούμε) τα στοιχεία εξόδου όπως θέλουμε πριν τα βγάλουμε στην οθόνη ή σε κάποιο αρχείο text.

Αν βάλεις στο πρόγραμμά σου την οδηγία "`#include <sstream>`" σου δίνεται η δυνατότητα να δηλώσεις:

```
istringstream ssin( aString );
ostringstream ssout;
```

Το *ssin* είναι ένα ρεύμα σαν αυτά που ξέρουμε μόνο που δεν διαβάζει ούτε το πληκτρολόγιο ούτε κάποιο αρχείο αλλά το

```
string aString( "17 -375 145.78 31e4" );
```

Να ένα πρόγραμμα από τα παλιά (§2.3) κάπως αλλαγμένο:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main()
{
   int    k, j;
   double x, y, t;
   string aString( "17 -375 145.78 31e4" ), outString;
   istringstream ssin( aString );
   ostringstream ssout;

   ssin >> k >> j >> x >> y;
   ssout << k << ' ' << j << ' ' << x << ' ' << y;
   outString = ssout.str();
   cout << outString << endl;
}
```

Με την:

```
ssin >> k >> j >> x >> y;
```

διαβάζονται οι 17, -375, 145.78, 31*e*4 ως τιμές των *k*, *j*, *x*, *y* αντιστοίχως. Αν μετά από αυτήν προσπαθήσεις να διαβάσεις (`ssin >> t`) δεν θα τα καταφέρεις. Αν ελέγξεις την *ssin*`.eof()` θα τη βρεις `true`.

Με την

```
ssout << k << ' ' << j << ' ' << x << ' ' << y;
```

οι τιμές των μεταβλητών γράφονται σε έναν ενταμιευτή τύπου *string* που έχει το *ssout*. Αυτόν τον ενταμιευτή τον παίρνουμε με την `ssout.str()` και τον αντιγράφουμε σε μια άλλη μεταβλητή τύπου *string*. Φυσικά θα μπορούσαμε να γράψουμε κατ' ευθείαν:

```
cout << ssout.str() << endl;
```

Τα ρεύματα προς/από *string* έχουν τις ίδιες ιδιότητες (και δυνατότητες) με τα ρεύματα προς/από αρχεία. Επιπλέον όμως έχουν τη μέθοδο *str* για τον χειρισμό του ενταμιευτή:

- Με τις `ssin.str(aString)` και `ssout.str(aString)` βάζουμε ως ενταμιευτή την (τιμή που έχει εκείνη τη στιγμή η) *aString*. Για το *ssout* αυτό έχει μικρή σημασία αφού, με το πρώτο γράψιμο, ο ενταμιευτής θα «καθαριστεί».
- Με τις `as = ssin.str()` και `as = ssout.str()` αντιγράφουμε στην *as* (τύπου *string*) την τιμή του ενταμιευτή (εκείνη τη στιγμή). Στην περίπτωση αυτήν η πρώτη έχει μικρή χρησιμότητα αφού την έχουμε βάλει εμείς.

  Στη συνέχεια θα ασχοληθούμε με τον χειρισμό «αριθμητικών» ορμαθών.

10.12.1 Ορμαθοί και Αριθμοί

Οι πεπειραμένοι προγραμματιστές δεν γράφουν εντολές που να διαβάζουν αριθμητικά δεδομένα παρά μόνον στην περίπτωση που διαβάζουν αρχείο text που έχει ήδη ελεγχθεί. Συνήθως διαβάζουν ορμαθούς χαρακτήρων και στη συνέχεια τους μετατρέπουν σε αριθμητικές τιμές ελέγχοντας τυχόν λάθη. Ποιο είναι το κέρδος;

- Διαβάζοντας αριθμητικές τιμές μπορεί να συναντήσεις λάθη, π.χ. "`,`" αντί για "`.`" στην υποδιαστολή, ελληνικό έψιλον αντί για "`e`" στον εκθέτη και άλλα παρόμοια. Όπως ήδη ξέρεις, αυτά τα προβλήματα «διαταράζουν» (μπορεί και να διακόψουν) την ομαλή εκτέλεση του προγράμματός σου.
- 'Οταν διαβάζεις χαρακτήρες τα πάντα είναι δεκτά! Ο έλεγχος εγκυρότητας γίνεται από το πρόγραμμα και οι χειρισμοί λαθών είναι ευθύνη του προγραμματιστή.

Τα ρεύματα από τιμές *string* είναι ένα καλό εργαλείο για τη δουλειά αυτή.

Ας πούμε ότι διαβάζεις έναν ορμαθό χαρακτήρων στη μεταβλητή

```
   string aString;
// . . .
   getline( cin, aString, '\n' );
```

Έχεις ζητήσει από τον χρήστη να πληκτρολογήσει έναν ακέραιο –που θέλεις να αποθηκεύσεις στη μεταβλητή (`int`) *k*– αλλά αντί για "`7`" σου πληκτρολόγησε "`&`". Αυτή είναι και η τιμή της *aString*. Για να περάσουμε από την *aString* στην *k* θα χρειαστούμε ένα ρεύμα:

```
istringstream ssin;
```

στο οποίο θα βάλουμε ως ενταμιευτή την *aString*:

```
ssin.str( aString );
```

και από αυτό θα αποπειραθούμε να διαβάσουμε:

```
ssin >> k;
if ( ssin.fail() )  // αποτυχία
```

Αμέσως μετά ελέγχουμε, με τη γνωστή μας *fail*, αν τα καταφέρουμε.

Στην περίπτωσή μας θα μας πει ότι αποτύχαμε και θα πρέπει να δώσουμε το κατάλληλο μήνυμα προς τον χρήστη.

Ας πούμε τώρα ότι ο χρήστης πληκτρολόγησε "**1&**" και αυτή είναι η τιμή της *aString*. Στην περίπτωση αυτήν η **ssin.fail()** θα δώσει **false**, δηλαδή: όλα πήγαν καλά! Αλλά η *k* θα πάρει τιμή 1. Εδώ πώς πιάνουμε το λάθος; Προσπαθούμε να διαβάσουμε 1 χαρακτήρα. Αν τα καταφέρουμε θα πει ότι η τιμή της *k* δεν έχει προκύψει από ολόκληρη την τιμή της *aString*. Άρα έχουμε πρόβλημα. Αν δεν τα καταφέρουμε και πάρουμε **ssin.eof()** θα πει ότι όλα πήγαν καλά.

Ας τα δώσουμε όλα μαζί: Πρώτα οι δηλώσεις

```
   istringstream ssin;
   int    k;
   string aString;
   char   c;
```

Και μετά η ανάγνωση και οι έλεγχοι:

```
   getline( cin, aString, '\n' );
   ssin.str( aString );
   ssin >> k;
   if ( ssin.fail() )
   // λάθος
   else // κάτι διάβασε
   {
      ssin >> c;
      if ( ssin.eof() ) // OK, δεν έχει άλλα, τα διάβασε όλα
         cout << k << endl;
      else
      // λάθος
   }
```

Με παρόμοιο τρόπο διαβάζουμε και πραγματικές τιμές.

Στη συνέχεια δίνουμε ένα παράδειγμα για να δούμε τη χρήση αυτών που είπαμε στα προγράμματά μας.

Παράδειγμα

Ας πάρουμε το πρόγραμμα της ελεύθερης πτώσης, που είδαμε στην §2.12, και ας κάνουμε την ανάγνωση του ύψους μέσω ορμαθού χαρακτήρων:

```
#include <iostream>
#include <cmath>
#include <string>
#include <sstream>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec2, η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
   string s;
   istringstream ssin;
   bool done( false );
   char c;

// Διάβασε το h
   while ( !done || h < 0 )
   {
      cout << " Δώσε μου το αρχικό ύψος (h >= 0) σε m: ";
      getline( cin, s, '\n' );
      ssin.clear();
      ssin.str( s );   ssin >> h;
      if ( ssin.fail() )
         cout << "απαράδεκτοι χαρακτήρες (" << s << ")" << endl;
      else
      {
         ssin >> c;
         if ( ssin.eof() )
```

```
            done = true;
         else
            cout << "απαράδεκτος χαρακτήρας (\'" << c << "\')"
                 << endl;
      } // if ( !ssin.fail
   } // while

// (g == 9.81) && (0 <= h <= DBL_MAX)
// Υπολόγισε τα tP, vP
// ΤΑ ΥΠΟΛΟΙΠΑ ΙΔΙΑ
```

Όπως βλέπεις, αντί για την `cin >> h` έχουμε βάλει:

```
getline( cin, s, '\n' );
ssin.clear();
ssin.str( s );   ssin >> h;
```

και μετά έρχονται οι έλεγχοι. Πρόσεξε την "`ssin.clear()`": αν, μετά το λάθος που βρίσκουμε όταν προσπαθήσει να διαβάσει το "`*)`", δεν δώσουμε `ssin.clear()`, στην επόμενη προσπάθεια ανάγνωσης, θα βγάλει το ίδιο λάθος ακόμη και αν τα κάνουμε όλα σωστά. Όλα αυτά τα βάλαμε σε μια **while** από την οποίαν βγαίνει μόνον αν πάρει σωστή τιμή (και μη αρνητική). Αυτό είναι κάπως ανελαστικό. Καλύτερα θα ήταν αν δίναμε και κάποια διαφυγή αν ο χρήστης μετανιώσει και θέλει να τα παρατήσει (π.χ. να πιέσει το <esc>).

Στη συνέχεια βλέπεις ένα παράδειγμα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος (h >= 0) σε m: *)
απαράδεκτοι χαρακτήρες (*))
 Δώσε μου το αρχικό ύψος (h >= 0) σε m: 8o
απαράδεκτος χαρακτήρας ('o')
 Δώσε μου το αρχικό ύψος (h >= 0) σε m:  -80
 Δώσε μου το αρχικό ύψος (h >= 0) σε m:  80
 Αρχικό ύψος = 80 m
 Χρόνος Πτώσης = 4.03855 sec
 Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
```

Το αντίστροφο, η μετατροπή μιας αριθμητικής τιμής σε *string*, είναι πολύ απλή. Στο παρακάτω πρόγραμμα δίνουμε μερικά παραδείγματα από το Κεφ. 1 κάπως αλλαγμένα:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
   ostringstream sout;
   string aString;

   sout.precision( 8 );
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;

   sout.setf( ios_base::scientific, ios_base::floatfield );
   sout.precision(8);
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;

   sout.setf( ios_base::fixed, ios_base::floatfield );
   sout.precision(8);
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;
   aString = sout.str();
   cout << aString << endl;
}
```

Εδώ, αντί για το *cout*, γράφουμε στο ρεύμα *sout*. Τελικώς, αντιγράφουμε τον ενταμιευτή του *sout* στη μεταβλητή *aString*. Να η τιμή της:

```
1234.5  123456.78  1.2345679e+008
1.23450000e+003  1.23456780e+005  1.23456789e+008
1234.50000000  123456.78000000  123456789.01230000
```

Από εδώ και πέρα μπορείς να χειριστείς την τιμή αυτή με όλα τα εργαλεία που έχεις για τον τύπο *string*.

10.13 Η Κληρονομιά της C: Πίνακες με Χαρακτήρες

Θα δούμε εν συντομία τη διαχείριση κειμένου από τη C διότι:

1. θα τη δεις σε πολλά προγράμματα C++ (οι προγραμματιστές δύσκολα εγκαταλείπουν αυτά που ξέρουν) και
2. είναι χρήσιμη ακόμη και όταν δουλεύεις με τον τύπο *string*.

Όπως είδαμε στην εισαγωγή του κεφαλαίου:

- αποθηκεύουμε κείμενα σε πίνακες με στοιχεία τύπου `char`,
- στο τέλος του κειμένου βάζουμε *απαραιτήτως* έναν `char(0)`· αυτό πρέπει να το παίρνουμε υπόψη μας όταν δηλώνουμε τον πίνακα.

Αν δηλώσεις:

```
char a[100] = "πάν μέτρον εκατό πόντοι";
```

ή

```
char a[] = "πάν μέτρον εκατό πόντοι";
```

ο `char(0)` εισάγεται αυτομάτως. Αν όμως δηλώσεις:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ',
             'π','ό','ν','τ','ο','ι','\0' };
```

θα πρέπει να βάλεις το `char(0)` όπως βλέπεις εδώ.[8]

Είδαμε ακόμη ότι μπορείς να γράφεις στην οθόνη όπως ξέρουμε:

```
cout << a << endl;
```

και ότι μπορούμε να διαβάσουμε από το πληκτρολόγιο με τη *cin.getline()*. Αν

```
char a[100];
```

μπορείς να διαβάσεις κείμενο από το πληκτρολόγιο, ως τιμή του a, με την:

```
cin.getline( a, 100 );
```

Πάντως, η κλάση *istream* του *cin*, έχει και μια μορφή της *get()* με την οποία μπορείς να διαβάσεις κείμενο:

```
cin.get( a, 100 );
```

Και εδώ, το δεύτερο όρισμα, δείχνει το μέγιστο πλήθος χαρακτήρων που είναι δεκτοί. Και εδώ, στο μήκος περιλαμβάνεται και μια θέση για το `char(0)`. Αλλά, όταν δουλεύεις με τη *get* μην ξεχνάς να διαβάζεις και το «τέλος γραμμής» (<enter>, `'\n'`):

```
cin.get( a, 100 );  cin.get( ceol );
```

όπου η *ceol* είναι μεταβλητή τύπου `char`.

Στη συνέχεια θα δούμε μερικές από τις συναρτήσεις που έχει η C++ (από τη C) για να διαχειρίζεται πίνακες με κείμενα. Για να τις χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου:[9]

```
#include <string>
```

Μπορούμε να αλλάξουμε την τιμή της a με εντολή εκχώρησης σαν την:

---

[8] Το `'\0'` είναι που γράφεται συνήθως. Αν βάλεις `char(0)` ή απλώς `0` και πάλι είναι μια χαρά.

[9] Αν βάλεις "`#include <cstring>`" έχεις μόνον τις συναρτήσεις της C. Αν δεν βάλεις το "`c`" τα έχεις όλα.

```
a = "όποιος φυλάει τα ρούχα του είναι demodé";
```

Όχι! Για να αλλάξεις την τιμή της a χρειάζεται μια συνάρτηση της C++, η *strcpy()* (**str**ing **cpy**):

```
strcpy( a, "όποιος φυλάει τα ρούχα του είναι demodé" );
```

ή ακόμη:

```
strcpy( a, b );
```

αν ο b είναι ένας παρόμοιος πίνακας, π.χ.:

```
char b[] = "πάν μέτρον εκατό πόντοι";
```

Η *strncpy()* είναι σαν την *strcpy()*, αλλά έχει και τρίτη παράμετρο όπου περιμένει έναν φυσικό αριθμό που καθορίζει πόσοι χαρακτήρες θα αντιγραφούν από το δεύτερο όρισμα στο πρώτο. Θέλει λίγη προσοχή:

```
char a[50] = "abcdefghijklmnopq";

strncpy( a, "123456789", 15 );
cout << a << endl;
strcpy( a, "abcdefghijklmnopq" );
strncpy( a, "123456789", 5 );
cout << a << endl;
```

Αποτέλεσμα:

```
123456789
12345fghijklmnopq
```

Την πρώτη φορά ζητήσαμε να αντιγραφούν 15 χαρακτήρες ενώ το μήκος του δεύτερου ορίσματος είναι 9. Αντιγράφηκαν λοιπόν όλοι οι χαρακτήρες και το `char(0)` που σημειώνει το τέλος του ορμαθού· έτσι τιμή του a γίνεται το `123456789`. Τη δεύτερη φορά ζητήσαμε να αντιγραφούν 5 χαρακτήρες. Αυτοί αντικατάστησαν τους 5 πρώτους χαρακτήρες της τιμής του a. Αν θέλεις τιμή του a να γίνει το `12345` θα πρέπει να δώσεις:

```
strncpy ( a, "123456789", 5 );  a[5] = char( 0 );
```

Πολύ συχνά έχεις την εξής περίπτωση:

```
string cpps;
char   cs[ L ];
```

όπου *L* θετικός ακέραιος και θέλεις να αντιγράψεις στον cs την τιμή της *cpps* ή, εν πάσει περιπτώσει, «όσο χωράει». Αυτό γίνεται ως εξής:

```
strncpy ( cs, cpps.c_str(), L-1 );  cs[L-1] = char( 0 );
```

Δηλαδή: αντιγράφουμε *L* - 1 χαρακτήρες το πολύ και στην τελευταία θέση βάζουμε τον φρουρό ώστε ο cs να είναι διαχειρίσιμος. Αν το μήκος της τιμής της *cpps* είναι μικρότερο από *L* - 1 θα υπάρχει και άλλο `char(0)` πιο πριν και θα είναι ο πραγματικός φρουρός.

Μπορείς να συγκρίνεις δύο πίνακες με κείμενα, με τη *strcmp()*, που είναι σαν την `compare`. Η `strcmp(a, b)` επιστρέφει ακέραιη

- αρνητική τιμή *αν* η τιμή του a προηγείται λεξικογραφικώς της τιμής του b,
- *θετική* τιμή *αν η τιμή του* a *έπεται λεξικογραφικώς της τιμής του* b,
- *μηδέν* (0) *αν οι τιμές των* a *και* b *είναι ίσες*.

Η *strncmp* είναι σαν την *strcmp* αλλά παίρνει και τρίτο όρισμα, έναν φυσικό αριθμό, που μας λέει πόσους χαρακτήρες να συγκρίνει. Οι:

```
char a[50] = "abcdefghijklmnopq", b[] = "abcdrstuvwxyz";
cout << strcmp(a, b) << "  " << strncmp(a, b, 3) << endl;
```

δίνουν:

```
-13  0
```

Δηλαδή: η τιμή του a προηγείται λεξικογραφικώς της τιμής του b (strcmp(a, b) `==` -13 < 0), αλλά οι τρεις πρώτοι χαρακτήρες τους είναι ίδιοι (*strncmp*`(`a, b, 3`) ==` 0).

Πολύτιμη είναι και η *stricmp()* είναι σαν την *strcmp()* αλλά βλέπει κεφαλαία και πεζά (λατινικά) γράμματα ίσα. Οι:

```
char a[] = "firstName", b[] = "FirstName", c[] = "firstname";
cout << strcmp(a, b) << "  " << strcmp(a, c) << endl;
cout << stricmp(a, b) << "  " << stricmp(a, c) << endl;
```

δίνουν:

```
1  -1
0  0
```

Η *strnicmp()* είναι για την *stricmp()* ό,τι είναι η *strncmp* για την *strcmp*.

Η *strlen()* (**str**ing **len**gth), επιστρέφει τιμή τύπου `size_t` που είναι το μήκος του ορμαθού που έχουμε αποθηκεύσει, χωρίς το τελικό `char(0)`: οι

```
char a[100], b[] = "πάν μέτρον εκατό πόντοι";

strcpy( a, "όποιος φυλάει τα ρούχα του είναι demodé" );
cout << strlen(a) << "  " << strlen(b) << endl;
cout << strlen( "μέτρον άριστον" ) << endl;
```

δίνουν:

```
39  23
14
```

Ο ορμαθός a είναι κενός αν:

- είναι ίσος με `""` (`strcmp(a, "") == 0`) ή
- έχει μήκος μηδέν (`strlen(a) == 0`) ή
- έχει `char(0)` στην αρχή του (`a[0] == char(0)`).

Μπορείς να χρησιμοποιείς οποιαδήποτε από αυτές τις συνθήκες στα προγράμματά σου.

Η `strcat(a, b)` κάνει το ίδιο πράγμα με την `a.append(b)`: επισυνάπτει στο τέλος της τιμής του a την τιμή του b. Δες το παρακάτω:

```
char s1[] = "πέντε", s2[] = "στο χέρι παρά δέκα", s3[50];

strcpy( s3, "κάλλιο " );
strcat( s3, s1 );  strcat( s3, " και " );  strcat( s3, s2 );
strcat( s3, " και καρτέρει" );
cout << s3 << endl;
```

Αποτέλεσμα:

```
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

Υπάρχει και η *strncat()*, που παίρνει και τρίτο όρισμα, έναν φυσικό αριθμό, που λέει πόσους χαρακτήρες από το δεύτερο όρισμα θα επισυναφθούν στο τέλος του πρώτου.

Αυτά τα λίγα δεν εξαντλούν τα εργαλεία της C++ για τα "C-style strings". Μερικά ακόμη υπάρχουν στην επόμενη παράγραφο. Θα πρέπει όμως να τονίσουμε ότι τα δίνουμε πιο πολύ για να μη σου είναι άγνωστα αν τα δεις σε κάποιο πρόγραμμα. Στα προγράμματα που θα γράφεις εσύ θα πρέπει να δουλεύεις με την κλάση *string*, που είναι σαφώς πιο εύχρηστη.

10.13.1 Ορμαθοί C και Αριθμοί

Και η C σου δίνει δυνατότητα να γράφεις σε (και να διαβάζεις από) πίνακες χαρακτήρων. Ειδικώς για μετατροπές ορμαθών σε αριθμούς και αντιστρόφως σου δίνει ορισμένες συναρτήσεις που θα δούμε εδώ αλλά κι αργότερα. Οι δηλώσεις για τα εργαλεία αυτά υπάρχουν στο `cstdlib` και θα πρέπει να το περιλάβεις στο πρόγραμμά σου, όταν τα χρησιμοποιείς.

Το πιο απλά εργαλεία είναι οι συναρτήσεις *atof()* (**a**lphanumeric **to** **f**loat), *atoi()* (**a**lphanumeric **to** **i**nt) και *atoll()* (**a**lphanumeric **to** **l**ong). Αυτές παίρνουν έναν ορμαθό χαρακτήρων (ή έναν πίνακα με τέτοια τιμή) και μας δίνουν μια τιμή τύπου `double`, η πρώτη, τύπου `int` η δεύτερη και τύπου `long` η τρίτη. Π.χ. οι εντολές:

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main()
{
   char s1[] = "12345", s2[] = "1.23456789e10",
        s3[] = "1.23ab45678";
   long l;
   double d;

   l = atol( s1 );   d = atof( s2 );
   cout << l << "  " << d << endl;
```

θα δώσουν:

```
12345  1.23457e+10
```

Η μετάφραση από χαρακτήρες σε αριθμό σταματάει μόλις βρεθεί παράνομος χαρακτήρας. Π.χ. οι:

```
   l = atol( s3 );   d = atof( s3 );
   cout << l << "  " << d << endl;
```

θα δώσουν:

```
1  1.23
```

Τι έγινε εδώ; Η *atoll()*, προσπαθώντας να διαβάσει έναν ακέραιο από τον ορμαθό `1.23ab45678`, και αφού δεχτεί το `1`, βρίσκει την `'.'`, που δεν είναι δεκτή σε μια σταθερά τύπου `long`. Έτσι, μας επιστρέφει τιμή `1`. Για την *atof()* είναι δεκτοί οι χαρακτήρες μέχρι `1.23`· πρώτος παράνομος χαρακτήρας είναι ο `'a'`. Έτσι, μας επιστρέφει τιμή `1.23`.

Από αυτές τις συναρτήσεις δεν μπορείς να πάρεις παραπάνω πληροφορίες για το τι δεν πήγε καλά.

Πιο πλήρη δουλειά, σε περίπτωση λάθους, κάνουν οι *strtod()* και *strtol()* που θα δούμε αργότερα.

10.14 * Ο Τύπος *std*::*wstring*

Στην §4.7 είδαμε τον τύπο `wchar_t` που κάθε του τιμή αποθηκεύεται σε 16 δυαδικά ψηφία. Έτσι, όπως λέγαμε, «*έχει 65536 διαφορετικές τιμές, αντί για τις 256 του* `char` *και χρησιμοποιείται για την υλοποίηση του προτύπου Unicode.*» Και ακόμη «*Μια σταθερά τύπου* `wchar_t` *είναι μια σταθερά τύπου* `char` *με το πρόθεμα "*`L`*", π.χ.:*

`L'a'  L'%'  L'ξ'`»

Σε έναν πίνακα με στοιχεία τύπου `wchar_t` μπορούμε να βάζουμε ως τιμή έναν ορμαθό τιμών τύπου `wchar_t` είτε με τη δήλωση:

```
   wchar_t w[5]= L"αβγδ";
```

είτε αργότερα:

```
   wcscpy( w, L"wsx" );
```

Εδώ βλέπουμε τα εξής:

- Όπως μια σταθερά τύπου `char` με το πρόθεμα "`L`" γίνεται σταθερά τύπου `wchar_t`, έτσι και ένας ορμαθός τιμών τύπου `char` με το πρόθεμα "`L`" γίνεται ορμαθός τιμών τύπου `wchar_t`.
- Όπως η αντιγραφή ορμαθών τύπου `char` γίνεται με τη *strcpy*, η αντιγραφή ορμαθών τύπου `wchar_t` γίνεται με τη *wcscpy*.

Γενικώς, για κάθε συνάρτηση *strX* υπάρχει αντίστοιχη συνάρτηση *wcsX* για τη διαχείριση ορμαθών τύπου `wchar_t`.

Η C++ μας δίνει τον τύπο *std*::*wstring*, ίδιο σχεδόν με τον *std*::*string*, στις μεταβλητές του οποίου μπορούμε να αποθηκεύουμε ορμαθούς `wchar_t`. Έτσι, μπορούμε να ορίσουμε:

```
   wchar_t w[5]= L"αβγδ";
   wstring w0, w1( L"abc" ), w2( 4, L'q' ), w3( w );
```

Προσπαθώντας να δούμε τις τιμές των μεταβλητών *wstring* συναντούμε ένα πρόβλημα: Μια από τις διαφορές του *wstring* από τον *string* είναι ότι δεν μπορούμε να βγάλουμε την τιμή μεταβλητής *wstring* σε κάποιο ρεύμα με τον "<<". Δίνουμε μια πρόχειρη λύση –ας πούμε για τη *w1*– ως εξής:

```
int ndx;
for ( ndx = 0; ndx < w1.length(); ndx = ndx+1 )
   cout << static_cast<unsigned char>(w1[ndx]);
cout << endl;
```

Με αυτόν τον τρόπο μπορούμε να δούμε ότι το *w0* είναι κενό ενώ για τα *w1*, *w2* και *w3* παίρνουμε αντιστοίχως:

```
abc
qqqq
αβγδ
```

Μπορείς να αλλάξεις την τιμή μιας τέτοιας μεταβλητής με εκχώρηση, π.χ.:

```
w0 = L"qazw";
```

Με τη *c_str* μπορείς να βγάλεις το περιεχόμενο σε πίνακα και με τη *wcscpy* να το αντιγράψεις σε έναν πίνακα με στοιχεία `wchar_t`:

```
wcscpy( w, w1.c_str() );
```

Τέλος, για να μη σου μείνουν αμφιβολίες, δίνοντας:

```
cout << typeid(w1[0]).name() << "  " << sizeof(w1[0]) << endl;
```

παίρνεις αποτέλεσμα:[10]

```
wchar_t  2
```

10.15 Εν Κατακλείδι ...

Η C++ μας δίνει πολλά και αξιόλογα εργαλεία για να διαχειριστούμε κείμενο. Όλα περιλαμβάνονται (ή σχετίζονται) με την κλάση *string*. Μάθαμε λοιπόν:

- Να δηλώνουμε μεταβλητές τύπου *string*.
- Να αλλάζουμε την τιμή τους.
- Να διαβάζουμε από και να γράφουμε σε ρεύματα αρχείων (ή τα *cin*, *cout*) τιμές μεταβλητών *string*.
- Να συγκρίνουμε τιμές τύπου *string*.
- Να αναζητούμε κείμενο μέσα σε άλλο κείμενο.
- Να μετατρέπουμε τιμές *string* σε αριθμητικές και αντιστρόφως.

Είδαμε εν συντομία και τα αντίστοιχα εργαλεία της C· θα τα δεις να χρησιμοποιούνται σε πολλές περιπτώσεις.

Ασκήσεις

Α Ομάδα

10-1 Γράψε πρόγραμμα που θα διαβάζει έναν ορμαθό χαρακτήρων και θα αποφαίνεται αν είναι ή δεν είναι της μορφής *qq*. Π.χ. οι "αβγαβγ", "zppquzppqu" είναι της μορφής *qq*, ενώ οι "αβγδε", "zppqqppz" δεν είναι.

10-2 Ξαναγράψε το πρόγραμμα του παραδείγματος της §10.5, έτσι ώστε να διαβάζει τα ονόματα των γλωσσών από αρχείο text.

[10] Borland C++ 5.5.

10-3 Ξαναγράψε το πρόγραμμα του παράδ. 1 της §8.10 έτσι ώστε να διαβάζει από το αρχείο ολόκληρες γραμμές και όχι χαρακτήρα προς χαρακτήρα.

10-4 Κάνε το ίδιο για το πρόγραμμα του παραδ. 2 της §8.10.

Β Ομάδα

10-5 Γράψε μια:

```
string getRight( string s, int n )
```

που θα επιστρέφει ως τιμή τους *n* τελευταίους (δεξιότερους) του *s*. Αν $n \le 0$ η τιμή θα είναι κενή, ενώ αν $n \ge s.length()$ η τιμή θα είναι αυτή της *s*.

10-6 Ο αριθμός μιας πιστωτικής κάρτας δημιουργείται ως εξής:

- Τα πρώτα ψηφία χαρακτηρίζουν την κάρτα. Στον Πίν. 10-1 βλέπεις τα χαρακτηριστικά για ορισμένες πολύ γνωστές πιστωτικές κάρτες.
- Στη συνέχεια υπάρχουν τα ψηφία που δείχνουν την τράπεζα.
- Μετά υπάρχει ο αριθμός του πελάτη.
- Το τελευταίο ψηφίο είναι **ψηφίο ελέγχου** (check digit) επιλεγμένο έτσι ώστε να ισχύει η συνθήκη εγκυρότητας LUHN.

Τύπος Κάρτας	**Προθεμα**	**Μήκος**	**Αλγόριθμος Ψηφίου Ελέγχου**
MASTERCARD	51-55	16	mod 10
VISA	4	13, 16	mod 10
AMEX	34, 37	15	mod 10
Diners Club/ Carte Blanche	300-305 36, 38	14	mod 10
Discover	6011	16	mod 10
enRoute	2014 2149	15	any
JCB	3	16	mod 10
JCB	2131 1800	15	mod 10

Πίν. 10-1 Στη στήλη **Πρόθεμα** βλέπεις το χαρακτηριστικό κάθε κάρτας και στη στήλη **Μήκος** το μήκος του αριθμού που χρησιμοποιεί.

Η συνθήκη εγκυρότητας LUHN υπολογίζεται ως εξής:

Βήμα 1: Πολλαπλασίασε επί δύο τα ψηφία που έχουν άρτιες θέσεις (θέση 1 η τελευταία, 2 η προτελευταία κ.ο.κ.)

Βήμα 2: Πρόσθεσε όλα τα ψηφία των αριθμών που προέκυψαν από τους διπλασιασμούς και αυτά που βρίσκονται σε μονές θέσεις.

Αν το άθροισμα είναι πολλαπλάσιο του 10 ο αριθμός είναι έγκυρος.

Παράδειγμα

Έστω ότι έχουμε τον αριθμό:

49927398716

Βήμα 1:

```
        4 9 9 2 7 3 9 8 7 1 6
         x2  x2  x2  x2  x2
------------------------------
         18   4   6  16   2
```

Βήμα 2: 4 +(1+8)+ 9 + (4) + 7 + (6) + 9 +(1+6) + 7 + (2) + 6 = 70

Άρα ο αριθμός είναι έγκυρος.

α) Γράψε συνάρτηση *isValidCrCardNum* που θα παίρνει (ως όρισμα) μια τιμή τύπου *string* που θα είναι αριθμός πιστωτικής κάρτας, θα εξετάζει το ψηφίο ελέγχου και θα επιστρέφει ως τιμή `true` αν ο αριθμός είναι έγκυρος (σωστό ψηφίο ελέγχου) και `false` αν δεν είναι.

β) Γράψε συνάρτηση *appendChkDigit* που θα παίρνει (ως όρισμα) μια τιμή τύπου *string* που θα είναι αριθμός πιστωτικής κάρτας χωρίς το τελευταίο ψηφίο. Η συνάρτηση θα συμπληρώνει το τελευταίο ψηφίο με βάση τα όσα είπαμε παραπάνω.

Γράψε πρόγραμμα που θα σου επιτρέπει να δοκιμάσεις τις συναρτήσεις που έγραψες.

10-7 Γράψε πρόγραμμα που θα διαβάζει ένα κείμενο από ένα αρχείο text και θα το αποθηκεύει σε μια μεταβλητή τύπου `string`. Κατά την ανάγνωση θα αντικαθιστά κάθε αλλαγή γραμμής με ένα διάστημα (κενό). Στη συνέχεια θα διαβάζει λέξεις από το πληκτρολόγιο και θα μας λέει πόσες φορές υπάρχει κάθε μια από αυτές στο κείμενο. Θα τελειώνει όταν πληκτρολογήσουμε τη λέξη `zzz`.

Γ Ομάδα

10-8 (*Πιο ρεαλιστική παραλλαγή της άσκ.* 8-7) Ένα αρχείο text, με όνομα στο δίσκο misthoi.txt, είναι γραμμένο ως εξής: Σε κάθε γραμμή έχει ένα επώνυμο εργαζόμενου και τρεις αριθμούς ως εξής:

```
ΣΙΩΠΑΚΗΣ\t1234.5\t7\t2\n
```

(το '`\t`' παριστάνει τον χαρακτήρα tab και το '`\n`' την αλλαγή γραμμής.) Ο πρώτος αριθμός (πραγματικός) –εδώ 1234.5– είναι ο βασικός μισθός ενός υπαλλήλου, ο δεύτερος –εδώ 7– ο αριθμός ετών υπηρεσίας του ίδιου υπαλλήλου και ο τρίτος –εδώ 2– ο αριθμός των παιδιών του. Το πλήθος των στοιχείων είναι άγνωστο.

Το περιεχόμενο του αρχείου δεν έχει ελεγχθεί. Έτσι, μπορεί να έχει:

- Κενές γραμμές (μικρό το κακό).
- Γραμμές από τις οποίες λείπουν στοιχεία.
- Λαθεμένα στοιχεία που παραβιάζουν κάποια από τα παρακάτω:
  - Το επώνυμο του εργαζόμενου δεν πρέπει να είναι κενό.
  - 800 ≤ βασικός μισθός ≤ 3500,
  - 0 ≤ χρόνια υπηρεσίας ≤ 35,
  - 0 ≤ αριθμός παιδιών ≤ 12.

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα δημιουργεί ένα νέο –το log.txt– με καταγραφή μιας γραμμής για κάθε λάθος που θα βρίσκει. Π.χ.:

```
Line 7 salary: 35650
Line 7 number of children: -4
Line 11 number of years: 51
Line 18 empty
Line 26 salary: 955000
Line 28 incomplete line
Line 36 name empty
```

Υπόδ.: Διάβαζε ολόκληρες γραμμές (μέχρι να βρεις '`\n`') με τη *getline*. Σε κάθε γραμμή θα πρέπει να υπάρχει το "`\t`" ακριβώς 3 φορές.

Μέρος Β
Τεχνικές Προγραμματισμού

κεφάλαιο

11

Πράξεις και Εντολές

Ο στόχος μας σε αυτό το κεφάλαιο:

- Να κάνουμε μια επανάληψη των πράξεων και των εντολών που έχουμε μάθει στο *Μέρος Α*.
- Να δούμε ιδιότητές τους που είχαμε αποσιωπίσει στο *Μέρος Α*.
- Να μάθουμε και μερικές εντολές που δεν τις μάθαμε καθόλου.

Προσδοκώμενα αποτελέσματα:

Όταν θα έχεις μελετήσει αυτό το τεύχος θα μπορείς να:

- Χρησιμοποιείς αποδοτικότερα τις δυνατότητες που σου προσφέρει η C++.
- Αποφεύγεις τις
  - «απαγορευμένες» εντολές,
  - «απαγορευμένες» χρήσεις εντολών

Έννοιες κλειδιά:

- *εκχώρηση*
- *παράσταση υπό συνθήκη*
- *εντολή* `for`
- *εντολή* `do-while`
- *επανάληψη n+½*
- *εντολή* `break`
- *εντολή* `switch`
- *ετικέτες - εντολή* `goto`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Στο *Μέρος Α* μάθαμε τις βασικές έννοιες και τεχνικές προγραμματισμού και τις εντολές που μας χρειάζονται για να τις υλοποιήσουμε. Συνοπτικώς μάθαμε τις εξής εντολές:

- Εντολή εισόδου, με την οποία διαβάζουμε τιμές μεταβλητών από το πληκτρολόγιο ή από κάποιο αρχείο.
- Εντολή εξόδου, με την οποία γράφουμε τις τιμές παραστάσεων στην οθόνη ή σε κάποιο αρχείο.
- Εντολή εκχώρησης, με την οποία εκχωρούμε την τιμή κάποιας παράστασης σε μια μεταβλητή.
- Δήλωση, με την οποία παραχωρείται στο πρόγραμμά μας μνήμη για την υλοποίηση μεταβλητής και –αν το ζητήσουμε– ορίζεται η αρχική τιμή της.
- Εντολές επιλογής **if**, **ifelse**, που μας επιτρέπουν να ζητούμε εκτέλεση εντολών υπό συνθήκη.
- Εντολή επανάληψης **while**, που μας επιτρέπει να ζητούμε την επαναλαμβανόμενη εκτέλεση των ίδιων εντολών υπό συνθήκη.
- Εντολή επανάληψης **for**, που τη χρησιμοποιήσαμε για μετρούμενες επαναλήψεις.

Όπως θα μάθουμε στη συνέχεια οι τρεις πρώτες είναι περιπτώσεις μιας εντολής, της *εντολής-παράστασης*.

Είδαμε ακόμη και διάφορες *πράξεις*, μαζί με τους αντίστοιχους *τελεστές*: τις αριθμητικές πράξεις (`+`, `-`, `*`, `/`, `%`), τις συγκρίσεις (`==`, `!=`, `<`, `<=`, `>`, `>=`), τις λογικές πράξεις (`&&`, `||`, `!`) και τις *ενικές* πράξεις:

- τα πρόσημα "`+`" και "`-`",
- "`&`", που μας δίνει τη διεύθυνση της μνήμης όπου βρίσκεται μια μεταβλητή,
- "`*`", που μας δίνει την τιμή της μεταβλητής που βρίσκεται σε κάποια διεύθυνση της μνήμης,
- "`sizeof`", που μας δίνει το "μέγεθος" μιας μεταβλητής σε ψηφιολέξεις και
- "`typeid`", που μας δίνει πληροφορίες για τον τύπο κάποιας παράστασης.

Τώρα, για την υλοποίηση των νέων εννοιών και τεχνικών που θα μάθουμε, θα χρειαστούμε και άλλες εντολές όπως και άλλες δυνατότητες εντολών που ήδη ξέρουμε.

Είναι σίγουρο ότι μπορείς να κάνεις την προγραμματιστική σου δουλειά χωρίς τα περισσότερα από τα «νέα» στοιχεία, αλλά αυτά μπορεί να κάνουν το πρόγραμμά σου πιο σίγουρο και πιο αποδοτικό.

Μερικά από τα νέα στοιχεία είναι επικίνδυνα και πρέπει να αποφεύγεις τη χρήση τους. Αυτά τα τονίζουμε. Άλλα μπορεί να είναι ενοχλητικά χωρίς να είναι επικίνδυνα. Το πώς και πότε θα τα χρησιμοποιείς είναι ζήτημα πείρας και ωριμότητας· θα βρεις την άκρη μόνος/η σου με τον καιρό.

11.1 Εκτελέσιμες Δηλώσεις

Όταν κάποιος μαθαίνει προγραμματισμό, ένα από τα πρώτα πράγματα που του διδάσκουν είναι να δηλώνει τις μεταβλητές του στην αρχή του προγράμματος ή του υποπρογράμματος. Αυτό άλλωστε απαιτούν και οι περισσότερες γλώσσες προγραμματισμού. Σε πολλές γλώσσες όλες οι μεταβλητές μιας συνάρτησης δημιουργούνται με την ενεργοποίηση της συνάρτησης και καταστρέφονται με τον τερματισμό της ενεργοποίησης.

Μέχρι τώρα και εμείς σου δίνουμε την ίδια συμβουλή, αλλά στη C++ τα πράγματα είναι διαφορετικά. Για να καταλάβεις τι θα αλλάξουμε και γιατί θα το αλλάξουμε δες το παρακάτω προγραμματάκι:

```
int main()
{
   MyType v1;

   cout << "going into block" << endl;
   { // <--- block start
      MyType v2( 5 );

      cout << " chkpoint 1" << endl;

      MyType v3( 15 );

      cout << " chkpoint 2" << endl;
   } // <--- block end
   cout << "coming out of block" << endl;
}
```

Ο τύπος *MyType* είναι σαν τον `int` με τη διαφορά ότι κάθε φορά που δημιουργείται ή καταστρέφεται μια μεταβλητή αυτού του τύπου το αναγγέλει[1]. Δες τι μας δίνει η εκτέλεση του προγράμματος:

```
creating a MyType variable. Initializing with 0
going into block
creating a MyType variable. Initializing with 5
 chkpoint 1
creating a MyType variable. Initializing with 15
 chkpoint 2
destroying a MyType variable having value 15
destroying a MyType variable having value 5
coming out of block
destroying a MyType variable having value 0
```

Αυτό που επιβεβαιώνεται εδώ, είναι το εξής:

- ***Μια μεταβλητή δημιουργείται όταν η εκτέλεση φτάσει στη δήλωσή της και καταστρέφεται όταν η εκτέλεση βγει από τη σύνθετη εντολή (block) στην οποίαν περιέχεται η δήλωση.***

Αργότερα θα τα ξαναπούμε πληρέστερα και ακριβέστερα.

Σκέψου τώρα το εξής: σε κάθε πρόγραμμα παραχωρούνται τρεις περιοχές μνήμης,

- η **στατική**, όπου υλοποιούνται οι καθολικές μεταβλητές (και μερικές άλλες που θα δούμε στη συνέχεια),
- η **αυτόματη** (automatic) ή μνήμη **στοίβας** (stack), όπου υλοποιούνται οι τοπικές μεταβλητές των σύνθετων εντολών και των συναρτήσεων και
- η **δυναμική**[2] (dynamic) μνήμη που θα δούμε στη συνέχεια.

Από αυτές η πιο περιορισμένη είναι η μνήμη στοίβας και θα πρέπει να τη χρησιμοποιούμε με φειδώ. Αλλά, αυτή ακριβώς είναι η μνήμη για την οποίαν συζητούμε στο παράδειγμά μας. Με βάση τα παραπάνω μπορούμε να δώσουμε νέον κανόνα για τη δήλωση των μεταβλητών:

- ***Δήλωνε τη μεταβλητή ακριβώς εκεί που τη χρειάζεσαι.***

Μερικοί προχωρούν ακόμη περισσότερο: *μη δηλώνεις μια μεταβλητή αν δεν ξέρεις τι αρχική τιμή να της δώσεις*. Ε, όχι και έτσι...

---

[1] Αργότερα θα μάθουμε πώς να υλοποιήσουμε έναν τέτοιον τύπο.

[2] Θα τη δεις και ως μνήμη **σωρού** (heap).

Αν ακολουθείς αυτόν τον κανόνα συνήθως θα συμμορφώνεσαι και με την (ακριβέστερη) σύσταση του (CERT 2009) που λέει:[3]

♦ ***Χρησιμοποίησε την ελάχιστη δυνατή εμβέλεια για όλες τις μεταβλητές και τις μεθόδους.***

11.2 Περιορισμός Τύπου

Πριν προχωρήσουμε να δούμε άλλες εντολές ας πούμε λίγα πράγματα για τον **περιορισμό τύπου** (type qualification).

Ήδη στην §2.4 είδαμε για πρώτη φορά τον ορισμό σταθεράς με όνομα:

```
const double g( 9.81 ); // m/sec²
```

Λέμε ότι το "`const`" είναι ένας **περιοριστής τύπου** (type qualifier)· εδώ *περιορίζει* τον τύπο `double`.

Η C++ μας δίνει άλλον έναν περιοριστή τύπου, τον "`volatile`" (ευμετάβλητος). Αν δηλώσεις:

```
volatile int flags;
```

πληροφορείς τον μεταγλωττιστή ότι η *flags* είναι μια μεταβλητή που *η τιμή της αλλάζει συχνά από συμβάντα εκτός προγράμματος* (π.χ. από κάτι που ήλθε στη σειριακή πόρτα). Επομενως, ο μεταγλωττιστής θα πρέπει να μην παίρνει την τιμή της από κάποιον καταχωριτή (για ταχύτερη εκτέλεση) αλλά από τη φυσική της διεύθυνση.

Οι τύποι με περιορισμό "`const`" ή "`volatile`" αναφέρονται συνοπτικώς και ως **τύποι με περιορισμό cv** (cv-qualified).

11.3 Όχι «Εντολή Εκχώρησης» αλλά «Πράξη Εκχώρησης»

Ναι, εντολή εκχώρησης δεν υπάρχει! Αυτό που υπάρχει είναι η *πράξη της εκχώρησης*: Αν έχουμε δηλώσει:

```
T v;
```

τότε με την `v =` *Π* γίνονται τα εξής:

- Υπολογίζεται η τιμή της παράστασης και μετατρέπεται στον τύπο της μεταβλητής `static_type<T>(`*Π*`)`.
- Η τιμή `static_type<T>(`*Π*`)` φυλάγεται ως τιμή της μεταβλητής.
- *Αποτέλεσμα της πράξης* είναι η `v`.[4]

Κι αυτά που λέγαμε μέχρι τώρα; Δεν είναι λάθος, διότι: όπως πιθανότατα έχεις ήδη καταλάβει (μάλλον από κάτι που θα έγραψες κατά λάθος σε κάποιο πρόγραμμα)

♦ ***Μια παράσταση είναι εντολή για τη C++.***

Εμείς θα *ονομάζουμε εντολή εκχώρησης μια εντολή της μορφής "`v` = Π", όπου η Π δεν θα περιέχει πράξεις εκχώρησης.*

Όπως καταλαβαίνεις, έχεις δικαίωμα να γράψεις:

```
w = v = u = 0;
```

Η C++ θα την εκτελέσει ως εξής:

```
w = (v = (u = 0));
```

---

[3] Σύσταση *DCL07*: "Use as minimal scope as possible for all variables and methods". «*Μέθοδοι*»!; Θα μάθεις αργότερα τι είναι αυτό...

[4] Πρόσεξε: «η `v`» και όχι «η τιμή της `v`». Θα καταλάβεις αργότερα...

Δηλαδή: η *u* θα πάρει τιμή 0, ενώ το αποτέλεσμα της πράξης, το 0, εκχωρείται στη *v*. Τέλος, το αποτέλεσμα αυτής της εκχώρησης, πάλι 0, εκχωρείται στη *w*. Άρα, με την εντολή αυτή, με τη μια ή την άλλη μορφή, εκχωρούμε την ίδια τιμή (στην περίπτωσή μας: 0) σε περισσότερες από μια μεταβλητές (στην περίπτωσή μας: τρεις). Θα την ονομάσουμε *εντολή πολλαπλής εκχώρησης*.

Ακόμη, έχεις δικαίωμα να γράψεις:

```
   cout << (v = 1.5) << endl;
```

Εδώ, η τιμή `static_type<T>(1.5)` θα εκχωρηθεί στη *v* και στη συνέχεια το αποτέλεσμα της πράξης (τιμή της *v*) θα εκτυπωθεί. Καλύτερα όμως να αποφεύγεις τέτοια χρήση. Δες το παρακάτω παράδειγμα (Borland C++) για να καταλάβεις:

```
   int v( 0 );
   cout << v << "   " << (v = 1.5) << endl;
```

Από εδώ θα περίμενες να πάρεις:

```
0   1
```

αλλά παίρνεις:

```
1   1
```

Αυτό γίνεται διότι τα ορίσματα της "`cout <<`" υπολογίζονται *από το τέλος προς την αρχή*. Έτσι, πρώτα υπολογίζεται η "`v = 1.5`" από όπου η *v* παίρνει τιμή 1.

Σε μια εκχώρηση, αριστερά του "`=`" δεν είναι απαραίτητο να υπάρχει μεταβλητή· ήδη είδαμε ότι μπορεί να υπάρχει και στοιχείο πίνακα. Γενικώς: *αριστερά του "=" μπορεί να υπάρχει μια* **τιμή-l** (*l*value)[5], *δηλαδή, παράσταση που καθορίζει μια περιοχή της μνήμης*. Φυσικά και το όνομα μιας σταθεράς καθορίζει μια περιοχή της μνήμης, αλλά απαγορεύεται να εμφανιστεί στο αριστερό μέρος μιας εκχώρησης. Στις εκχωρήσεις θέλουμε μια **τροποποιήσιμη** (modifiable) τιμή-*l*.

Τώρα όμως, στον υπολογισμό της εκχώρησης, έχουμε *άλλο ένα βήμα*: τον υπολογισμό της τιμής-*l*, δηλαδή: *τον καθορισμό της περιοχής της μνήμης όπου θα αποθηκευτεί η τιμή της παράστασης*. Και όταν η εκχώρηση τιμής γίνεται σε μια μεταβλητή, δεν έχουμε πρόβλημα· όταν όμως γίνεται σε στοιχείο πίνακα τότε...

Παράδειγμα

Έστω ότι έχουμε:

```
   int k;
   double a[5];
// . . .
   a[0] = 5;  a[1] = 10.5;
// . . .
   k = 0;
   a[k] = a[k] + (k = k+1);
   cout << a[0] << "  " << a[1] << endl;
```

Από τον gcc (Dev-C++) θα πάρουμε:

```
6  10.5
```

που βγαίνει ως εξής: εδώ, πρώτα υπολογίζεται η παράσταση "`a[k]`" σε "`a[0]`" *και για το αριστερό και για το δεξιό μέλος*. Στη συνέχεια υπολογίζεται ο όρος: "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης. Τελικώς, αυτό που μένει να υπολογισθεί είναι το: "`a[0] = (a[0] + 1)`" και έτσι αυξάνεται κατά 1 η τιμή του a[0].

Αν το πρόγραμμά μας μεταγλωττισθεί στη MS Visual C++ v5, θα πάρουμε:

```
5  6
```

[5] Το "*l*" για ιστορικούς λόγους, από το *l*eft part (αριστερό μέρος). Όπως πιθανότατα μαντεύεις, υπάρχει και **τιμή-r** (*r*value) που εμφανίζεται στο δεξιό (*r*ight) μέρος του τελεστή της εκχώρησης.

που βγαίνει ως εξής: Πρώτα υπολογίζεται η παράσταση "`a[k] + (k = k+1)`" αλλά η VC++ προτάσει τον υπολογισμό του "`a[k]`" που είναι το a[0]. Στη συνέχεια υπολογίζεται η "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης "`a[k] + (k = k+1)`" ως "`a[0] + 1`". Τέλος έρχεται η ώρα υπολογισμού της τιμής-*l*, που είναι η a[k]. Αλλά τώρα η *k* έχει τιμή 1 και έτσι τελικώς η εντολή υπολογίζεται ως: "`a[1] = a[0]+1`". Έτσι υπολογίζεται η τιμή της παράστασης: Στη συνέχεια αυξάνεται κατά 1 η τιμή του a[1].

Αν το πρόγραμμά μας μεταγλωττισθεί στη Borland C++ v5, θα πάρουμε:

```
5  11.5
```

που βγαίνει ως εξής: Και εδώ, πρώτα υπολογίζεται η παράσταση "`a[k] + (k = k+1)`" αλλά τώρα πρώτα υπολογίζεται ο όρος: "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης. Έτσι, αυτό που μένει να υπολογισθεί είναι το: "`a[1] = (a[1] + 1)`" και έτσι αυξάνεται κατά 1 η τιμή του a[1].

Οι BC++v5 και gcc έχουν ως κοινό σημείο το ότι η τιμή της "`a[k]`" υπολογίζεται *μια φορά μόνον* είτε ως "`a[1]`" (BC++) είτε ως "`a[0]`" (gcc).

Στο παράδειγμα φαίνεται παραστατικότατα ότι *έχουμε υπολογισμό της τιμής-l*. Λόγω διαφορών στη σειρά εκτέλεσης πράξεων μπορεί να πάρουμε διαφορετικά αποτελέσματα.

Όπως λέγαμε και στην §2.10, η C++ μας δίνει μερικές «συντομογραφίες» πολύ συνηθισμένων εκχωρήσεων. Π.χ., αν τα *x*, *b* είναι οποιουδήποτε αριθμητικού τύπου, αντί για:

```
x = x + b;
```

μπορούμε να γράψουμε:

```
x += b;
```

Παρομοίως και για τις άλλες πράξεις:

`x +=` *Π*`;` είναι ισοδύναμη με: `x = x +` *Π*`;`
`x -=` *Π*`;` είναι ισοδύναμη με: `x = x -` *Π*`;`
`x *=` *Π*`;` είναι ισοδύναμη με: `x = x *` *Π*`;`
`x /=` *Π*`;` είναι ισοδύναμη με: `x = x /` *Π*`;`
`x %=` *Π*`;` είναι ισοδύναμη με: `x = x %` *Π*`;`

Για τις `/=` και `%=` η παράσταση *Π* θα πρέπει να παίρνει τιμή μη μηδενική. Για την τελευταία: η *x* και η τιμή της *Π* θα πρέπει να είναι ακέραιου τύπου.

Ειδικώς για την περίπτωση που θέλουμε να αυξήσουμε ή να μειώσουμε την τιμή μιας μεταβλητής κατά 1, υπάρχουν και άλλες συντομογραφίες:

`++x;` που είναι ισοδύναμη με: `x = x + 1;` ή `x += 1;`
`--x;` που είναι ισοδύναμη με: `x = x - 1;` ή `x -= 1;`

όπως και οι παραλλαγές τους: οι `x++` και `x--` αυξάνουν / μειώνουν την τιμή της *x* κατά 1, αλλά *το αποτέλεσμα της πράξης είναι η αρχική τιμή της x*. Δες το παρακάτω παράδειγμα:

```
 0:   #include <iostream>
 1:   using namespace std;
 2:   int main()
 3:   {
 4:      int x, y; // οποιουδήποτε αριθμητικού τύπου
 5:
 6:      x = 5;  y = ++x;   cout << x << "   " << y << endl;
 7:      x = 5;  y = --x;   cout << x << "   " << y << endl;
 8:      x = 5;  y = x++;   cout << x << "   " << y << endl;
 9:      x = 5;  y = x--;   cout << x << "   " << y << endl;
10:   }
```

Αποτέλεσμα:

```
6   6
4   4
6   5
4   5
```

Όπως βλέπεις, στις γρ. 8-9, η *y* παίρνει την τιμή που είχε η *x* (5) *πριν* αυτή αλλάξει.

Και τι γίνεται με τις συντομογραφίες όταν πρέπει να υπολογιστεί το αριστερό μέρος (τιμή-*l*); Η C++ μας λέει ότι:

- `x θ= y` ***σημαίνει*** `x = x θ y` ***με τη διαφορά ότι η x υπολογίζεται μια φορά μόνον.***

Αν λοιπόν, στο παράδειγμα που είδαμε πιο πριν, γράψουμε:

```
k = 0;
a[k] += (k = k+1);
cout << a[0] << "  " << a[1] << endl;
```

θα πάρουμε είτε (gcc):

```
6  10.5
```

είτε (Borland C++ v5)[6]:

```
5  11.5
```

Στο *Μέρος Α*, θέλοντας να εστιάσουμε την προσοχή μας σε πολύ βασικές και κρίσιμες έννοιες, *αποφύγαμε* –με πολλή προσοχή– τη χρήση αυτών των συντομογραφιών. Από εδώ και πέρα –και πάλι με πολλή προσοχή– *θα τις χρησιμοποιούμε*. Σχετικώς, διάβασε την επόμενη παράγραφο και το Πλ. 11.1.

11.3.1 * Ο "++" για τον Τύπο `bool`

Όπως είπαμε, οι "`++`" και "`--`" μπορούν να δράσουν σε μεταβλητή (γενικώς: τιμή-*l*) οποιουδήποτε αριθμητικού τύπου. Ειδικώς ο πρώτος μπορεί να δράσει και σε μεταβλητές τύπου `bool`. Αν η *b* είναι μια τέτοια μεταβλητή, μετά την εκτέλεση της "`++b`" (ή της "`b++`"), η τιμή της είναι `true`.

Δεν θα το χρησιμοποιήσουμε ποτέ· το λέμε απλώς για να το ξέρεις...[7]

11.4 Οι Εκχωρήσεις και οι Συνθήκες Επαλήθευσης

Ας πούμε ότι έχουμε:

```
// x == 5 && k == 0 && q > 0
   y = (x += q) + ++k;
```

Τι θα ισχύει μετά την εντολή εκχώρησης; Προφανώς όχι αυτά που λέγαμε στο Κεφ. 2. Παρόμοια προβλήματα θα βρεις και στην εφαρμογή των συμπερασματικών κανόνων των **if** και **while** (και αυτών που θα δεις στη συνέχεια) στην περίπτωση που οι συνθήκες περιέχουν εκχωρήσεις.

Βέβαια, ο απαιτητικός αναγνώστης θα πει: «Αν γράψω τα παραπάνω ως εξής:

```
// x == 5 && k == 0 && q > 0
   x += q;  ++k;
   y = x + (k-1);
```

μπορώ να βγάλω συμπέρασμα, έτσι δεν είναι; Γιατί να μην αλλάξουμε τους κανόνες ώστε να καλύπτουμε όλες τις περιπτώσεις;» Απάντηση: *Καλύτερα να κρατήσουμε τους κανόνες απλούς και να γράφουμε όπως στη δεύτερη περίπτωση και όχι όπως στην πρώτη*. Θα τηρούμε αυτήν την αρχή σε ό,τι γράφουμε από εδώ και πέρα, όπως το τηρήσαμε και μέχρι τώρα. Δες το Πλ. 11.1.

Παρατήρηση:▶

Όπως θα δεις στη συνέχεια, θα χρησιμοποιούμε τον προθεματικό τελεστή (`++k`, `--k`) και όχι τον μεταθεματικό (`k++`, `k--`). Ο λόγος είναι ο εξής:

---

[6] Δυστυχώς, η MS VC++ θα δώσει και πάλι: "`5  6`", υπολογίζοντας το a[*k*] μια φορά ως a[0] και μια ως a[1].

[7] Στο C++11 η χρήση του "`++`" για μεταβλητές τύπου `bool` αποθαρρύνεται.

Πλαίσιο 11.1

Συντομογραφίες: Πότε Ναι και Πότε Όχι

1) **ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ** την πολλαπλή εκχώρηση `x = y = z =` *Π*, με την προϋπόθεση, φυσικά, ότι α) η Π δεν έχει εκχωρήσεις και β) δεν υπάρχουν μπερδέματα από τις μετατροπές τύπου.

2) **ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ** εντολές σαν τις:

```
++k;
j++;
p += (q+4);
```

όπου μεταβάλλεται η τιμή μιας μεταβλητής μόνον.

ΔΕΝ ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ εντολές σαν τις:

```
y = ++p + (x = q/2);
x = (y += p+1);
```

όπου μεταβάλλονται τιμές δύο ή περισσοτέρων μεταβλητών. Φυσικά, για εντολές σαν την:

```
a[k] += (k = k+1);
```

ούτε λόγος!

- η "`++k`" είναι ισοδύναμη με τις "`k += 1`" και "`k = k + 1`" και
- η "`--k`" είναι ισοδύναμη με τις "`k -= 1`" και "`k = k - 1`"

σε κάθε χρήση, «νόμιμη» ή «παράνομη».◄

11.5 Παράσταση Υπό Συνθήκη

Στο Κεφ. 5 είδαμε για πρώτη φορά το εξής πρόβλημα: έχουμε δύο πραγματικούς αριθμούς, *x*, *y*, και θέλουμε να εκχωρήσουμε την τιμή του μεγαλύτερου (όχι μικρότερου) από αυτούς σε έναν τρίτο, ας τον πούμε *maxxy*. Μια από τις λύσεις που δώσαμε ήταν η εξής:

```
if ( x > y )  maxxy = x;
        else  maxxy = y;
```

Η C++ σου επιτρέπει να λύσεις το πρόβλημα με μια εντολή εκχώρησης, με χρήση του τελεστή "`?:`"

```
maxxy = (x > y) ? x : y;
```

Η παράσταση δεξιά του "`=`" είναι μια **παράσταση υπό συνθήκη** (conditional expression).

Παράδειγμα
Αντί για την παρακάτω συνάρτηση –που μας δίνει το πρόσημο του *x*:

Πλαίσιο 11.2

Παράσταση Υπό Συνθήκη

Ο τελεστής `?` είναι *τριαδικός*, δηλαδή δέχεται τρία ορίσματα:

συνθήκη, "`?`", *παράσταση*-1, "`:`", *παράσταση*-2

όπου: η συνθήκη είναι λογική παράσταση και οι δύο παραστάσεις δίνουν αποτέλεσμα ίδιου τύπου.

Ο υπολογισμός γίνεται ως εξής: Πρώτα υπολογίζεται η συνθήκη. Αν ισχύει τότε υπολογίζεται η *παράσταση*-1 και η τιμή της είναι τιμή όλης της παράστασης αλλιώς (αν η συνθήκη δεν ισχύει) υπολογίζεται η *παράσταση*-2 και η τιμή της είναι τιμή όλης της παράστασης. Φυσικά, οι *παράσταση*-1 και *παράσταση*-2 μπορεί να είναι, και αυτές, παραστάσεις υπό συνθήκη.

```
int xSign( int x )
{
   int fv( 0 );
   if ( x < 0 )  fv = -1;
   else if ( x > 0 )  fv = 1;
   return fv;
} // xSign
```

μπορείς να γράψεις:

```
int xSign( int x )
{
   return ( x < 0 ) ? -1 : ( x == 0 ) ? 0 : 1;
} // sign
```

Αν το μόνο που μας ενδιαφέρει είναι το πρόσημο της *x* και δεν θέλουμε να γράψουμε συνάρτηση, μπορούμε να γράψουμε:

```
   cout << ( (x < 0) ? -1 : (x == 0) ? 0 : 1 ) << endl;
```

🖘🖘🖘

Οι παρενθέσεις στις συνθήκες δεν είναι απαραίτητες.

11.6 Η Εντολή "for"

Στο Μέρος Α (§6.4) είπαμε ότι «*Η **for** της C++ έχει πολύ περισσότερες δυνατότητες. Προς το παρόν είδαμε –και θα χρησιμοποιούμε– μόνον αυτές που μας χρειάζονται.*» Τώρα ήρθε η ώρα να μάθουμε και τις υπόλοιπες.

Η μορφή της εντολής **for** που χρησιμοποιήσαμε μέχρι τώρα ήταν για την μετρούμενη επανάληψη και μόνον:

```
for ( v = a; v <= t; v = v + b )  E;
```

ή

```
for ( v = a; v >= t; v = v - b )  E;
```

Αν $b > 0$,

- στην πρώτη περίπτωση η *v* παίρνει τιμές *a*, *a*+*b*, ... *a*+*nb* όπου η *n* έχει τιμή τέτοια ώστε: $a + nb \le t < a + (n+1)b$,
- στη δεύτερη περίπτωση η *v* παίρνει τιμές *a*, *a*-*b*, ... *a*-*nb* όπου η *n* έχει τιμή τέτοια ώστε: $a - (n+1)b < t \le a - nb$.

Και στις δύο περιπτώσεις, για κάθε τιμή της *v* εκτελείται η *E*.

Αλλά η for έχει μια πολύ γενικότερη μορφή και πολύ περισσότερες δυνατότητες. Το συντακτικό της είναι:

"**for**", "**(**", *αρχική εντολή της for*, [*συνθήκη*,] "**;**", [*παράσταση*,] "**)**", *εντολή*"**;**"

όπου:

αρχική εντολή της for = { *εντολή-παράσταση* | *απλή δήλωση* } "**;**"

Η *συνθήκη* μπορεί να μην υπάρχει· στην περίπτωση αυτή ότι έχει τιμή **true**. Η *παράσταση* μπορεί επίσης να μην υπάρχει· στην περίπτωση αυτή η φροντίδα για το τέλος της επανάληψης αφήνεται στην επαναλαμβανόμενη *εντολή*. Η *εντολή* και η *αρχική εντολή της for*, όπως θα δούμε στη συνέχεια, μπορεί να είναι κενές.

Ας ξεκινήσουμε με ένα παράδειγμα όπου η *αρχική εντολή της for* είναι απλή δήλωση.

Παράδειγμα 1🖘

Έστω ότι έχουμε δύο μονοδιάστατους πίνακες και μια μεταβλητή

```
   double a[50], b[80], m( 0 );
```

και θέλουμε να αντιγράψουμε τις τιμές των στοιχείων a[0]..a[9] στα στοιχεία b[10]..b[19] και να υπολογίσουμε, στην *m*, τη μέση τιμή των τιμών που αντιγράφουμε. Γράφουμε:

```
   for ( int k(0), j(10);  k < 10;  ++k, ++j )
   {
```

```
      b[j] = a[k];  m += a[k];
   } // for
   m /= 10;
```

Ας ξεκινήσουμε από την *αρχική εντολή της* ***for***. Στην περίπτωσή μας είναι δήλωση δύο μεταβλητών:

```
int k(0), j(10);
```

με την οποία δίνουμε και αρχικές τιμές. Τώρα πρόσεξε το εξής: *Οι k και j είναι γνωστές μόνον μέσα στη* ***for****· δεν μπορείς να τις χρησιμοποιήσεις έξω από αυτήν.*

Η συνθήκη είναι από αυτές που ξέρουμε: "`k <= 9`".

Ποια είναι η παράσταση; Η "`++k, ++j`", δηλαδή μια *ακολουθία παραστάσεων* (δες παρακάτω την §11.11). Τι τιμή θα παίρνει κάθε φορά; Δεν μας νοιάζει! Αυτό που μας ενδιαφέρει είναι ότι κάθε φορά αυξάνει τις τιμές των *k* και *j*.

> *Ακολουθία Παραστάσεων*: Παράσταση που αποτελείται από άλλες παραστάσεις που χωρίζονται ανά δύο με ένα κόμμα. Τιμή της παράστασης είναι αυτή της τελευταίας από τις παραστάσεις που την απαρτίζουν. Για περισσότερα δες παρακάτω την §11.11.

Βλέπουμε λοιπόν ότι:

- ***Μπορούμε να δηλώνουμε, σε μια for, τοπικές μεταβλητές που α) είναι γνωστές μόνον μέσα στη for και β) ζουν όσο διαρκεί η εκτέλεση της for.***

Οι τοπικές μεταβλητές της **for** δεν διαφέρουν σε τίποτε από τις τοπικές μεταβλητές μιας συνάρτησης.

Όταν η *αρχική εντολή της* ***for*** είναι *εντολή-παράσταση* είναι συχνά μια ακολουθία εντολών που δίνει αρχικές τιμές σε ορισμένες μεταβλητές.

Παράδειγμα 2

Ξαναγράφουμε το πρόγραμμα Μέση Τιμή 1 της §5.1:

```
 0:   #include <iostream>
 1:   // πρόγραμμα: Μέση Τιμή 1
 2:   using namespace std;
 3:
 4:   int main()
 5:   {
 6:      const int n( 10 );
 7:
 8:      int m;      // Μετρητής των επαναλήψεων
 9:      double x;   // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
10:      double sum; // Το μερικό (τρέχον) άθροισμα.
11:                  // Στο τέλος έχει το ολικό άθροισμα.
12:      double avrg;// Μέση Αριθμητική Τιμή των x (<x>)
13:
14:      for ( sum = 0, m = 1; m <= n;  m = m + 1 )
15:      {
16:         cout << "Δώσε έναν αριθμό: ";  cin >> x;   // x == tm
17:         sum = sum + x;    // (x == tN) && (sum = Σ(j:1..m)tj)
18:      } // for
19:      avrg = sum / n;
20:      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = "
21:           << avrg << endl;
22:   }
```

Τι κάναμε εδώ; Πήραμε τις δύο εντολές προετοιμασίας της **while**, τις:

```
sum = 0;  m = 1;
```

και τις βάλαμε για *αρχική εντολή της* ***for*** (γρ. 14). Πρόσεξε ότι αντικαταστήσαμε το '`;`' που υπήρχε μεταξύ τους με ένα κόμμα ('`,`') για να σχηματίσουμε μια *ακολουθία παραστάσεων*.

Ακόμη πήραμε τη συνθήκη της **while** και τη βάλαμε ως συνθήκη της **for**.

Τέλος, πήραμε την τελευταία εντολή της περιοχής επανάληψης, την `m = m + 1`, και τη βάλαμε στη θέση της παράστασης. Θα μπορούσαμε να αφήσουμε την εντολή στη θέση της και να μην βάλουμε παράσταση:

```
   for ( sum = 0, m = 1; m <= n; )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
      sum = sum + x;           // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;
   } // for
```

Αλλά και αντιστρόφως: θα μπορούσαμε να μεταφέρουμε στην παράσταση, εκτός από την `m = m + 1`, και την `sum = sum + x`:

```
   for ( sum = 0, m = 1; m <= n; sum = sum + x, m = m + 1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
   } // for
```

Ε, αφού είναι έτσι, γιατί να μην τις πάμε όλες; Και δεν τις παμε…

```
   for ( sum = 0, m = 1;
         m <= n;
         cout << "Δώσε έναν αριθμό: ", cin >> x,
         sum = sum + x, m = m + 1 );
```

ή, με τις συντομογραφίες:

```
   for ( sum = 0, m = 1;
         m <= n;
         cout << "Δώσε έναν αριθμό: ", cin >> x,
         sum += x, ++m );
```

Και έτσι μείναμε χωρίς επαναλαμβανόμενη εντολή!

Όλες αυτές οι **for** που βλέπεις είναι ισοδύναμες με την αρχική **while** μαζί με τις δύο εντολές προετοιμασίας.

Πρόσεξε και κάτι άλλο: *η sum δεν μπορεί να δηλωθεί μέσα στη **for** διότι θέλουμε την τιμή της μετά το τέλος εκτέλεσης της **for***. Η *m* όμως μας χρειάζεται μόνο μέσα στη **for**. Μήπως θα μπορούσαμε να γράψουμε:

```
   for ( sum = 0, int m( 1 ); m <= n; . . . );
```

Όχι! Είπαμε ότι η αρχική εντολή της **for** μπορεί να είναι *εντολή-παράσταση* ή *απλή δήλωση*. Δεν μπορεί να είναι λίγο από το ένα και λίγο από το άλλο.

Αν εγκαταλείψουμε την αρχή «*όλες οι μεταβλητές δηλώνονται στην αρχή της συνάρτησης*» και τηρήσουμε την αρχή «*κάθε μεταβλητή δηλώνεται εκεί που μας χρειάζεται*» το πρόγραμμά μας γράφεται:

```
// πρόγραμμα: Μέση Τιμή 1 με for
#include <iostream>
using namespace std;
int main()
{
   const int n( 10 );

   double sum( 0 );  // Το μερικό (τρέχον) άθροισμα.
                     // Στο τέλος έχει το ολικό άθροισμα.
   for ( int m(1); m <= n; ++m )
   {
      double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
      cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
      sum += x;                // (x == tm) && (sum = Σ(j:1..m)tj)
   } // while

   double avrg( sum / n );     // Μέση Αριθμητική Τιμή των x (<x>)
   cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
} // main
```

Γενικώς, η:

```
Ep1; Ep2; ... Epn;
while ( S )
{  Er1; Er2; ... Erk; Er(k+1); Er(k+2); ... Er(k+m);  }
```

μπορεί να γραφεί ως:

```
for ( Ep1, Ep2, ... Epn; S; Er(k+1), Er(k+2), ... Er(k+m) )
{  Er1; Er2; ... Erk; }
```

Αλλά προσοχή! *Οι εντολές που μεταφέρεις είτε στην αρχική εντολή της **for** είτε στην παράσταση θα πρέπει να είναι εντολές-παραστάσεις* ώστε να δημιουργούνται ακολουθίες παραστάσεων. Δηλαδή δεν μπορείς να μεταφέρεις εκεί εντολές **if**, **ifelse**, **while** ή **for**, από αυτές που ξέρουμε μέχρι τώρα[8]. Έτσι, αν θελήσεις να γράψεις με for την επανάληψη του Μέση Τιμή 4 (§5.1.2) θα γράψεις:

```
for ( sum = 0,  n = 0,  selSum = 0,  selN = 0,
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x;
      x != FROUROS;
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x )
{
   n = n + 1;                  // Αυτά γίνονται για
   sum = sum + x;              // όλους τους αριθμούς
   if (0 < x && x <= 10)       // 'Ελεγχος - Επιλογή
   {
      selSum = selSum + x;      // Αυτά γίνονται μόνο για
      selN = selN + 1;          // τους επιλεγόμενους αριθμούς
   } // if
} // for
```

και η περιοχή της επανάληψης δεν μπορεί να μικρύνει περισσότερο.

Το ότι μπορείς να βγάζεις τις εντολές από την περιοχή επανάληψης και να τα βάζεις στην επικεφαλίδα της **for** δεν σημαίνει ότι πρέπει να το κάνεις οπωσδήποτε. Δες στο προηγούμενο παράδειγμα τη **for** χωρίς περιοχή επανάληψης: αυτό φτάνει και περισσεύει.

Αν θέλεις να αποδείξεις την ορθότητα μιας **for**, σκέψου την ισοδύναμη **while**. Αν, ας πούμε, έχεις:

```
// P
for (Ep1, Ep2, ... Epn; S; Er(k+1), Er(k+2), ... Er(k+m))
{  Er1; Er2; ... Erk; }
// Q
```

θα πρέπει να αποδείξεις (*I* η αναλλοίωτη):

1. ```
   // P
      Ep1; Ep2; ... Epn;
   // I
   ```
2. ```
   // I && S
      Er1; Er2; ... Erk; Er(k+1), Er(k+2), ... Er(k+m);
   // I
   ```
3. (*I* && !*S*) ⇒ *Q*

11.7 Η Εντολή "do-while"

Εκτός από τη **while** και τη **for**, η C++ έχει και μια άλλη εντολή επανάληψης, τη **do-while**. Δες το Πλ. 11.3 και το Σχ. 11-1.

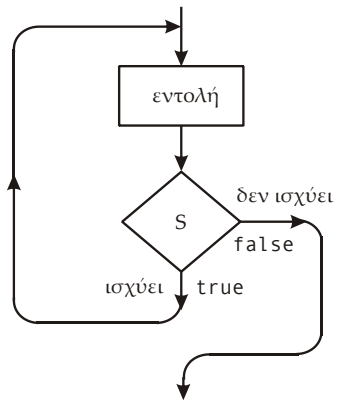
Η σημαντική διαφορά της από τη **while** είναι ότι

- στη **while** πρώτα ελέγχεται η συνθήκη και η εντολή εκτελείται μόνον αν ισχύει, ενώ
- στη **do-while** πρώτα εκτελείται η εντολή και μετά ελέγχεται η συνθήκη.

Έτσι:

[8] Και είναι εντολές-παραστάσεις οι εντολές εισόδου και εξόδου; Ναι, είναι! Διάβασε παρακάτω...

♦ ***Ενώ στη while η επαναλαμβανόμενη εντολή μπορεί να μην εκτελεσθεί ούτε μία φορά, στη do-while θα εκτελεσθεί μία φορά τουλάχιστον.***



Σχ. 11-1 Λογικό διάγραμμα της **do-while**.

Είναι σαφές ότι μπορούμε να ζήσουμε και χωρίς τη **do-while**. Ας δούμε όμως ένα παράδειγμα που είναι προτιμότερη από τη **while**. Έστω ότι θέλουμε να διαβάσουμε από το πληκτρολόγιο έναν *θετικό* αριθμό. Να τι θα κάναμε με τη **while**:

```
cout << " Δώσε θετικό: ";  cin >> x;
while ( x <= 0 )
{
   cout << " Δώσε θετικό: ";  cin >> x;
} // while
```

Με τη **do-while** τα πράγματα είναι απλούστερα:

```
do {
   cout << " Δώσε θετικό: ";  cin >> x;
}  while ( x <= 0 );
```

Ο συμπερασματικός κανόνας της **do-while** είναι λίγο πιο πολύπλοκος από αυτόν της **while**. Και εδώ βέβαια υπάρχει *αναλλοίωτη* που ισχύει πριν από την εντολή και μετά από αυτήν. Φυσικά, μετά την εντολή δεν ισχύει η συνθήκη συνέχισης. Έστω ότι έχουμε:

```
// I
do E while ( S );
// I && !S
```

Έστω τώρα ότι:

$$I\ \{\ E\ \}\ Q$$

Αν μετά την εκτέλεση της E ισχύει η S θα ξαναεκτελεσθεί η E. Θα πρέπει λοιπόν από την Q && S να συνάγεται η αναλλοίωτη. Έτσι, φτάνουμε στον συμπερασματικό κανόνα της **do-while**:

$$\frac{I\{E\}Q,\ \ Q\&\&S \Rightarrow I}{I\ \{\texttt{do}\ E\ \texttt{while}\ (S)\}\ I\ \&\&\ (!S)}$$

Πώς θα μπορούσαμε να υλοποιήσουμε τη **while** με τη **do-while**; Όπως είπαμε παραπάνω «στη **while** πρώτα ελέγχεται η συνθήκη και η εντολή εκτελείται μόνον αν ισχύει, ενώ στη **do-while** πρώτα εκτελείται η εντολή και μετά ελέγχεται η συνθήκη». Το πρόβλημά μας λοιπόν είναι να ελέγξουμε την πρώτη φορά εκτέλεσης της E. Αυτό μπορεί να γίνει με μια

Πλαίσιο 11.3

Η Εντολή do-while

"`do`", *εντολή*, "`while`", "`(`", *παράσταση*, "`)`" ;

και εκτελείται ως εξής:

εκτελείται η *εντολή*
υπολογίζεται η τιμή της παράστασης
αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
αλλιώς, αν είναι `true` τότε

> εκτελείται η *εντολή*
> υπολογίζεται η τιμή της παράστασης
> αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
> αλλιώς, αν είναι `true` τότε
>
> > εκτελείται η *εντολή*
> > υπολογίζεται η τιμή της παράστασης
> > κ.ο.κ.

if. Έτσι, οι:

```
while ( S )
{
   E
}
```

```
if ( S )
{
   do { E } while ( S );
}
```

είναι ισοδύναμες.

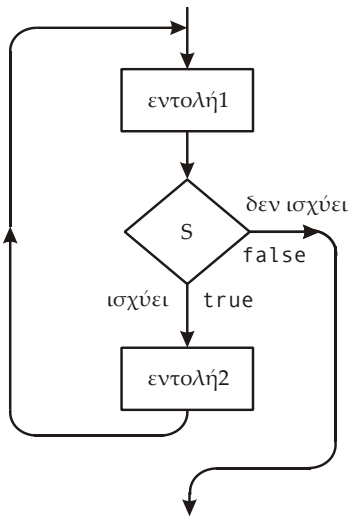
11.8 Η Επανάληψη "n+½" – Η Εντολή "break"

Μέχρι τώρα μάθαμε επαναληπτικές εντολές που ελέγχουν τη συνθήκη συνέχισης στην αρχή, όπως η **while** και η **for**, ή στο τέλος, όπως η **do-while**, των επαναλαμβανόμενων εντολών. Φυσικά, η πιο γενική περίπτωση είναι όταν έχουμε τον έλεγχο κάπου ενδιάμεσα. Π.χ. στη γλώσσα Ada υπάρχει η εντολή:

```
loop
  E1
when St exit;
  E2
endloop
```

Εδώ, εκτελείται η εντολή *E*1 και ελέγχεται η τιμή της συνθήκης *St*:

- αν έχει τιμή `true` τότε διακόπτεται η επανάληψη και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί μετά το `endloop`,
- αν έχει τιμή `false` τότε εκτελούνται οι εντολές *E*2 και *E*1, ελέγχεται η τιμή της συνθήκης *St* κ.ο.κ.



Σχ. 11-2 Λογικό διάγραμμα της επανάληψης *n*+½.

Αυτή η επανάληψη λέγεται **επανάληψη *n*+½**, διότι την τελευταία φορά εκτελείται μόνο η «μισή» περιοχή επανάληψης (*E*1).

Η C++ δεν έχει τέτοια εντολή αλλά δεν είναι δύσκολο να την υλοποιήσουμε. Γράφουμε μια αέναη επανάληψη και μετά θα πρέπει να βρούμε έναν τρόπο για να βγαίνουμε από την επανάληψη όταν θέλουμε. Το «όταν θέλουμε» μπορεί να γίνει με μια **if**. Για την έξοδο από την επανάληψη η C++ μας δίνει την εντολή `break`· εκτέλεσή της έχει ως αποτέλεσμα να συνεχιστεί η εκτέλεση με την εντολή που ακολουθεί την επανάληψη.

Υλοποιούμε λοιπόν την επανάληψη *n*+½ ως εξής:

```
while ( true )
{
   E1
if ( !S ) break;
   E2
} // while
```

ή

```
for ( ; ; )
{
   E1
if ( !S ) break;
   E2
} // for
```

ή

```
do
{
   E1
if ( !S ) break;
   E2
} while ( true );
```

Όπως βλέπεις, σε σχέση με την εντολή της Ada, κάναμε μια μικρή αλλαγή: γράψαμε την **if** έτσι που η συνθήκη *S* να είναι συνθήκη συνέχισης, όπως συμβαίνει στις εντολές επανάληψης της C++.

Και εδώ θα έχουμε μια αναλλοίωτη, *I*, που θα ισχύει πριν ενώ μετά θα ισχύει η: *I* **&&** (**!***S*). Για την εντολή *E*1 θα έχουμε:

$$I \{ E1 \} Q$$

Μετά την *E*2 θα εκτελεσθεί ξανά η *E*1· θα πρέπει λοιπόν μετά την *E*2 να ξαναπαίρνουμε την αναλλοίωτη:

$$Q \;\&\&\; S \{ E2 \} I$$

Αν μετά την *E*1 δεν ισχύει η *S* η εκτέλεση της επανάληψης διακόπτεται. Θα πρέπει λοιπόν να έχουμε:

$$(Q \texttt{ \&\& !}S) \Rightarrow (I \texttt{ \&\& !}S)$$

Μπορούμε να απλουστεύσουμε τα πράγματα ζητώντας η Q να είναι η αναλλοίωτη I και να γράψουμε τον κανόνα:

$$\frac{I\texttt{\{}E1\texttt{\}}I, I\texttt{\&\&}S\texttt{\{}E2\texttt{\}}I}{I\texttt{\{ while(true)\{}E1\texttt{ if(!}S\texttt{)break;}E2\texttt{\}\}}I\texttt{\&\&(!}S\texttt{)}}$$

Και κάτι ακόμη για τη **break**: Αν έχεις επαναληπτικές εντολές φωλιασμένες, τη μία μέσα στην άλλη, η **break** διακόπτει την εκτέλεση της πιο εσωτερικής επανάληψης μέσα στην οποία βρίσκεται. Για παράδειγμα, δες την παρακάτω περίπτωση:

```
      while ( S1 )
      {
          :
         do {
            . . .
              for (...)
              {
                :
                ... break; ...
              } // for
 A ------>   . . .
         } while ( S2 );
           . . .
      } // while
```

Τρεις επαναληπτικές εντολές, η μια μέσα στην άλλη και στην πιο εσωτερική μια **break**. Τι θα συμβεί με την εκτέλεσή της; Θα διακοπεί η εκτέλεση *της for μόνον* και η εκτέλεση του προγράμματος θα συνεχιστεί στο σημείο Α.

11.9 Η Εντολή "switch"

Ξαναδές τα παραδ. 1 και 2 στην §5.3· και τα δύο είναι προγράμματα με πολλαπλές επιλογές, το δεύτερο όμως, για τον υπολογιστή τσέπης, έχει το εξής χαρακτηριστικό: η εντολή που θα εκτελεσθεί επιλέγεται από την τιμή μιας μεταβλητής (*oper*)· για διαφορετικές –αποδεκτές– τιμές της μεταβλητής εκτελούνται διαφορετικές εντολές.

Η C++ μας δίνει τη δυνατότητα να γράφουμε τέτοια προγράμματα πιο παραστατικά, με χρήση της εντολής **switch**:

```
switch ( παράσταση )
{
   case c1: E1
   case c2: E2
     :
   default: Ed
}
```

Η **switch**, μετά τη λέξη-κλειδί **switch**, περιμένει, μέσα σε παρενθέσεις, μια παράσταση που υπολογίζει τιμή ακέραιου τύπου (**int**, **long int**, **char**, **bool** κλπ). Στη συνέχεια έχει μια (σύνθετη) εντολή. Στην εντολή αυτή υπάρχουν εντολές με *ετικέτες* της μορφής "**case**", *σταθερά*. Στο τέλος μπορεί να υπάρχει και εντολή με την ετικέτα **default**.

Όταν εκτελείται η **switch**:

- Υπολογίζεται κατ' αρχάς η τιμή της παράστασης.
- Στη συνέχεια η εκτέλεση συνεχίζεται από την εντολή που η ετικέτα της έχει σταθερά ίση με την τιμή της παράστασης.
- Αν δεν υπάρχει εντολή με τέτοια σταθερά η εκτέλεση συνεχίζεται με την εντολή που έχει ετικέτα **default** (αν υπάρχει).

Όπως καταλαβαίνεις, θα πρέπει να βάζεις ετικέτα **default** όταν οι σταθερές στις ετικέτες δεν καλύπτουν όλες τις τιμές που μπορεί να πάρει η παράσταση.

Αλλά προσοχή! Ας πούμε ότι η παράσταση πήρε την τιμή *c*1. Στην περίπτωση αυτή θα εκτελεσθούν οι εντολές *E*1 και η εκτέλεση θα συνεχιστεί με τις εντολές *E*2 κλπ. Αν θέλεις να εκτελεσθούν μόνον οι *E*1 θα πρέπει να βάλεις στο τέλος τους μια εντολή **break**.

Ας δούμε πώς γράφεται το πρόγραμμα του υπολογιστή τσέπης με τη **switch**:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;          // Τα ορίσματα της πράξης
   char oper;              // Το σύμβολο της πράξης

   cin >> oper >> x1 >> x2;
   switch ( oper )
   {
      case '+':  cout << (x1 + x2) << endl;  break;
      case '-':  cout << (x1 - x2) << endl;  break;
      case '*':  cout << (x1 * x2) << endl;  break;
      case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                           else  cout << " Δεν γίνεται" << endl;
                 break;
       default:  cout << " Η πράξη (+,-,*,/) στην πρώτη θέση"
                      << endl;
   } // switch
} // main
```

Στην περίπτωσή μας παράσταση είναι η μεταβλητή *oper* τύπου `char`. Στη συνέχεια έχει μια (σύνθετη) εντολή. Στην εντολή αυτή υπάρχουν εντολές με ετικέτες `case '+'`, `case '-'`, `case '*'`, `case '/'`, `default`. Όταν εκτελείται η **switch** ελέγχεται κατ' αρχάς η τιμή της *oper*. Ας πούμε ότι αυτή βρίσκεται να είναι: `'*'`. Στην περίπτωση αυτήν εκτελείται η εντολή "`cout << (x1 * x2) << endl`" και η "`break`" που την ακολουθεί, οπότε και διακόπτεται η εκτέλεση της **switch**.

Έστω τώρα, ότι θέλουμε να αναγνωρίζει ως σύμβολο πολλαπλασιασμού και το (γράμμα) `'x'` αλλά, όταν δοθεί, εκτός από το αποτέλεσμα να γράφει και το μήνυμα `"προτιμότερο το '*'"`. Να πώς πρέπει να γραφεί το πρόγραμμά μας:

```
   switch ( oper )
   {
      case '+':  cout << (x1 + x2) << endl;  break;
      case '-':  cout << (x1 - x2) << endl;  break;
      case 'x':  cout << "προτιμότερο το \'*\'" << endl;
      case '*':  cout << (x1 * x2) << endl;  break;
      case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                           else  cout << " Δεν γίνεται" << endl;
                 break;
       default:  cout << " Η πράξη (+,-,*,/) στην πρώτη θέση"
                      << endl;
   } // switch
```

Αν δώσουμε:

```
* 3 4
```

παίρνουμε:

```
12
```

ενώ αν δώσουμε:

```
x 3 4
```

παίρνουμε:

```
προτιμότερο το '*'
12
```

Και αν θέλουμε να βγάζει το αποτέλεσμα αλλά όχι το μήνυμα; Τότε δεν βάζουμε την εντολή που βγάζει το μήνυμα. Η παρακάτω **switch** δέχεται τα `'x'`, `'*'` για πολλαπλασιασμό και τα `'/'`, `':'` για διαίρεση:

```
   . . .
   case '-':  cout << (x1 - x2) << endl;  break;
   case 'x':
   case '*':  cout << (x1 * x2) << endl;  break;
   case ':':
   case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                        else  cout << " Δεν γίνεται" << endl;
              break;
   . . .
```

Πώς θα μπορούσαμε να υλοποιήσουμε τις **if** και **ifelse** μόνον με τη **switch**; Αφού λάβουμε υπόψη μας ότι ο `bool` είναι ακέραιος τύπος, έχουμε τις εξής ισοδυναμίες:

```
if ( S )
{
   Et
}
```

```
switch ( S )
{
   case true: Et;
}
```

```
if ( S )
{
   Et
}
else
{
   Ef
}
```

```
switch ( S )
{
   case true:  Et; break;

   case false: Ef;
}
```

- ***Όταν έχουμε πολλαπλές επιλογές που εξαρτώνται από την (ακέραιη) τιμή μιας παράστασης θα προτιμούμε την εντολή switch από τις φωλιασμένες ifelse.***

11.9.1 Τοπικές Μεταβλητές στη "switch"

Η **switch** έχει τη δική της σύνθετη εντολή (από τη '{' μέχρι τη '}'). Μπορούμε λοιπόν μέσα σε αυτήν να δηλώσουμε τοπικές μεταβλητές! Μπορούμε; Για δες ένα παράδειγμα:

```
switch ( oper )
{
   int q( 5 );

   case '+':  . . .
   case '-':  . . .
   . . .
} // switch
```

Πρόσεξε τώρα: ο μεταγλωττιστής θα πρέπει να βάλει εντολές δημιουργίας της μεταβλητής *q* «αμέσως μετά την '{'» (ας πούμε...). Θα βάλει εντολές καταστροφής της *q* «αμέσως πριν από την '}'» (ας πούμε πάλι...). Μόνο που η καταστροφή θα γίνει οπωσδπήποτε ενώ η δημιουργία δεν θα γίνει ποτέ! Γιατί; Διότι μετά τον έλεγχο της τιμής της *oper* η εκτέλεση θα προχωρήσει κατ' ευθείαν σε κάποιαν από τις ετικέτες case παρακάμπτοντας τη δήλωση της *q*.

Το ίδιο θα συμβεί και σε περιπτώσεις σαν την:

```
switch ( oper )
{
   case '+':  int q( 5 ); . . .
   case '-':  . . .
   . . .
} // switch
```

αν δεν επιλεγεί η περίπτωση '+'.

Φυσικά, δεν έχεις κανένα τέτοιο πρόβλημα στην περίπτωση:

```
switch ( oper )
{
   case '+':  {  int q( 5 ); . . .  }
   case '-':  . . .
   . . .
```

```
   } // switch
```

11.10 * Ετικέτες – Η Εντολή "goto"

Οι περισσότερες γλώσσες προγραμματισμού περιλαμβάνουν την περίφημη (με την κακή έννοια) εντολή **goto** (Dijkstra 1988)· την έχει και η C++. Μέχρι τώρα αποφύγαμε να την παρουσιάσουμε, μια και θα έπρεπε να τη συνοδεύουμε με τη συμβουλή «μην τη χρησιμοποιείς». Τώρα, μπορούμε να την παρουσιάσουμε, να δούμε από που ξεκινούν τα προβλήματα και να καταλάβουμε πότε μπορούμε να τη χρησιμοποιούμε και πότε όχι. Πάντως, η χρήση της δεν είναι απαραίτητη, όπως αποδεικνύεται με το θεώρημα Bohm-Jacopini (Bohm & Jacopini 1966).

Ας ξεκινήσουμε όμως με τις **ετικέτες** (labels), που είναι απαραίτητες για τη χρήση της **goto**.

Εκτός από τις ετικέτες "`default`" και "`case`", *σταθερά*, που είδαμε ότι χρησιμοποιούνται στη **switch**, η C++ σου επιτρέπει να βάλεις μπροστά από οποιαδήποτε εντολή, μια **ετικέτα** π.χ.:

```
gvl: cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
```

Το `gvl` είναι η ετικέτα της εντολής

```
cout << " Δώσε...ΤΕΛΟΣ: "
```

Στη C++ η ετικέτα είναι ένα αναγνωριστικό και μπαίνει μπροστά από οποιαδήποτε εντολή, ακολουθούμενη από το χαρακτήρα `':'`.

Η εντολή **goto** είναι η πιο «ωμή» παρέκκλιση από τη σειριακή εκτέλεση. Τό νόημά της είναι: «Μη συνεχίσεις με την επόμενη εντολή αλλά με την εντολή που έχει την ετικέτα που σου δίνω».

Η διαφορά της από τις εντολές υπό συνθήκη ή τις επαναληπτικές είναι η εξής: εκεί λέγαμε: παράκαμψε αυτές τις εντολές (ή εκτέλεσε ξανά και ξανά αυτές τις εντολές) αν ισχύει αυτή η συνθήκη. Εδώ είναι σαν να λέμε: πήγαινε να συνεχίσεις εκεί, γιατί έτσι μου αρέσει! Ή τουλάχιστον, έτσι φαίνεται, πολύ συχνά.

Δες το παρακάτω παράδειγμα:

```
    if ( x < 0 )  goto ts;
       absX = x;
       goto nxt;
 ts: absX = -x;
nxt: cout << x << "  " << absX << endl;
```

Τί κάνουν αυτές οι εντολές; Αυτά τα απλά:

```
     if ( x >= 0 )  absX = x;
              else  absX = -x;
     cout << x << "  " << absX << endl;
```

Πριν προχωρήσουμε να απαριθμήσουμε τα δεινά που θα πέσουν στο κεφάλι μας (ή τουλάχιστον στο πρόγραμμά μας) αν χρησιμοποιούμε την εντολή **goto**, ας δούμε μερικούς περιορισμούς σχετικά με τη χρήση της.

Και ας ξεκινήσουμε με την **εμβέλεια** (scope) μιας ετικέτας.

- Μια ετικέτα μπαίνει σε μια μόνο εντολή στο σώμα μιας συνάρτησης και είναι γνωστή μόνο μέσα στη συνάρτηση αυτή.

Φυσική συνέπεια του παραπάνω περιορισμού είναι ο περιορισμός: *με τη **goto** μπορείς να πηγαίνεις σε άλλα σημεία της ίδιας συνάρτησης*.

Η C++ βάζει άλλον ένα περιορισμό σχετικά με τη χρήση της **goto**, αλλά τον αφήνουμε για αργότερα, αφού τώρα δεν μπορείς να τον κατανοήσεις. Θα βάλουμε όμως έναν περιορισμό που έχει σχέση με τον σωστό προγραμματισμό:

♦ ***Η goto δεν πρεπει να παραπέμπει απ' έξω προς το εσωτερικό δομημένων εντολών.***

Τα παρακάτω *δεν επιτρέπονται*:

```
    . . .
    goto lb55;
    . . .
    while ( i <= n )
    {
          . . .
lb55:  x = x - 1;
          . . .
    } // while
```

(η `goto lb55` παραπέμπει στο εσωτερικό της **while**)

```
    . . .
    if (x <= 0)  goto lb66;
    . . .
    if (x == 1)
       a = a + 1;
    else
lb66:  n = n + 1;
```

(η `goto lb66` παραπέμπει στο εσωτερικό της **ifelse**)

Όπως θα κατάλαβες, από το παράδειγμα που δώσαμε πιο πάνω, η χρήση της **goto** κάνει ένα πρόγραμμα δυσανάγνωστο. Και επειδή είναι παραδεκτό απ' όλους ότι «ένα καλό πρόγραμμα είναι και ευανάγνωστο», η χρήση της **goto** βάει ερωτηματικά για την ποιότητα του προγράμματος.

Τα προβλήματα που προέρχονται από τη χρήση της **goto**, πιο συγκεκριμένα, είναι:

- δυσκολία επαλήθευσης του προγράμματος,
- δυσκολία τροποποίησης του προγράμματος.

Θα τα μελετήσουμε στην επόμενη παράγραφο.

11.10.1 Προβλήματα με τη Χρήση της Εντολής goto

Κοίταξε το κομμάτι (υποθετικού) προγράμματος που δίνουμε παρακάτω:

```
        . . .
_________________ A
      goto lb75;
        . . .
_________________ B
 lb75: . . .
       . . .
```

Η εντολή που υπάρχει στην `lb75` θα εκτελείται είτε μετά από εκτέλεση της εντολής `goto lb75` είτε, με την κανονική (σειριακή) ροή του προγράμματος, μετά την εκτέλεση της προηγούμενής της εντολής. Έτσι, η συνθήκη που απαιτούμε να ισχύει πριν από την εκτέλεση αυτής της εντολής θα πρέπει να συνάγεται και από την P_A και από την P_B. Αυτό δεν είναι τετριμμένο. Και αν εξασφαλισθεί όταν γράφεται αρχικά το πρόγραμμα, είναι πολύ πιθανό ότι θα πάψει να ισχύει όταν γίνει μια διόρθωση ή μια τροποποίηση.

Αν έχεις βέβαια περισσότερες από μια **goto** για την ίδια ετικέτα, τα πράγματα χειροτερεύουν.

Υπάρχουν περιπτώσεις που δεν υπάρχει πρόβλημα με τη χρήση της **goto**; Ναι, όταν δεν υπάρχουν απαιτήσεις από την εντολή που έχει την ετικέτα ή υπάρχουν απαιτήσεις περιορισμένες. Χαρακτηριστική περίπτωση η διακοπή εκτέλεσης λόγω κάποιας απρόβλεπτης κατάστασης (εξαίρεσης), π.χ.

```
lb99: switch ( errCode )
      {
         case 1: cout << " Διαίρεση δια 0" << endl;

         case 2: cout << " Απρόβλεπτο τέλος στο Αρχείο" << endl;

         case 3: cout << " Πολλές τιμές για τον πίνακα Z"
```

```
                    << endl;
          . . .
    } // switch
    exit( EXIT_FAILURE );
```

Τί απαίτηση έχουμε όταν εκτελείται κάποια `goto lb99`; Η *errCode* να έχει το σωστό κωδικό σφάλματος, ώστε να γραφεί το σωστό μήνυμα. Μετά... έχουμε "`exit(EXIT_FAILURE)`". Αυτή η χρήση της **goto** δεν είναι τραγική.

Τώρα ξέρεις!

- Η C++ σου δίνει τη **goto** αλλά και όσα εργαλεία σου χρειάζονται για να μη τη χρησιμοποιείς. Στο κεφάλαιο αυτό προσπαθήσαμε να σου παρουσιάσουμε αυτά τα εργαλεία σε βάθος: για να καταλάβεις το λόγο ύπαρξής τους και την ανάγκη για τη χρήση τους. Χρησιμοποίησέ τα και *απόφυγε τη χρήση της* ***goto***.
- Οι μεγάλοι μάστορες ξέρουν καλά τους κανόνες, όπως «*μη χρησιμοποιείς τη* ***goto***», ξέρουν όμως και πότε να τους παραβιάζουν. Αυτό προσπαθήσαμε να σου δείξουμε στην παράγραφο αυτήν. Όταν θα νοιώσεις ότι έγινες μεγάλος μάστορας χρησιμοποίησε και τη **goto**! Φυσικά, ο τρόπος που θα τη χρησιμοποιήσεις θα δείχνει και πόσο μεγάλος μάστορας είσαι...

Ένας μεγάλος μάστορας, ο D. Knuth, έχει γράψει μια, κλασική πια, ανάλυση σε βάθος σχετικά με την εντολή **goto** (Knuth 1977). Αξίζει να τη μελετήσεις· θα σου μάθει πολλά.

11.11 * Η Εντολή "continue"

Η εντολή `continue` χρησιμοποιείται μέσα στην περιοχή επανάληψης μιας **while** ή μιας **do-while** ή μιας **for** και σου δίνει τη δυνατότητα να τερματίσεις μια εκτέλεσή της.

- Αν η επανάληψη γίνεται με **while** ή με **do-while**: μετά την (εκτέλεση της) `continue` η εκτέλεση συνεχίζεται με υπολογισμό της συνθήκης.
- Αν η επανάληψη γίνεται με **for**: μετά την `continue` υπολογίζεται η *παράσταση* (τρίτο τμήμα της **for**) και στη συνέχεια υπολογίζεται η συνθήκη.

Η επανάληψη του *Μέση Τιμή* 1, που γράψαμε με **for** στην §11.7, θα μπορούσε να γραφεί ως εξής:

```
for ( sum = 0,  n = 0,  selSum = 0,  selN = 0,
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x;
      x != FROUROS;
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x )
{
   n = n + 1;                          // Αυτά γίνονται για
   sum = sum + x;                      // όλους τους αριθμούς
   if ( x <= 0 || 10 < x ) continue;   // Έλεγχος - Επιλογή
   selSum = selSum + x;                // Αυτά γίνονται μόνο για
   selN = selN + 1;                 // τους επιλεγόμενους αριθμούς
} // for
```

Βέβαια, όπως βλέπεις, η συνθήκη αυτής της **if** είναι η *αντίθετη* της συνθήκης που είχαμε αρχικώς.

Όπως καταλαβαίνεις, μπορείς να ζήσεις και χωρίς την **continue**.

11.12 * Ακολουθία Παραστάσεων

Η C++ έχει κληρονομήσει από τη C και μια «περίεργη» πράξη: την ακολουθία παραστασεων, όπου ο τελεστής της πράξης είναι το κόμμα (`,`). Ας πούμε ότι δηλώνουμε:

```
double x( 0.1 ), y( 3.7 ), z( 7.4 ), q;
```

και δίνουμε:

```
q = (x, y, z);
```

```
cout << q << endl;
```

Μέσα στην παρένθεση έχουμε τρεις (πολύ απλές) παραστάσεις. Αυτό που θα γίνει είναι το εξής: θα υπολογισθούν οι τιμές των τριών παραστάσεων –δηλαδή: 0.1, 3.7 και 7.4 αντιστοίχως– και η τιμή της τελευταίας είναι τιμή της ακολουθίας παραστάσεων που θα εκχωρηθεί στην *q*. Πράγματι, η εντολή εξόδου μας δίνει:

```
7.4
```

Η παρένθεση χρειάζεται; Ναι! Δες τι θα γίνει αν γράψουμε:

```
q = x, y, z;
cout << q << endl;
```

Αποτέλεσμα:

```
0.1
```

Γιατί; Διότι ο τελεστής `=` της εκχώρησης έχει μεγαλύτερη προτεραιότητα από το "`,`" και έτσι πρώτα θα εκχωρηθεί η τιμή της *x* στην *q* και μετά θα υπολογισθεί η τιμή της ακολουθίας (που θα είναι και πάλι 7.4).

Να και μια πιο δύσκολη περίπτωση: Οι

```
q = (x = 8.3, y = x - 4, z = y - 4);
cout << q << "  " << x << "  " << y << "  " << z << endl;
```

θα δώσουν:

```
0.3  8.3  4.3  0.3
```

Όπως είδες, χρησιμοποιήσαμε την ακολουθία παραστάσεων στην εντολή **for** και μόνον. Γενικώς, δεν θα τη δεις και πολύ συχνά μπροστά σου.

11.13 Υπολογισμός Παράστασης

Στον πίνακα του Παρ. Ε βλέπεις τα χαρακτηριστικά των πράξεων που έχουμε μάθει μέχρι τώρα. Δεν βλέπεις την προσεταιριστικότητα των πράξεων, αλλά γι' αυτήν ο κανόνας είναι απλός:

- η προσεταιριστικότητα των δυϊκών πράξεων, εκτός από την εκχώρηση, είναι από τα αριστερά προς τα δεξιά,
- η προσεταιριστικότητα των ενικών πράξεων και της εκχώρησης είναι από τα δεξιά προς τα αριστερά.

Για παράδειγμα: η `a + b + c` υπολογίζεται ως `(a + b) + c`, ενώ η `a = b = c` υπολογίζεται ως `a = (b = c)`.

Αλλά προσοχή! Αν έχουμε `a*b + c/d + f(e)` ο παραπάνω κανόνας μας λέει *μόνον* ότι ο υπολογισμός θα γίνει ως εξής: `(a*b + c/d) + f(e)`.

- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `(a*b + c/d)` και μετά η `f(e)`.
- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `a*b` και μετά η `c/d`.

Κάθε υλοποίηση της C++ θα κάνει τους υπολογισμούς με τη σειρά που «θέλει». Αν θέλεις να επιβάλεις συγκεκριμένη σειρά εκτέλεσης των πράξεων θα πρέπει να σπάσεις την παράσταση σε μικρότερες και να τις γράψεις με την κατάλληλη σειρά.

Στις λογικές πράξεις:

- Αν ένας από τους παράγοντες της πράξης "`&&`" υπολογιστεί `false` δεν είναι καθόλου σίγουρο ότι θα υπολογιστεί και ο άλλος, αφού το αποτέλεσμα της πράξης θα είναι `false`.
- Αν ένας από τους παράγοντες της πράξης "`||`" υπολογιστεί `true` δεν είναι καθόλου σίγουρο ότι θα υπολογιστεί και ο άλλος, αφού το αποτέλεσμα της πράξης θα είναι `true`.

Στο Παρ. Ε μπορείς να βρεις επίσης τις *Συνήθεις Αριθμητικές Μετατροπές* (ΣΑΜ), δηλαδή ορισμένες μετατροπές τύπων που γίνονται αυτομάτως όταν υπολογίζεται μια αριθ-

μητική παράσταση. Οι ΣΑΜ εφαρμόζονται όταν γίνονται οι πράξεις "`+`", "`-`", "`*`", "`/`" μεταξύ αριθμητικών τιμών και καθορίζουν τον τύπο του αποτελέσματος.

11.14 Εν Κατακλείδι...

Όπως βλέπεις, η C++ μας δίνει πολλά εργαλεία. Πότε θα τα χρησιμοποιούμε και πότε όχι; Μια γενική συμβουλή είναι η εξής:

- ***Χρησιμοποιούμε τα προγραμματιστικά εργαλεία στο βαθμό που το πρόγραμμά μας γράφεται έτσι που να είναι καθαρή η λογική του και απλή η απόδειξη ορθότητας.***

Εξειδικεύουμε τη συμβουλή μας:

1. Μπορείς να χρησιμοποιείς τις συντομογραφίες της εκχώρησης σε ανεξάρτητες εντολές αλλά όχι μέσα σε παραστάσεις. Δηλαδή, δεν υπάρχει πρόβλημα αν γράψεις `x += y*a/q` ή `++p` αλλά απόφευγε να γράφεις `y = ++p + (x = q/2)`. *Σε κάθε εντολή εκχώρησης θα πρέπει να αλλάζει η τιμή μιας μεταβλητής μόνον*. Η μόνη περίπτωση που εξαιρείται από τον κανόνα είναι η πολλαπλή εκχώρηση: `x = y = z =` *Π*, με την προϋπόθεση, φυσικά, ότι η *Π* δεν έχει εκχωρήσεις.
2. Η **for** είναι βολική. Γενικά είναι πολύ βολικό να βλέπεις *σε μια γραμμή*, αρχικές εντολές, συνθήκη συνέχισης και τρόπο μεταβολής των μεταβλητών σου. Χρησιμοποίησε ως οδηγό τη *μία γραμμή*. Αν την υπερβαίνεις καλύτερα να χρησιμοποιείς **while**.
3. Στη **switch** μη ξεχνάς τις **break**. Αν πρέπει να μη βάλεις **break** (σαν το προτελευταίο παράδειγμα της §11.10) γράψε σχόλιο που να το τονίζει.
4. Απόφευγε τη χρήση της **goto**.

Αυτά ως συμβουλές· βέβαια υπάρχει και η πείρα που οδηγεί σε προσωπικές προτιμήσεις και σε προσωπικό ύφος. Στο τέλος-τέλος, δημοκρατία έχουμε και... de gustibus et coloribus non disputandum est.[9]

Ασκήσεις

Α Ομάδα

11-1 Στην άσκ. 5-4 (*Μέρος Α*) ρωτούσαμε «*ποια μαθηματική συνάρτηση υλοποιούν οι παρακάτω εντολές (όπου οι μεταβλητές a, b, c και d είναι τύπου* `int`)»:

```
if ( a > b )  y = a; else y = b;
if ( c > y )  y = c;
if ( d > y )  y = d;
```

Προφανώς, η *y* παίρνει ως τιμή, τελικώς, τη μέγιστη από τις τιμές των *a, b, c, d*.

Μπορείς να δώσεις στη *y* την ίδια τιμή με παράσταση υπό συνθήκη;

11-2 Ξαναλύσε την άσκ. 5-6 (*Μέρος Α*) χρησιμοποιώντας τη **switch**: "*Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 (θετικούς αριθμούς) και θα υπολογίζει και θα τυπώνει τη συνολική αντίσταση R. Πριν πάρει τις τιμές των αντιστάσεων, το πρόγραμμα θα ζητάει να διαβάσει, από το πληκτρολόγιο, τον τρόπο σύνδεσης των αντιστάσεων. Οι δεκτές απαντήσεις θα είναι:*

- `'P'` *ή* `'p'` *για παράλληλη σύνδεση,*
- `'S'` *ή* `'s'` *για σύνδεση εν σειρά.*

Να διατυπωθεί η προϋπόθεση και το πρόγραμμα να την ελέγχει."

[9] για γεύσεις και χρώματα δεν (πρέπει να) υπάρχει διένεξη.

11-3 Ξαναλύσε την άσκ. 6-15 (*Μέρος Α*) χρησιμοποιώντας τις εντολές που έμαθες σε αυτό το μάθημα: "*Θέλουμε ένα πρόγραμμα που θα παρακολουθεί έναν παίκτη του μπάσκετ κατά τη διάρκεια ενός παιγνιδιού. Το πρόγραμμα θα παίρνει τα εξής στοιχεία:*
`'1'` *κάθε φορά που ο παίκτης επιτυγχάνει καλάθι με ελεύθερη βολή,*
`'2'` *κάθε φορά που ο παίκτης επιτυγχάνει δίποντο,*
`'3'` *κάθε φορά που ο παίκτης επιτυγχάνει τρίποντο και*
`'4'` *κάθε φορά που ο παίκτης κάνει φάουλ.*
Πληκτρολογώντας `'0'` *δείχνουμε στο πρόγραμμα ότι τελείωσε το παιχνίδι. Όταν ο παίκτης συμπληρώσει πέντε φάουλ, θα πρέπει να βγαίνει το μήνυμα:* `"ΒΓΑΙΝΕΙ ΕΞΩ ΜΕ 5 ΦΑΟΥΛ"`.

Όταν τελειώσει η συμμετοχή του παίκτη –είτε λόγω αποβολής είτε λόγω τέλους του παιχνιδιού– θα γράφεται στην οθόνη η στατιστική του."

11-4 Στο *Μέρος Α* μάθαμε να διαβάζουμε τις τιμές των στοιχείων ενός πίνακα με *N* θέσεις, από ένα αρχείο μέσω του ρεύματος *t* με τις εξής εντολές:

```
m = 0;  t >> x[m];
while ( !t.eof() && m < N-1 )
{
   m = m + 1;  t >> x[m];
} // while
if ( t.eof() )  count = m;
          else  count = m + 1;
```

Τώρα που έμαθες όλες της δυνατότητες της **for**, ξαναγράψε τα παραπάνω με **for**.

11-5 Ξαναλύσε την άσκ. 9-1 χρησιμοποιώντας τις εντολές που έμαθες σε αυτό το μάθημα: "*Έστω ένας πίνακας*

```
double a[10];
```

Γράψε εντολές που θα τον «αναποδογυρίσουν». Δηλαδή να αντιμεταθέσουν τις τιμές των στοιχείων του:

(a[0] ↔ a[9]) (a[1] ↔ a[8]) (a[2] ↔ a[7]) (a[3] ↔ a[6]) (a[4] ↔ a[5])"

κεφάλαιο

12

Πίνακες ΙΙ – Βέλη

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να χρησιμοποιείς πολυδιάστατους (συνήθως δισδιάστατους) πίνακες και βέλη. Με τα βέλη μόλις αρχίζουμε...

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς πολυδιάστατους πίνακες στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *πολυδιάστατος πίνακας*
- *αποθήκευση στοιχείων πίνακα*
- *παράμετρος-πίνακας*
- *βέλος* (*pointer*)
- *αριθμητική βελών*
- *τυποθεώρηση* `const`
- *παράμετροι της* **main**

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Μέχρι τώρα, στο Μέρος Α, μάθαμε να χρησιμοποιούμε −και χρησιμοποιήσαμε− μονοδιάστατους πίνακες. Φυσικά, σε πολλές περιπτώσεις χρειαζόμαστε πολυδιάστατους πίνακες· συνήθως δισδιάστατους.

Δυνατότητα για χρήση τέτοιων πινάκων μας δίνουν όλες οι γλώσσες προγραμματισμού. Εδώ όμως η C++ (και η C) έχουν μια ιδιαιτερότητα: Για να μπορέσεις όμως να τους αξιοποιήσεις πλήρως θα πρέπει να ξέρεις τα «μυστικά» της υλοποίησής τους.

Το βασικό εργαλείο για τον σκοπό αυτόν είναι το **βέλος** (pointer). Θα αρχίσουμε λοιπόν να το μαθαίνουμε καλύτερα και −καλώς ή κακώς− θα το βρούμε μπροστά μας πολλές φορές στη συνέχεια,

12.1 Πίνακες και Βέλη

Στην §10.13 λέγαμε ότι αν ορίσεις:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ','π','ό','ν','τ','ο','ι','\0' };
```

και δώσεις:

```
cout << a << endl;
```

θα δεις στην οθόνη σου:

```
πάν μέτρον εκατό πόντοι
```

Και τι θα γίνει αν, ας πούμε, δηλώσεις:

```
double x[] = { 0.1, 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9 };
```

και δώσεις:

```
cout << x << endl;
```

Έκπληξη! Να το αποτέλεσμα:

```
0x22ff20
```

(ή κάτι παρόμοιο.)

Τι είναι αυτό; Αλλά, κάτι μας θυμίζει... Είχαμε δει κάτι τέτοιο στην §2.7.1. Λέγαμε ότι πρόκειται για μια ακέραιη τιμή γραμμένη στο δεκαεξαδικό σύστημα, μια *τιμή-βέλος*, που δείχνει μια *διεύθυνση* (θέση) της μνήμης. Λες να είναι το ίδιο πράγμα; Ας ζητήσουμε τη διεύθυνση του πρώτου στοιχείου (x[0]) του πίνακα· θα χρησιμοποιήσουμε τον τελεστή "`&`":

```
cout << &(x[0]) << endl;
```

Αποτέλεσμα:

```
0x22ff20
```

Η ίδια τιμή!

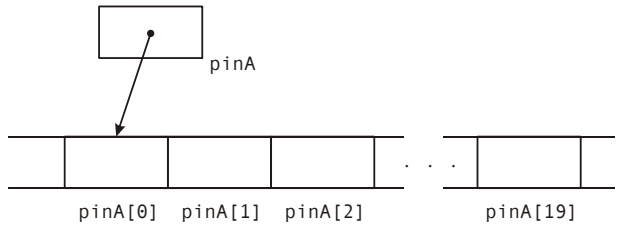
Κάτι παρόμοιο θα συμβεί όποιος και αν είναι ο τύπος (εκτός από `char`) των συνιστωσών του πίνακα.

Φυσικά δεν είναι τυχαίο. Για τη C++:

- ***Ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.***

Στο Σχ. 12-1 ξαναδίνουμε το Σχ. 9-1 αναθεωρημένο σε συμφωνία με τα παραπάνω. Όπως καταλαβαίνεις, αν κάνεις αποπαραπομπή στο όνομα του πίνακα, θα πρέπει να πάρεις το x[0]:

```
cout << (*x) << endl;
```



Σχ. 12-1 »Διόρθωση» του Σχ. 9-1. Η σχηματική παράσταση πίνακα και των στοιχείων του με το βέλος προς το πρώτο στοιχείο (`pinA[0]`) του πίνακα.

Αποτέλεσμα:

```
0.1
```

Τι διαφορά έχει το βέλος `x` από το `p` που δηλώνουμε ως:

```
double* p;
```

Το `x` είναι ένα *σταθερό βέλος* ενώ η `p` είναι μια *μεταβλητή-βέλος*. Αν έχουμε δηλώσει:

```
double r1, r2;
```

μπορούμε να γράψουμε:

```
p = &r1;  p = &r2;
```

ενώ απαγορεύεται να αλλάξουμε την τιμή του βέλους `x`.

Με βάση αυτά, δες πώς μπορεί να γραφεί η *vectorSum()* που πρωτοείδαμε στην §9.3:

```
double vectorSum( const double* x, int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Πρόσεξε ότι το μόνο που άλλαξε είναι η επικεφαλίδα, που ήταν:

```
double vectorSum( const double x[], int n, int from, int upto )
```

Στο σώμα της συνάρτησης δεν υπάρχει αλλαγή. Επίσης, δεν υπάρχει αλλαγή στη χρήση.

Παρατηρήσεις: ▶

1. Πάντως αξίζει να κάνουμε μερικές αλλαγές μέσα στο σώμα της συνάρτησης που δεν έχουν σχέση με το βέλος:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   double sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

2. Ο συμβολισμός με το βέλος δεν είναι κατ' ανάγκην προτιμότερος.

- Θα δεις στη συνέχεια ότι πολλές φορές θέλουμε να περάσουμε σε μια συνάρτηση (μέσω παραμέτρου) βέλος προς κάποιο αντικείμενο.
- Μέσα στη συνάρτηση δεν έχεις τρόπο να ξεχωρίσεις αν μια παράμετρος-βέλος δείχνει μια μεταβλητή ή έναν πίνακα.

Για λόγους τεκμηρίωσης λοιπόν, γενικώς,

♦ ***Όταν έχουμε παράμετρο-πίνακα θα προτιμούμε τον συμβολισμό με τις αγκύλες.***

και θα χρησιμοποιούμε παράμετρο-βέλος όταν περνούμε βέλος προς ένα αντικείμενο.

Ειδικώς για ορμαθούς χαρακτήρων της C –δηλαδή πίνακες χαρακτήρων με τον φρουρό `'\0'` στο τέλος– και οι δύο συμβολισμοί είναι καλοί αφού ναι μεν ο ορμαθός παριστάνεται με πίνακα αλλά θεωρείται ένα αντικείμενο.◀

Στην §9.3 γράφαμε ακόμη: «*Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων* [*ενός πίνακα*]*; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη vectorSum()* [*χωρίς παραμέτρους περιοχής επεξεργασίας*] *και για την περίπτωση αυτή.*» Τώρα μπορούμε να το δούμε αυτό. Η απλή *vectorSum()* είναι:

```
double vectorSum( const double* x, int n )
{
```

```
    double sum( 0 );

    for ( int m(0); m < n; ++m ) sum += x[m];
    return sum;
} // vectorSum
```

Για τον πίνακα που δηλώσαμε πιο πάνω, τα πέντε τελευταία στοιχεία είναι τα x[4], x[5], x[6], x[7], x[8]. Αφού η συνάρτηση περιμένει το πρώτο όρισμα να δείχνει την αρχή του πίνακα και το δεύτερο το πλήθος των στοιχείων του την καλούμε ως εξής:

```
cout << vectorSum( &x[4], 5 ) << endl;
```

Δηλαδή, «ξεγελάμε» τη συνάρτηση περνώντας της για αρχή του πίνακα το στοιχείο `x[4]`.

Προσοχή όμως: Αυτό είναι ένα *τέχνασμα* που στηρίζεται στο ότι ξέρουμε πώς γίνεται η υλοποίηση των εννοιών στη C++ (και στη C)[1]. Το να περνούμε τον πίνακα (`const double x[], int n`) και την περιοχή επεξεργασίας (`int from, int upto`) είναι *πάγια τεχνική*.

12.2 Για τον Περιορισμό "const"

Λέγαμε στην προηγούμενη παράγραφο, μετά τη δήλωση "`double x[] = {...}`": «*Το* `x` *είναι ένα σταθερό βέλος* [...] *απαγορεύεται να αλλάξουμε την τιμή του βέλους* `x`.» Πράγματι, αν έχεις δηλώσει:

```
double r1;
```

η εντολή "`x = &r1`" δεν θα γίνει δεκτή από τον μεταγλωττιστή.

Παρ' όλο που τα διαγνωστικά είναι διαφορετικά, αυτό μας θυμίζει την περίπτωση που δηλώναμε (§2.4):

```
const double g( 9.81 ); // m/sec²
```

και –στη συνέχεια– ο μεταγλωττιστής δεν μας επιτρέπει τη "`g = 5.32`".

Πάντως, αν ξεχάσουμε την υλοποίηση και το βέλος και δεχθούμε ότι τιμή ενός πίνακα είναι οι τιμές όλων των στοιχείων του, τότε πιο κοντά σε αυτήν τη χρήση του "`const`" είναι αυτό που μάθαμε στην §9.3, στην παράμετρο-πίνακα μιας συνάρτησης:

```
double vectorSum( const double x[], int n, int from, int upto )
```

την οποία μπορούμε να γράψουμε και ως: "`const double* x`".

Με αυτά που μαθαίνουμε τώρα, βλέπουμε ότι, όταν πρόκειται για βέλος, υπάρχει και άλλη δυνατότητα: να αλλάζει ή να μην αλλάζει η τιμή του βέλους (δηλαδή η διεύθυνση της μνήμης που δείχνει). Μπορείς να επιβάλεις τη *σταθερότητα του βέλους* με την εξής δήλωση:

```
double* const x( &r1 );
```

Τελος, για να έχουμε *σταθερό βέλος προς σταθερή τιμή* θα πρέπει να δηλώσουμε:

```
const double* const x( &g );
```

Τέτοιες δηλώσεις θα βρεις συνήθως στις τυπικές παραμέτρους συναρτήσεων. Εκεί φυσικά δεν βάζουμε αρχική τιμή· αυτή καθορίζεται αυτομάτως όταν καλείται η συνάρτηση και είναι η αντίστοιχη πραγματική παράμετρος.

12.2.1 Τυποθεώρηση "const"

Ας ξαναγυρίσουμε στην §9.5.1. Στη συνάρτηση *linSearch()* (με φρουρό) υποχρεωθήκαμε να αλλάξουμε το "`const int v[]`" σε "`int v[]`" διότι ο μεταγλωττιστής δεν δεχόταν τις εντολές:

```
        v[upto+1] = x;       // φρουρός
    // . . .
```

[1] ... και –καλώς ή κακώς– θα το δεις σε πολλά βιβλία, σε πολλούς διαδικτυακούς τόπους κλπ.

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Αν σκεφτούμε ότι στην πραγματικότητα "`const int v[]`" σημαίνει "`const int* v`" μπορούμε να κάνουμε το εξής: Ορίζουμε μια μεταβλητή

```
        int* ncv( v );
```

Η *ncv* είναι βέλος προς `int`, όπως και η *v*, αλλά δεν είναι `const`. Βάζοντας την *ncv* να δείχνει όπου και η *v* θα μπορούμε να τροποιούμε τα στοιχεία του πίνακα *v* μέσω αυτής. Αυτό όμως δεν μπορεί να γίνει ή τουλάχιστον δεν μπορεί να γίνει έτσι· δεν το επιτρέπει ο μεταγλωττιστής. Μπορεί να γίνει με τη λεγόμενη **τυποθεώρηση const**:

```
        int* ncv( const_cast<int*>(v) );
```

δηλαδή: το ncv δείχνει τον ίδιο πίνακα που δείχνει και το v, αλλά, αφού δεν έχει "`const`" (`const_cast<int*>`) μας δίνει συνατότητα τροποποίησης.

Να η νέα μορφή της *linSearch()*:

```
 0: int linSearch( const int v[], int n,
 1:                int from, int upto, int x )
 2: {
 3:    int fv( -1 );
 4:
 5:    if ( 0 <= from && from <= upto && upto < n )
 6:    {
 7:       int* ncv( const_cast<int*>(v) );
 8:       int save( v[upto+1] ); // φύλαξε το v[upto+1]
 9:       ncv[upto+1] = x;       // φρουρός
10:       int k( from );
11:       while ( v[k] != x )  ++k;
12:       if ( k <= upto )
13:          fv = k;
14:       ncv[upto+1] = save;    // όπως ήταν στην αρχή
15:    // (from <= fv <= upto && v[fv] == x) ||
16:    //       (fv == -1 && (για κάθε j:from..upto • v[j] != x))
17:    }
18:    return fv;
19: } // linSearch
```

Πρόσεξε τώρα τα εξής:

- Στην επικεφαλίδα (γρ. 0) *δεν αλλάξαμε* το "`int v[]`" σε "`int* v`". Όπως είπαμε, το πρώτο είναι προτιμότερο μια και δείχνει πίνακα.
- Μεταφέραμε τις δηλώσεις των *save* (γρ. 8) και *k* (γρ. 10) εκεί που μπορούμε να ορίσουμε και την αρχική τους τιμή.
- Στη γρ. 7 βάλαμε τη δήλωση του *ncv* με την τυποθεώρηση `const`.
- Έτσι, όταν στη γρ. 9 και στη γρ. 14 αλλάζουμε την τιμή του ncv`[`*upto*+1`]` αλλάζουμε στην πραγματικότητα την τιμή του v`[`*upto*+1`]`.

Με την τυποθεώρηση **const** μπορείς να αφαιρείς ή να προσθέτεις τον περιορισμό "`const`" (ή τον "`volatile`"). Για παράδειγμα, ας πούμε ότι θέλουμε να έχουμε έναν πίνακα που οι τιμές των στοιχείων του δεν θα αλλάζουν κατά τη διάρκεια εκτέλεσης του προγράμματος. Αλλά, αυτές οι τιμές δεν είναι σταθερές γνωστές όταν γράφουμε το πρόγραμμα. Θα πρέπει κάθε φορά που εκτελείται να διαβάζονται από κάποιο αρχείο. Μπορούμε να κάνουμε το εξής:

```
   double xf[100];
// ανάγνωση τιμών του xf από το αρχείο
   const double* x( const_cast<const double*>(xf) );
```

Τώρα, μέσω του x, μπορούμε να χρησιμοποιούμε τα στοιχεία χωρίς να υπάρχει περίπτωση να αλλάξουμε κάποια τιμή κατά λάθος. Δυστυχώς όμως και το xf υπάρχει και η ζημιά μπορεί να γίνει από εκεί.

Να ένας άλλος τρόπος πιο ασφαλής:

```
   const double x[100] = { 0 };
   double* xf( const_cast<double*>(x) );
```

```
// ανάγνωση τιμών του xf από το αρχείο
   xf = 0;
```

Εδώ χρησιμοποιούμε το βέλος xf για να δώσουμε τιμές στα στοιχεία του x, που είναι δηλωμένος `const`. Μετά από αυτό, του βάζουμε τιμή 0 και τον «αποσυνδέουμε» από τον πίνακα x. Για την τιμή βέλους 0 τα λέμε παρακάτω.

Αργότερα θα δούμε παράδειγμα τυποθεώρησης **const** και σε παράμετρο αναφοράς

Τα ίδια μπορείς να κάνεις και με το "`volatile`" αλλά κάτι τέτοιο δεν είναι και τόσο χρήσιμο.

12.3 Πράξεις με Βέλη

Τώρα μπορούμε να δούμε και κάτι άλλο: Με τα βέλη προς στοιχεία πίνακα μπορείς να κάνεις και πράξεις, για την ακρίβεια πρόσθεση και αφαίρεση. Έτσι, έχουν νόημα τα `x+1`, `x+2`,... που μας δίνουν τις θέσεις των στοιχείων `x[1]`, `x[2]`,... Με αποπαραπομπή (`*(x+1)`, `*(x+2)`,...) παίρνουμε τα στοιχεία:

```
   cout << (*x) << "  " << (*(x+1)) << "  " << (*(x+2)) << endl;
```

Αποτέλεσμα:

```
0.1  1.2  2.3
```

Εδώ πρόσεξε το εξής: Οι παρενθέσεις *είναι απαραίτητες* διότι *η πράξη της αποπαραπομπής έχει μεγαλύτερη προτεραιότητα από την πρόσθεση*. Πράγματι από την

```
   cout << (*x+1) << "  " << (*(x+1)) << endl;
```

θα πάρουμε:

```
1.1  1.2
```

Το `1.1` είναι στην πραγματικότητα το αποτέλεσμα της πράξης: `*x+1` που δεν είναι τίποτε άλλο από το `x[0]+1`.

Να λοιπόν η σχέση μεταξύ *δείκτη στοιχείου* και *βέλους προς στοιχείο* πίνακα: Αν

```
T x[N];
```

και *k* φυσικός από 0 μέχρι *N*-1, τότε:

```
x[k] == *(x + k)
```

Πρόσεξε το εξής: το `*(x+k)` δεν είναι η τιμή του στοιχείου αλλά το ίδιο το στοιχείο (δηλαδή μια *τιμή-l*). Αυτό σημαίνει ότι μπορείς να δώσεις:

```
   *(x+3) = 7.77;
```

για να αλλάξεις την τιμή του `x[3]`.

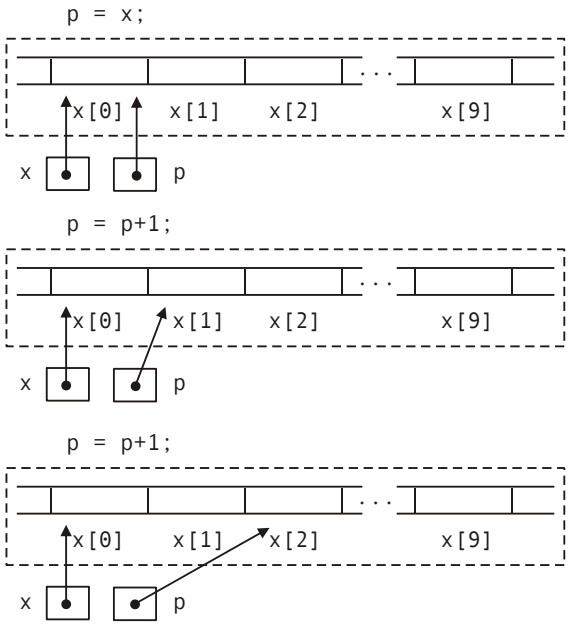
Είδαμε πιο πριν ότι μπορούμε να υπολογίσουμε το άθροισμα των στοιχείων ενός πίνακα με τις:

```
      double sum( 0 );
      for ( int m(0); m < N; ++m ) sum += x[m]; // άθροισμα
```

Μπορούμε να κάνουμε τον ίδιο υπολογισμό με βέλη. Ας πούμε ότι έχουμε μια μεταβλητή-βέλος *p* προς μεταβλητή τύπου `double`. Μπορούμε να δώσουμε τιμή στην *p* με την εντολή: `p = x`. Με την `*p` παίρνουμε την τιμή του x[0]. Αυξάνοντας την τιμή της *p* κατά 1 (`p=p+1` ή `++p`) βάζουμε την *p* να δείχνει το x[1] και με αποπαραπομπή (`*p`) παίρνουμε την τιμή του. Δες το Σχ. 12-2. Για να πάρουμε λοιπόν το άθροισμα μπορούμε να γράψουμε:

```
      double sum( 0 );
      for ( double* p(x); p != &x[N]; ++p )  sum += *p;
```

Δηλαδή: Θέλω μια μεταβλητή *p* τέτοια ώστε το `*p` να είναι τύπου `double`. Αρχικώς θα δείχνει το στοιχείο `x[0]` του πίνακα x (`p(x)`). Στη συνέχεια θα πηγαίνει στο επόμενο στοιχείο (`++p`) και θα σταματήσει όταν βρεθεί να δείχνει την πρώτη θέση μετά το τελευταίο στοιχείο του πίνακα (`&x[N]`). Αυτή η δήλωση και η τελευταία **for** εξετάζονται πιο εκτεταμένα στην επόμενη υποπαράγραφο.



Σχ. 12-2 Ο πίνακας και τα βέλη. Τιμή του πίνακα (`x`) είναι η διεύθυνση του πρώτου στοιχείου (x[0]). Εδώ, βάζουμε ένα βέλος που δείχνει το x[0], αντί να βάλουμε ως τιμή της x την `0x0064fdbc` (ή κάτι παρόμοιο),. Μετά την εκτέλεση της εκχώρησης "`p = x`" την ίδια τιμή έχει και η *p*. Όταν αυξήσουμε την τιμή της *p* κατά 1 (`p=p+1`) η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[1]). Αν αυξήσουμε ξανά την τιμή της *p* κατά 1 η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[2]) κ.ο.κ..

Πολλά πράγματα μαζί! Ας τα δούμε ένα-ένα.

12.3.1 Μια Θέση Μετά το Τέλος

Πριν από οποιαδήποτε πράξη να πούμε το εξής: Στο παραπάνω παράδειγμα εμφανίστηκε ένα "`&x[N]`". Τι είναι αυτό; Τα στοιχεία του πίνακα καταλαμβάνουν τις θέσεις από 0 μέχρι *N*-1. Σωστό! Αλλά η C++ σου επιτρέπει να έχεις βέλος που να δείχνει στο `x[N]` (past the end pointer), αρκεί:

- Να μήν προσπαθήσεις να κάνεις αποπαραπομπή.
- Να το χρησιμοποιείς μόνο σε συγκρίσεις με άλλα βέλη (π.χ.: "`p != &x[N]`")

12.3.2 Αρχική Τιμή και Εκχώρηση

Ας ξεκινήσουμε από τον ορισμό αρχικής τιμής "`double* p(x)`". Σε μια μεταβλητή-βέλος μπορείς να δίνεις ως αρχική τιμή την τιμή μιας παράστασης-βέλος του ίδιου τύπου. Από όσα ξέρουμε μέχρι τώρα μπορείς να δώσεις:

- μια άλλη μεταβλητή-βέλος ίδιου τύπου· αν *p*1 τύπου `double*`: "`double* p(p1)`", οπότε το *p* δείχνει την ίδια θέση της μνήμης που δείχνει και το *p*1 (*p* == *p*1 ή `*`*p* και `*`*p*1 είναι το ίδιο αντικείμενο)
- το όνομα ενός πίνακα, όπως εδώ: "`double* p(x)` ", οπότε το *p* δείχνει το πρώτο στοιχείο του πίνακα (*p* == &x[0])
- τη διεύθυνση μιας μεταβλητής, π.χ.: "`double* p(&sum)`", οπότε το *p* δείχνει τη μεταβλητή *sum* (*p* == &*sum* ή `*`*p* είναι η *sum*).

  Όπως θα δούμε στη συνέχεια, μπορείς ακόμη να δώσεις:
- τιμή μηδέν (0): "`double* p( 0 )`",

- την τιμή αριθμητικής πράξης με βέλη, π.χ.: "`double* p(x+1)`".

Προσοχή! ▶

Σε ένα βέλος τύπου `const T*` μπορείς να βάλεις ως αρχική τιμή βέλος τύπου `T*`, π.χ.:

```
int a[17];
const int* pa( a );
```

Το αντίθετο δεν επιτρέπεται. Μπορείς να το πετύχεις με την κατάλληλη τυποθεώρηση, π.χ.:

```
unsigned int myStrLen( const char* cs )
{
   char* p( const_cast<char*>(cs) );
// . . .
```

(από ένα παράδειγμα που θα δεις στη συνέχεια.) ◀

Όταν δηλώνεις ένα βέλος, χωρίς να του δώσεις αρχική τιμή αυτό δείχνει σε κάποια τυχαία θέση μέσα στη μνήμη. Αν προσπαθήσεις να το χρησιμοποιήσεις μπορεί να κάνεις και κάποια ζημιά. Γι' αυτό οι προγραμματιστές συνηθίζουν να δίνουν αρχική τιμή στα βέλη, με τη δήλωσή τους. Αν δεν ξέρουν τι τιμή να δώσουν δίνουν την τιμή μηδέν ("`0`")[2] που:

- είναι συμβατή με οποιονδήποτε τύπο-βέλους,
- σημαίνει (κατά κοινή συμφωνία): *αυτό το βέλος δεν δείχνει κάτι* (που μας ενδιαφέρει) και
- είναι τιμή που μπορεί να ελεγχθεί (`if (p == 0)...`), όπως θα δούμε στη συνέχεια.

Με τους ίδιους τρόπους μπορείς να αλλάζεις, στη συνέχεια, την τιμή ενός βέλους με μια εντολή εκχώρησης. Στο δεξιό μερος μπορεί να υπάρχει "`0`" ή παράσταση που θα μας δώσει τιμή-βέλος ίδιου τύπου. Για παράδειγμα:

```
p = p1;
p = x;
p = &sum;
p = 0;
p = x + 1;
```

Θυμίσου, για παράδειγμα, ότι στην §12.2, αποσυνδέσαμε το βέλος *xf* από τον πίνακα x βάζοντας "`xf = 0`".

Για τον περιορισμό "`const`", ισχύει και για την εκχώρηση ο περιορισμός που είπαμε πιο πάνω για την απόδοση αρχικής τιμής.

Ένα σημείο που χρειάζεται προσοχή είναι το εξής: Ας πούμε ότι έχουμε δηλώσει:

```
int a( 10 ), b( 20 );
int* pa;
int* pb;
```

και στη συνέχεια δίνουμε:

```
pa = &a;  pb = &b;
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 12-3(α): το βέλος *pa* δείχνει την *a*, που έχει τιμή 10, και το βέλος *pb* δείχνει την *b* που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή "`a = b`". Η εικόνα της μνήμης είναι αυτή του Σχ. 12-3 (β). Η τιμή της *a* γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

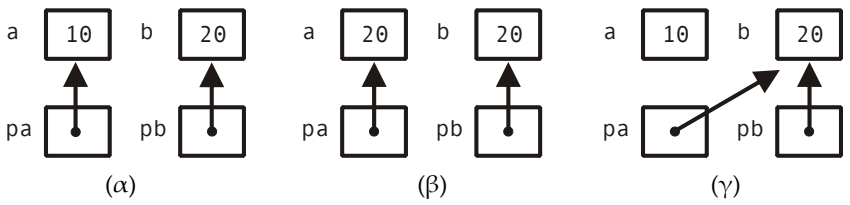
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: "`pa = pb`". Οι τιμές των μεταβλητών *a* και *b* δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *pa*: το βέλος *pa* δείχνει εκεί που δείχνει και το *pb*: τη *b* (Σχ. 12-3 (γ)).

Φυσικά, και στις δύο περιπτώσεις, αν δώσουμε την εντολή:

```
cout << *pa << "  " << *pb << endl;
```

---

[2] Οι προερχόμενοι από τη C αντί "`0`" (μηδέν) βάζουν τη σταθερά "`NULL`" που ορίζεται με τη μάκρο "`#define NULL 0`". Ο B. Stroustrup συμβουλεύει να προτιμούμε το "`0`". Πάντως στη νέα τυποποίηση C++11 εισάγεται το όνομα (*std*`::`)"*nullptr*".



Σχ. 12-3 Αρχικώς, μετά την εκτέλεση των εντολών "`pa = &a; pb = &b;`" έχουμε την εικόνα (α). Η εικόνα (β) προκύπτει αν στη συνέχεια εκτελεσθεί η εκχώρηση: "`a = b`". Η εικόνα (γ) προκύπτει αν, μετά την (α), εκτελεσθεί η εκχώρηση "`pa = pb`".

θα πάρουμε αποτέλεσμα:

```
20  20
```

Υπάρχει και ένας περιορισμός σχετικά με την εκχώρηση: *η μεταβλητή και η παράσταση θα πρέπει να είναι του ίδιου τύπου*. Έτσι, αν έχουμε δηλώσει:

```
double v;
```

δεν επιτρέπεται να γράψουμε[3]: "`pa = &v`".

Προσοχή! ▶

Πριν προχωρήσουμε να επισημάνουμε κάτι που μπορεί να σε οδηγήσει σε «παράξενα» λάθη μεταγλώττισης. Αν θελήσεις να μαζέψεις τις δύο δηλώσεις:

```
int* pa;
int* pb;
```

σε μία, θα πρέπει να γράψεις:

```
int *pa, *pb;
```

Αν γράψεις κατά λάθος:

```
int* pa, pb;
```

θα σημαίνει ότι η *pa* είναι βέλος προς `int` αλλά η *pb* είναι `int`! ◀

12.3.3 Πρόσθεση και Αφαίρεση

Αν έχεις δηλώσει:

```
const unsigned int N = ...;
T x[N];
```

ορίζονται οι πράξεις: `+`: *T*`*`×`int` ↠ *T*`*` και `-`: *T*`*`×`int` ↠ *T*`*` ή, ακριβέστερα:

`+`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`+`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

Πιό πάνω είδαμε ότι αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] `..` x[*N*`-1`], τότε *το* `p+1` *δείχνει το επόμενο στοιχείο*. Αντιστοίχως, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[1] .. x[*N*], τότε *το "*`p-1`*" δείχνει το προηγούμενο στοιχείο*.

Γενικώς, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] .. x[*N*`-1`], τότε

- *Το "*`p + k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***μετά*** *το στοιχείο που δείχνει το p*.
- *Το "*`p - k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***πριν*** *το στοιχείο που δείχνει το p*.

[3] Αν επιμένεις πολύ γίνεται, με την κατάλληλη τυποθεώρηση· αλλά, πού θα σου χρησιμεύσει κάτι τέτοιο (εκτός από το "σκαλίζεις" τη μνήμη);

Προσοχή! ▶

1. Δεν ειναι τυχαίες οι συνεχείς αναφορές σε πίνακες. *Οι πράξεις αυτές ορίζονται μόνο για βέλη προς στοιχεία ενός πίνακα.*

2. Παρ' όλο που λέμε ότι «τιμή του βέλους *p* είναι μια διεύθυνση της μνήμης» *το* "`p + 1`" ***δεν*** *είναι η επόμενη διεύθυνση* (π.χ. η διεύθυνση της επόμενης ψηφιολέξης). Είναι η διεύθυνση του επόμενου στοιχείου του πίνακα.◀

Γιατί γράψαμε τις συναρτήσεις μας μερικές; Διότι, για παράδειγμα, αν το *p* δείχνει το x[2] απαγορεύονται πράξεις σαν τις `p-6` ή `p+(-5)`.

- ***Είναι υποχρέωση του προγραμματιστή να φροντίσει ώστε το αποτέλεσμα πράξης βελών να είναι βέλος προς στοιχείο πίνακα ή προς την πρώτη θέση μετά το τελευταίο στοιχείο.***

Οι συντομογραφίες "`++`", "`--`", "`+=`", "`-=`" ισχύουν και έχουν το νόημα που ξέρεις προσαρμοσμένο στα παραπάνω. Έτσι, αν το *p* δείχνει το x[2] το "`++p`" αλλάζει την τιμή του *p* ώστε να δείχνει το x[3] (ενώ αντιθέτως το "`--p`" θα το πήγαινε στο x[1]). Αν βάλεις "`p += 2`", το *p* θα πάει από το x[2] στο x[4].

Υπάρχει μια ακόμη συνάρτηση (πράξη)· είναι μεταξύ βελών και είναι ολική:

`-`: [`&x[0] .. &x[`*N*`]`] × [`&x[0] .. &x[`*N*`]`] → [`-`*N* `..` *N*]

Αν δύο βέλη, `p1`, `p2`, δείχνουν στοιχεία του ίδιου πίνακα, ας πούμε τα `x[k1]` και `x[k2]`, έχει νόημα η διαφορά "`p1-p2`", που δείχνει τον αριθμό των στοιχείων που πρέπει να διανύσουμε για να φτάσουμε από το ένα στοιχείο στο άλλο· στην περίπτωσή μας "`k1-k2`".

Ας δούμε δύο παραδείγματα που χρησιμοποιούν πράξεις μεταξύ βελών.

Παράδειγμα 1 – Μήκος ορμαθού C

Όπως μάθαμε στο Κεφ. 10 (§10.13), στη C παριστάνουμε τα κείμενα σε πίνακες με στοιχεία τύπου `char` που έχουν στο τέλος (ως φρουρό) τον χαρακτήρα `'\0'`. Μια από τις συναρτήσεις που μας δίνει η C για τον χειρισμό τέτοιων πινάκων είναι η *strlen()* που μας δίνει το μήκος του κειμένου. Εδώ θα γράψουμε μια τέτοια συνάρτηση, ας την πούμε *myStrLen()*.

Αφού παρατηρήσουμε ότι αν *l* το μήκος ενός κειμένου αποθηκευμένου στον πίνακα cs τοτε ο φρουρός βρίσκεται στο στοιχείο cs`[`*l*`]`, γράφουμε τη συνάρτησή μας ως εξής:

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l] != '\0' )  ++l;
   return l;
} // myStrLen
```

Γράφουμε τώρα το ίδιο πράγμα με βέλη, για να πάρουμε μια πιο γρήγορη συνάρτηση:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *p != '\0' )  ++p;
   return p-cs;
} // myStrLen
```

Το *p* θα διατρέξει ολόκληρον τον πίνακα ξεκινώντας από το `cs[0]` (διεύθυνση `&cs[0]` ή απλώς `cs`). Ναι, αλλά πώς θα γίνει αυτό; Έχουμε βάλει "`const char* p(cs)`"! Αν έχεις αυτήν την απορία γύρισε πίσω, στην §12.2. Αυτό που λέμε εδώ είναι ότι *δεν επιτρέπεται μεταβολή αυτού που δείχνει το βέλος· μεταβολή του βέλους επιτρέπεται.*

Αν όμως σε ενοχλεί το "`const`" πρόσεξε τα εξής: Ο μεταγλωττιστής δεν θα επιτρέψει να γράψουμε `char* p(cs)` αφού το cs είναι "`const char`". Η τυποθεώρηση **const** είναι απαραίτητη:

```
   char* p( const_cast<char*>(cs) );
```

Ο *p* προχωρεί με τη `++p` όσο αυτό που δείχνει δεν έχει τον φρουρό (`*p != '\0'`).

Τελικώς, η συνάρτηση επιστρέφει τη διαφορά του βέλους προς τον φρουρό από το βέλος προς την αρχή.

Ένας προγραμματιστής C θα απορούσε ήδη με το ότι στην πρώτη μορφή επιμένουμε να γράφουμε πολλά αντί να χρησιμοποιήσουμε τον μεταθεματικό "`++`":

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l++] != '\0' );
   return l-1;
} // myStrLen
```

Πρόσεξε ότι εδώ παίρνουμε την τιμή του *l* για να υπολογίσουμε το στοιχείο cs`[`*l*`]` και αυξάνουμε την τιμή του *l*. Έτσι, αφ'ενός δεν έχουμε (άλλη) επαναλαμβανόμενη εντολή στη **while**, αφ' ετέρου η τιμή του *l* θα αυξηθεί και στην τελευταία επανάληψη, όταν βρούμε τον φρουρό. Γι' αυτό και επιστρέφουμε *l*-1.

Να το γράψουμε με βέλη; Νάτο:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) != '\0' );
   return p-cs-1;
} // myStrLen
```

Πρόσεξε ότι χρησιμοποιούμε μεταθεματικό "`++`" στο βέλος. Φυσικά, θα πρέπει και εδώ να αφαιρέσουμε 1.

Και βέβαια, ένας προγραμματιστής C δεν θα έλεγχε ποτέ για μηδέν με αυτόν τον τρόπο· θα το έγραφε πολύ πιο απλά:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) );
   return p-cs-1;
} // myStrLen
```

🖘🖘🖘

Παράδειγμα 2 – Επισύναψη ορμαθού C 🖙

Στην §10.13 λέγαμε και για τη *strcat*: «*Η* `strcat(a, b)` *κάνει το ίδιο πράγμα με την* `a.append(b)`*: επισυνάπτει στο τέλος της τιμής του a την τιμή του b.*» Η επικεφαλίδα της είναι:

```
char* strcat( char* dest, const char* src )
```

Η *strcat*, αφού επισυνάψει στο τέλος του *dest* ένα αντίγραφο του *src*, επιστρέφει ως τιμή βέλος προς την αρχή του *dest*.

Ας υλοποιήσουμε μια τέτοια *myStrCat* με πίνακες και δείκτες· χωρίς βέλη:

```
char* myStrCat( char* dest, const char* src )
{
   int j( 0 );
   while ( dest[j] != '\0' ) ++j;
   int k(0);  // εδώ έχουμε: dest[j] == '\0'
   while ( src[k] != '\0' )
   {
      dest[j] = src[k];
      ++j;  ++k;
   } // for
   dest[j] = '\0';
   return dest;
} // myStrCat
```

Πρόσεξε ότι δεν έχουμε βάλει οποιονδήποτε έλεγχο. Σκέψου, ως άσκηση, τους ελέγχους που θα πρέπει να βάλουμε.

Η ίδια συνάρτηση με βέλη μπορεί να γραφεί ως εξής:

```
char* myStrCat( char* dest, const char* src )
{
   char* pd( dest );
   while ( *(pd++) != '\0' );
   --pd;       // εδώ έχουμε το pd να δείχνει τον φρουρό
   const char* ps( src );
   while ( *ps != '\0' )
   {
      *pd = *ps;
      ++pd;  ++ps;
   }
   *pd = '\0';
   return dest;
} // myStrCat
```

Γιατί "`--pd`"; Ξαναδιάβασε το προηγούμενο παράδειγμα...

12.3.4 Ο Τύπος "ptrdiff_t"

Στο Παράδ. 1 της προηγούμενης παραγράφου, σε συμμόρφωση με τη φιλοσοφία της C++, βάλαμε ως τύπο επιστροφής "`size_t`".

Εδώ θα πρέπει να αναφέρουμε ότι το αποτέλεσμα της πράξης

`-`: [`&x[0] .. &x[`N`]`] × [`&x[0] .. &x[`N`]`] → [`-`N `..` N]

είναι τύπου "`std::ptrdiff_t`" (*poin*ter *diff*erence) . Αν ψάξεις στο `cstddef` (ή το `stddef.h`) θα βρεις:

```
typedef int     ptrdiff_t;
```

ή κάτι παρόμοιο.

Όταν γράφουμε "`return p-cs`" ζητούμε να επιστραφεί μια τιμή τύπου `ptrdiff_t`. Επειδή όμως αυτή είναι σίγουρα μη αρνητική μετατρέπεται σε τύπο `size_t`.

12.3.5 Συγκρίσεις

Ένα άλλο πράγμα που μπορείς να κάνεις με τα βέλη είναι η σύγκριση για ισότητα ("`==`") ή για ανισότητα ("`!=`"). Έτσι, βλέπεις στη **for** της άθροισης τη σύγκριση: "`p != x+N`" ή "`p != &x[N]`".

Μπορείς να συγκρίνεις δύο βέλη του ίδιου τύπου. Αργότερα θα δούμε ότι μερικές φορές χρειάζεται να συγκρίνουμε και βέλη διαφορετικού τύπου. Αυτό γίνεται με την κατάλληλη τυποθεώρηση.

Παράδειγμα

Στη *vectorSum* δεν έχουμε βάλει ελέγχους. Έτσι, μπορεί να κληθεί με $n \le 0$, με *from* ή/και *upto* έξω από την περιοχή 0..n-1. Αυτά σου τα αφήνουμε ως άσκηση.

Εδώ θα δούμε ένα άλλο πρόβλημα: Όπως καταλαβαίνεις, η *vectorSum* μπορεί να κληθεί με πρώτη παράμετρο 0 (μηδέν), δηλαδή *χωρίς πίνακα*. Και αν μεν έχουμε και n `==` 0 μπορούμε να βγάλουμε μηδενικό άθροισμα. Αν όμως έχουμε n `>` 0 τότε έχουμε προφανώς λάθος. Να λοιπόν η συνάρτηση με αυτόν τον έλεγχο:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   if ( x == 0 && n > 0 )
   {  cerr << "η vectorSum κλήθηκε με ανύπαρκτο πίνακα" << endl;
      exit( EXIT_FAILURE );  }
// άλλοι έλεγχοι
   double sum( 0 );
   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

12.4 Πολυδιάστατοι Πίνακες

Ας πούμε ότι έχουμε ορίσει:

```
typedef int   intArr [ 51 ];
```

και δηλώνουμε:

```
   intArr    mat [ 11 ];
```

Τι είναι μια συνιστώσα του mat, π.χ. η `mat[7]`; Ένας *πίνακας* με 51 συνιστώσες τύπου `int`. Πώς θα αναφερθούμε στην 23η συνιστώσα αυτού του πίνακα; Αν θεωρήσουμε ότι έχουμε έναν μονοδιάστατο πίνακα με όνομα `mat[7]` το 23ο στοιχείο του θα είναι: `(mat[7])[23]`. Η C++ μας επιτρέπει να γράψουμε απλούστερα: `mat[7][23]`.

Η δήλωση του mat θα μπορούσε να δοθεί ισοδυνάμως και ως εξής:

```
   int   mat[ 11 ][ 51 ];
```

Ο mat είναι ένας **πίνακας δυο διαστάσεων** (two-dimensional array) και έχει 11×51 στοιχεία τύπου `int`.

Όταν δηλώνουμε έναν πίνακα, μονοδιάστατο ή πολυδιάστατο, ο τύπος συνιστωσών μπορεί να είναι οποιοσδήποτε. Έτσι, οι πίνακες pinA, pinB, pinC και pinD, που δηλώνονται στο παρακάτω παράδειγμα, είναι πίνακες μιας διάστασης, δύο, τριών και τεσσάρων διαστάσεων αντίστοιχα.

```
   int    pinA[ 10 ];
   char   pinB[ 10 ][ 30 ];
   double pinC[ 5 ][ 10 ][ 20 ];
   bool   pinD[ 5 ][ 10 ][ 20 ][ 8 ];
```

- Ο πίνακας pinA αποτελείται από 10 στοιχεία τύπου `int`,
- ο πίνακας pinB από 300 (= 10×30) στοιχεία τύπου `char`,
- ο πίνακας pinC από 1000 (= 5×10×20) στοιχεία τύπου `double` και τέλος
- ο πίνακας pinD από 8000 (= 5×10×20×8) στοιχεία τύπου `bool`.

Να μερικά παραδείγματα γραφής στοιχείων πολυδιάστατων πινάκων, σύμφωνα με τις παραπάνω δηλώσεις:

```
α) pinB[1][1], pinB[1][29], pinB[2][1], pinB[2][29],
   pinB[9][1], pinB[9][29]
β) pinC[1][1][1], pinC[5][1][1], pinC[1][9][1],
   pinC[1][1][19], pinC[5][9][19]
γ) pinD[1][1][1][1], pinD[1][1][1][2], pinD[4][9][19][7],
   pinD[4][9][19][8]
```

Με τη δήλωση μπορείς να δώσεις και αρχική τιμή. Αφού, όπως είπαμε, ένας δισδιάστατος πίνακας είναι (μονοδιάστατος) πίνακας μονοδιάστατων πινάκων θα πρέπει να μπορούμε να δώσουμε κάτι σαν:

```
   double x[4][2] = { {0.1, 1.2}, {2.3, 3.4}, {4.5, 5.6},
                      {6.7, 7.8} };
```

Και πράγματι, αυτό είναι σωστό! Και ποια είναι τα στοιχεία αυτού του μονοδιάστατου πίνακα; Τα `x[0]`,… `x[3]`. Οι εντολές:

```
   for ( int r(0); r < 4; ++r )
      cout << x[r] << "  " << &x[r][0] << endl;
```

θα μας δώσουν:

```
0x22ff30  0x22ff30
0x22ff40  0x22ff40
0x22ff50  0x22ff50
0x22ff60  0x22ff60
```

Δηλαδή, το `x[r]` είναι βέλος προς το `x[r][0]`.

Στη C++ τα στοιχεία ενός δισδιάστατου πίνακα *αποθηκεύονται κατά γραμμές*: πρώτα τα στοιχεία της γραμμής 0, στη συνέχεια τα στοιχεία της γραμμής 1 κλπ. Αυτό όμως δεν σημαίνει ότι έχουμε την υποχρέωση να επεξεργαζόμαστε τα στοιχεία με κάποια συγκεκριμένη σειρά. *Δεν υπάρχει κανένας περιορισμός σχετικά με τη σειρά επεξεργασίας των στοιχείων του πίνακα. Η σειρά αυτή καθορίζεται μόνο από τις ανάγκες και τη λογική του προγράμματος και επιλέγεται ελεύθερα από τον προγραμματιστή.*

Ας δούμε μερικά παραδείγματα:

Παράδειγμα 1 - Άθροισμα στοιχείων τρισδιάστατου πίνακα

Αν έχουμε δηλώσει:

```
double pinC[ 5 ][ 10 ][ 20 ];
```

οι τρεις παρακάτω φωλιασμένες **for** υπολογίζουν το άθροισμα, *sum*, των στοιχείων του pinC:

```
sum = 0;
for ( int k(0); k < 5; ++k ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int m(0); m < 20; ++m ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int m . . .
   } // for ( int l . . .
} // for ( int k . . .
```

Όπως καταλαβαίνεις, ο τρίτος δείκτης, δηλαδή ο *m*, είναι ο δείκτης που αλλάζει τιμή πιο συχνά από τους άλλους και αυτό επειδή βρίσκεται στην πιο εσωτερική **for**. Ο δείκτης *m* διατρέχει όλες τις τιμές του, δηλ.τις τιμές 0 μέχρι 19, για κάθε νέο συνδυασμό τιμών των δεικτών *k* και *l* (δηλ. για 50 συνδυασμούς). Έτσι, η εσωτερική **for** εκτελείται 1000 (=20×50) φορές. Αντίθετα, ο δείκτης *k* είναι ο δείκτης που *μόνο μια φορά* διατρέχει τις τιμές του (από 0 ως 4), ενώ ο δείκτης *l* διατρέχει 5 φορές τις τιμές του (από 0 μέχρι 9). Έτσι, η ενδιάμεση **for** εκτελείται 5 φορές.

Θα μπορούσαμε να υπολογίσουμε το άθροισμα και με τις:

```
sum = 0;
for ( int m(0); m < 20; ++m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(0); k < 5; ++k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
} // for ( int m . . .
```

Στη περίπτωση αυτή, ο πρώτος δείκτης, *k*, είναι αυτός που αλλάζει τιμή πιο συχνά και διατρέχει 200 φορές τις τιμές του, δηλ. τις τιμές 0 μέχρι 4. Ο δείκτης *m* διατρέχει μόνο μιά φορά τις τιμές του (0 ως 19) και ο *l* διατρέχει 20 φορές τις τιμές του (από 0 μέχρι 9).

Τέλος, δες και μια τρίτη περίπτωση:

```
sum = 0;
for ( int m(19); m >= 0; --m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(9); l >= 0; --l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(4); k >= 0; --k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
```

```
   } // for ( int m . . .
```

Στην περίπτωση αυτήν έχουμε το ίδιο φώλιασμα με τη δεύτερη, αλλά οι δείκτες διατρέχουν τις τιμές από το τέλος προς την αρχή.

Παράδειγμα 2 - Ανάγνωση τιμών στοιχείων δισδιάστατου πίνακα

Ας δούμε όμως και το εξής πρόβλημα: Σε ένα αρχείο, που το διαβάζουμε μέσω του ρεύματος *ins*, έχουμε τις τιμές των στοιχείων ενός δισδιάστατου πίνακα:

```
   double   a[ 5 ][ 7 ];
```

Οι τιμές είναι γραμμένες *κατά γραμμές*. Τι θα πει αυτό; Σε κάθε γραμμή του αρχείου υπάρχουν οι (επτά) τιμές μιας γραμμής του πίνακα. Στην 1η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 0 του πίνακα, στη 2η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 1 του πίνακα κ.ο.κ. Πώς διαβάζουμε τις τιμές των στοιχείων του πίνακα;

```
   for ( int r(0); r < 5; ++r )
   {
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][c];
      } // for ( int c . . .
   } // for ( int r . . .
```

Όπως καταλαβαίνεις για κάθε τιμή της *r* διαβάζουμε τις τιμές των στοιχείων της γραμμής *r* του πίνακα. Αυτό γίνεται με την:

```
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][s];
      } // for ( int s . . .
```

και φυσικά εδώ *δεν μπορούμε να αλλάξουμε τη σειρά* (φωλιάσματος) των δύο **for**.

Βέβαια, υπάρχει και η περίπτωση να δοθούν οι τιμές των στοιχείων *κατά στήλες*. Δηλαδή, σε κάθε γραμμή του αρχείου υπάρχουν οι (πέντε) τιμές μιας στήλης του πίνακα Στην περίπτωση αυτή είμαστε υποχρεωμένοι να διαβάσουμε ως εξής:

```
   for ( int c(0); c < 7; ++c )
   {
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
   } // for ( int c . . .
```

Τώρα, η

```
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
```

διαβάζει τη στήλη *c* του πίνακα.

Παρομοίως γίνεται και το γράψιμο των στοιχείων δισδιάστατου πίνακα κατά γραμμές ή κατά στήλες (άσκ. 8-4).

Παράδειγμα 3 - Συμμετρικότητα τετραγωνικού πίνακα

Ένας τετραγωνικός πίνακας –δηλαδή: δισδιάστατος πίνακας με ίσους αριθμούς γραμμών και στηλών– λέγεται *συμμετρικός* (ως προς την πρώτη διαγώνιο) αν για όλα τα στοιχεία του ισχύει η ιδιότητα $a_{rc} == a_{cr}$. Οι παρακάτω εντολές ελέγχουν αν ο

```
   double a[N][N];
```

(όπου *N* σταθερά τύπου `int` με θετική τιμή) είναι συμμετρικός:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
```

```
   {
      for ( int c(0); c < N; ++c )
      {
         if( a[r][c] != a[c][r] )  symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Πράγματι, κάνουμε την υπόθεση ότι ο a είναι συμμετρικός (`symmet = true`) και στη συνέχεια αν βρούμε κάποιο ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$ διορθώνουμε την τιμή της *symmet* σε `false`.

Αυτό το κομμάτι προγράμματος είναι σπάταλο από άποψη χρόνου επεξεργασίας διότι, πριν απ' όλα:

- Ελέγχει κάθε ζευγάρι δυο φορές, π.χ. ενώ θα ελέγξει αν $a_{23} \neq a_{32}$, αργότερα θα ελέγξει και αν $a_{32} \neq a_{23}$, πράγμα άχρηστο.
- Ελέγχει –και μάλιστα δυο φορές– αν κάθε στοιχείο της πρώτης διαγωνίου είναι ίσο με τον εαυτό του.

Αυτές οι ατέλειες διορθώνονται αν αρχίζουμε την εξέταση της κάθε γραμμής ένα στοιχείο μετά τη διαγώνιο. Αν πάρουμε υπόψη μας ότι το στοιχείο της διαγωνίου στη γραμμή *r* είναι το a_{rr}, θα πρέπει να γράψουμε:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Στη χειρότερη περίπτωση –όταν ο πίνακας *είναι* συμμετρικός– η `if( a[r][c] != a[c][r] )...` θα εκτελεσθεί $\frac{1}{2}N(N-1)^2$ φορές, ενώ στον πρώτο αλγόριθμο θα εκτελεσθεί N^2 φορές.

Υπάρχει όμως και άλλη σπατάλη: αν βρούμε *ένα* ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$, δεν έχει νόημα να συνεχίσουμε τις συγκρίσεις· *ξέρουμε ότι ο πίνακας δεν είναι συμμετρικός*.

Στο παρακάτω κομμάτι έχουμε διορθώσει και αυτήν την ατέλεια:

```
   symmet = true;
   for ( int r(0); symmet && r < N; ++r )
   {
      for ( int c(r+1); symmet && c < N; ++c )
      {
         if ( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Δηλαδή, έχουμε πάλι μετρούμενες επαναλήψεις με τη διαφορά ότι τώρα βάλαμε και πρόσθετους ελέγχους για το αν ανιχνεύθηκε παραβίαση της συμμετρίας, οπότε σταματούμε αμέσως.

Μπορούμε να αποφύγουμε τη διπλή συνθήκη στις **for** αν ξαναγράψουμε, χρησιμοποιώντας τη `break`:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if ( a[r][c] != a[c][r] )
         {  symmet = false;   break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
```

Αν εκτελεσθεί η πρώτη **break** θα διακόψει την εκτέλεση της εσωτερικής **for**. Για να διακόψουμε και την εκτέλεση της εξωτερικής χρειάζεται η δεύτερη **break** που εκτελείται όταν βρει τη *symmet* με τιμή **false**. Αυτή η μετατροπή επιταχύνει σημαντικά το πρόγραμμά μας αφού επιταχύνει την εσωτερική **for** με την απλούστευση της συνθήκης συνέχισης.

Πάντως χρησιμοποιώντας την «καταραμένη» εντολή **goto**, μπορούμε να το κάνουμε ακόμη ταχύτερο:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
      {
         for ( int c(r+1); c < N; ++c )
         {
            if ( a[r][c] != a[c][r] )
            {  symmet = false;  goto lb99;  }
         } // for ( int c . . .
      } // for ( int r . . .
lb99: .....
```

Αυτή είναι μια θεμιτή χρήση της **goto** και κανείς δεν μπορεί να ισχυρισθεί ότι οι άλλες γραφές είναι καλύτερες από αυτήν. Πάντως, πρόσεξε και μια λεπτομέρεια: η εντολή με την ετικέτα **lb99** θα πρέπει να βρίσκεται ακριβώς μετά το τέλος των **for** και όχι οπουδήποτε αλλού μέσα στο πρόγραμμά μας.

Το πρώτο κομμάτι προγράμματος, με τις δύο **for**, περνάει από *όλα* τα στοιχεία του πίνακα (όλοι οι δυνατοί συνδυασμοί των *r* και *c*). Το δεύτερο κομμάτι σαρώνει τα στοιχεία του πίνακα που βρίσκονται *πάνω από την κύρια διαγώνιο* του πίνακα. Τα κομμάτια προγράμματος τρίτο, τέταρτο και πέμπτο σαρώνουν τα στοιχεία του πίνακα που βρίσκονται πάνω από την κύρια διαγώνιο του πίνακα στη χειρότερη περίπτωση.

Παράδειγμα 4 - Πολλαπλασιασμός πινάκων

Έστω ότι ο δισδιάστατος πίνακας a αποτελείται από *l* γραμμές και *m* στήλες, ενώ ο επίσης δισδιάστατος πίνακας b αποτελείται από *m* γραμμές και *n* στήλες. Να υπολογιστεί το γινόμενό τους c, που είναι δισδιάστατος πίνακας *l* γραμμών και *n* στηλών.

Τα στοιχείο του πίνακα c, στη γραμμή *r* και τη στήλη *c* (c_{rc}), υπολογίζεται, όπως γνωρίζουμε από τα μαθηματικά, με τον παρακάτω τύπο:

$$c_{rc} = \sum_{k=0}^{m-1} a_{rk} b_{kc}$$

όπου: $r = 0..l\text{-}1$, $c = 0..n\text{-}1$.

Δηλαδή, το στοιχείο c_{rc} είναι το άθροισμα όλων των όρων $a_{rk}b_{kc}$, όπου ο δείκτης *k* διατρέχει τις τιμές $0..m\text{-}1$.

Αυτός ο υπολογισμός μπορεί να δοθεί στη C++ με μιά **for** και μια εντολή εκχώρησης:

```
      c[r][c] = 0;
      for ( int k(0); k < m; ++k )
         c[r][c] += a[r][k]*b[k][c];
```

Το παρακάτω πρόγραμμα διαβάζει τα στοιχεία των πινάκων a και b, υπολογίζει το γινόμενό τους, δηλ. βρίσκει τα στοιχεία του πίνακα c και μετά τυπώνει τα στοιχεία του γινομένου c (καθώς και των πινάκων a και b), όπως βλέπεις στη συνέχεια:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   const int l = 3,  m = 5,  n = 2;

   int a[ l ][ m ],  b[ m ][ n ];
   int c[ l ][ n ];
```

```
   ifstream atx( "arr.txt" );
// Διάβασε τα στοιχεία του a
   for ( int r(0); r < l; ++r )
      for ( int c(0); c < m; ++c )  atx >> a[r][c];
// Διάβασε τα στοιχεία του b
   for ( int r(0); r < m; ++r )
      for ( int c(0); c < n; ++c )  atx >> b[r][c];
   atx.close();

// Πολλαπλασιασμός Πινάκων
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         c[r][c] = 0;
         for ( int k(0); k < m; ++k )
            c[r][c] += a[r][k]*b[k][c];
      } // for (c . . .
   } // for (r . . .

// Γράψε τα στοιχεία των a, b, c
   cout << " Στοιχεία του πίνακα a" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < m; ++c )
      {
         cout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα b" << endl;
   for ( int r(0); r < m; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << b[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα c" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << c[r][c] << "  ";
      }
      cout << endl;
   }
}
```

Για να δοκιμάσουμε το πρόγραμμα γράφουμε με τον κειμενογράφο το παρακάτω αρχείο:

```
1    2    3    4    5
6    7    8    9    1
1    4    7    2    5
10   10
12   45
47   18
12   31
18   29
```

και το φυλάγουμε με όνομα "arr.txt" (σε μορφή text). Όταν εκτελεσθεί το πρόγραμμα μας δίνει:

```
 Στοιχεία του πίνακα a
  1    2    3    4    5
  6    7    8    9    1
  1    4    7    2    5
```

```
 Στοιχεία του πίνακα b
 10   10
 12   45
 47   18
 12   31
 18   29
 Στοιχεία του πίνακα c
313  423
646  827
501  523
```

12.5 Η Σειρά Αποθήκευσης

Συνήθως, η σειρά αποθήκευσης των στοιχείων ενός πολυδιάστατου πίνακα θα σου είναι αδιάφορη. Υπάρχει όμως μια τουλάχιστον περίπτωση, που θα δούμε στην επόμενη παράγραφο, όπου σου χρειάζεται. Ας την δούμε λοιπόν.

Όπως αναφέραμε προηγουμένως, «τα στοιχεία ενός δισδιάστατου πίνακα *απόθηκεύονται κατά γραμμές*». Ας δούμε τι σημαίνει αυτό πιο συγκεκριμένα.

Έστω ότι έχουμε δηλώσει:

```
double a[5][7];
```

Η αποθήκευση των στοιχείων ξεκινάει από το a[0][0]. Η επόμενη θέση είναι για το a[0][1], η μεθεπόμενη για a[0][2] κ.ο.κ. Έξη θέσεις μετά το a[0][0] υπάρχει το τελευταίο στοιχείο της γραμμής 0, το a[0][6]. Μετά από αυτό έχουμε τη θέση για το a[1][0], το πρώτο στοιχείο της γραμμής 1. Όπως καταλαβαίνεις, το στοιχείο a[*r*][*c*] βρίσκεται 7*r* + *c* θέσεις μετά το a[0][0].

Προσοχή: ▶
Όταν λέμε «θέση», εννοούμε με την έννοια που είδαμε στην §12.3.2, δηλαδή θέση μνήμης που μπορεί να αποθηκεύσει ένα στοιχείο του πίνακα, στην περίπτωσή μας, μια τιμή τύπου `double` (συχνότατα αυτό σημαίνει 8 ψηφιολέξεις).◀

Ας κάνουμε ένα πείραμα για να επιβεβαιώσουμε τον παραπάνω τύπο. Δίνουμε τιμές στα στοιχεία του a ως εξής:

```
for ( int r(0); r <= 4; ++r )
{
   for ( int c(0); c <= 6; ++c )
   {
      a[r][c] = r + 0.1*c;
   } // for ( int c . . .
} // for ( int r . . .
```

Έτσι, τα στοιχεία της τελευταίας γραμμής (γραμμή 4) παίρνουν τιμές: 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6.

Τώρα, θα προσπαθήσουμε να δούμε τον a ως μονοδιάστατο. Δηλώνουμε:

```
double* p;
```

και βάζουμε:

```
p = &a[0][0];
```

Αφού η C++ καταλαβαίνει τα βέλη και τους πίνακες με τον ίδιο τρόπο, μπορούμε να δούμε το *p* ως μονοδιάστατο πίνακα, με στοιχεία τύπου `double`, που ξεκινάει από το a[0][0]. Από εκεί όμως ξεκινάει η αποθήκευση του a! Με το *p* λοιπόν μπορούμε να χειριστούμε τον a ως μονοδιάστατο πίνακα.

Σύμφωνα με τον τύπο που δώσαμε παραπάνω, τα στοιχεία της γραμμής 4 του a βρίσκονται στις θέσεις από 7×4 + 0 μέχρι 7×4 + 6 μετά το a[0][0]. Ζητούμε λοιπόν και μεις να δούμε τις τιμές αυτών των στοιχείων του πίνακα *p*:

```
r = 4;
for ( int c(0); c <= 6; ++c )
{
```

```
      cout << p[r*7+c] << "  ";
   }
   cout << endl;
```

Αποτέλεσμα; Αυτό ακριβώς που περιμένουμε:

```
4  4.1  4.2  4.3  4.4  4.5  4.6
```

Αν λοιπόν έχεις δηλώσει:

```
   T a[Nr][Nc];
```

τότε:

- ***το στοιχείο* a[*r*][*c*] *βρίσκεται* $r \cdot N_c + c$ *θέσεις (τύπου T) μετά το* a[0][0].**

12.5.1 Τρισδιάστατοι και Πολυδιάστατοι Πίνακες

Γενικότερα, αν έχεις δηλώσει:

```
   T a[N1][N2]...[Nm];
```

τότε:

- ***το στοιχείο* a[k_1][k_2]...[k_m] *βρίσκεται***

  (...($k_1 \times N_2 + k_2$)$\times N_3$ + ... + k_{m-1})$\times N_m + k_m$

  ***θέσεις (τύπου T) μετά το* a[0][0]...[0].**

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  a[2][2][2];
   int* ip;

   for ( int l(0); l < 2; ++l )
      for ( int r(0); r < 2; ++r )
         for ( int c(0); c < 2; ++c )
            a[l][r][c] = 100*(l+1) + 10*(r+1) + c+1;
   ip = &a[0][0][0];
   for ( int k(0); k < 8; ++k )
      cout << ip[k] << "  ";
   cout << endl;
}
```

Αποτέλεσμα:

```
111  112  121  122  211  212  221  222
```

Στα πρώτα τέσσερα στοιχεία το πρώτο ψηφίο, που βγαίνει από την τιμή της l(+1), έχει τιμή 1. Αυτά είναι τα στοιχεία που βρίσκονται στο «επίπεδο» $l = 0$ ενώ τα τέσσερα τελευταία βρίσκονται στο επίπεδο $l = 1$. Τα δύο πρώτα στοιχεία βρίσκονται στη γραμμή $r = 0$ του επιπέδου $l = 0$.

Το στοιχείο `a[1][0][1]`, που έχει τιμή 212, το βλέπουμε ως ip[((1×2 + 0)×2 + 1], δηλαδή ip[5].

Και το `a[1][0]` τι είναι; Τι είναι στην περίπτωση αυτήν το `a[1]`; Αντί για άλλη απάντηση δοκίμασε τις εντολές:

```
   for ( int l(0); l < 2; ++l )
   {
      cout << a[l] << "  " << &a[l][0][0] << endl;
      for ( int r(0); r < 2; ++r )
         cout  << "   " << a[l][r] << "  " << &a[l][r][0] << endl;
   }
```

που θα δώσουν:

```
0x22ff50  0x22ff50
   0x22ff50  0x22ff50
```

```
   0x22ff58  0x22ff58
0x22ff60  0x22ff60
   0x22ff60  0x22ff60
   0x22ff68  0x22ff68
```

Δηλαδή: το `a[l]` είναι βέλος προς το στοιχείο `a[l][0][0]` ενώ το `a[l][r]` είναι βέλος προς το `a[l][r][0]`.

12.6 Παράμετρος Πίνακας (ξανά)

Ας πούμε ότι έχουμε ορίσει έναν τύπο:

```
typedef char   ProSthlh[13];
```

με στόχο να παραστήσουμε σε στοιχεία αυτού του τύπου στήλες ΠροΠο. Αν δηλώσουμε π.χ.:

```
   ProSthlh c;
```

μπορούμε να δώσουμε:

```
   c[0] = '1';  c[1] = '1';  c[2] = 'X';...
   c[11] = 'X';  c[12] = '2';
```

Ύστερα από αυτό, μπορούμε να παραστήσουμε ένα δελτίο ΠροΠο, με *N* στήλες, με έναν πίνακα:

```
   ProSthlh deltio[N];
```

Θέλουμε μια συνάρτηση που θα παίρνει ένα δελτίο ΠροΠο και τη νικήτρια στήλη και θα μας επιστρέφει, ως τιμή, το πλήθος των στηλών του δελτίου που κερδίζουν (συμφωνούν με τη νικήτρια σε 11 σημεία τουλάχιστον).

Μπορούμε να γράψουμε το εξής σχέδιο:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
      μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      if ( ne >= 11 )  ++es;
   } // for
   return es;
} // winners
```

Η μέτρηση του πλήθους *ne* σωστών σημείων της στήλης deltio[*k*] γίνεται ως εξής:

```
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
```

Και να ολόκληρη η συνάρτηση:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
      {
         if ( deltio[k][j] == nik[j] )  ++ne;
      } // for ( int j . . .
      if ( ne >= 11 )  ++es;
   } // for ( int k . . .
   return es;
```

```
} // winners
```

Πώς την καλούμε; Έστω ότι έχουμε δηλώσει στη **main**:

```
   ProSthlh c, d[138];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners(d, 138, c) << endl;
```

Παρά την προσπάθεια που κάναμε (;) για να κρύψουμε την αλήθεια, αυτή δεν κρύβεται: στη μέτρηση των σωστών σημείων βλέπουμε το δελτίο ως *δισδιάστατο πίνακα*. Να αφήσουμε κατά μέρος τον τύπο ProSthlh και να πούμε τα πράγματα με το όνομά τους;

```
int winners2( char deltio[][13], int N, char nik[] )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
      if (ne >= 11)  ++es;
   } // for ( int k...
   return es;
} // winners2
```

Εδώ βλέπεις πώς περνούμε μια παράμετρο-δισδιάστατο πίνακα: το πλήθος των στηλών (13) γράφεται, το πλήθος των γραμμών όχι απαραίτητα. Αν θέλεις να περάσεις έναν πολυδιάστατο πίνακα θα πρέπει να καθορίζεις όλες τις διαστάσεις εκτός από την πρώτη.

Αν έχουμε δηλώσει στη **main**:

```
   char c[13], d[138][13];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners2(d, 138, c) << endl;
```

Μέχρι εδώ καλά. Δες τώρα ένα άλλο πρόβλημα.

Ας πούμε τώρα ότι θέλουμε να περάσουμε έναν δισδιάστατο πίνακα χωρίς να έχουμε καθορισμένο πλήθος στηλών· θέλουμε να περάσουμε τα πλήθη γραμμών και στηλών ως παραμέτρους. Πώς μπορεί να γίνει αυτό; Με βάση αυτά που είπαμε στην προηγούμενη παράγραφο. Δηλαδή *περνούμε τον πίνακα ως μονοδιάστατο*. Ας το δούμε με ένα παράδειγμα.

Παράδειγμα

Θέλουμε μια συνάρτηση που θα παίρνει έναν τετραγωνικό πίνακα a με στοιχεία τύπου `double` και θα μας επιστρέφει τιμή `true` αν ο a είναι συμμετρικός και `false` αν δεν είναι.

Η ιδέα είναι η εξής: Όταν καλούμε τη συνάρτηση θα της δίνουμε τη διεύθυνση του πρώτου στοιχείου του πίνακα και πλήθος *N* γραμμών και στηλών. Μέσα στη συνάρτηση δεν θα γράφουμε `a[r][c]` αλλά `a[r*n+c]`.

```
bool isSymmetric( const double* a, int n ) // double a[n][n]
{
   bool symmet( true );

   for ( int r(0); r < n; ++r )
   {
      for ( int c(r+1); c < n; ++c )
      {
//       if ( a[r][c] != a[c][r] ) {  symmet = false;  break;  }
         if ( a[r*n+c] != a[c*n+r] ) {  symmet = false;  break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
   return symmet;
} // isSymmetric
```

Ας δούμε τώρα πώς την καλούμε. Έστω ότι έχουμε:

```
double p1[2][2] = { {1,2}, {3,4} },
       p2[3][3] = { {1, 0, 1.5}, {0, 1.8, 2}, {1.5, 2, 4.1} };
```

Αν θέλουμε να ελέγξουμε τη συμμετρικότητα των p1 και p2 με τη συνάρτησή μας γράφουμε:

```
if ( isSymmetric(&p1[0][0], 2) )  cout << " ο p1 ναι" << endl;
                            else  cout << " ο p1 όχι" << endl;
if ( isSymmetric(&p2[0][0], 3) )  cout << " ο p2 ναι" << endl;
                            else  cout << " ο p2 όχι" << endl;
```

€€€

Όπως βλέπεις –και στην περίπτωση αυτήν– χρησιμοποιούμε αυτά που ξέρουμε για την εσωτερική παράσταση για να περάσουμε ως παράμετρο και να χειριστούμε έναν δισδιάστατο πίνακα. Αυτό είναι μάλλον ένα τέχνασμα αλλά είναι ο μόνος τρόπος που έχουμε.[4]

12.6.1 Και Άλλα Τεχνάσματα

Στην Άσκ. 9-6 ζητούμε μια «*συνάρτηση rawSum() που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο r1 και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής r1.*»

Όποιος έχει διαβάσει με προσοχή αυτά που είδαμε μέχρι τώρα θα πεί: «Δεν χρειάζεται, έχουμε τη *vectorSum*!» Πράγματι, αν

```
double a[5][7], z;
```

και θέλουμε το άθροισμα των στοιχείων της γραμμής 2 μπορούμε να γράψουμε:

```
z = vectorSum( a[2], 7, 0, 6 );
```

ή

```
z = vectorSum( &a[2][0], 7, 0, 6 );
```

Εδώ στηριζόμαστε στο πώς αποθηκεύει η C++/C τους δισδιάστατους πίνακες και θεωρείται τέχνασμα. Λύσε λοιπόν την Άσκ. 9-6 με πιο «ορθόδοξο» τρόπο.

Ας δούμε όμως και μια περίπτωση που τα τεχνάσματα δεν δουλεύουν. Έχουμε τον πίνακα:

```
double p3[6][9] = { {1, 2, 4, 0, 5, 2, 7, 9, 0},
                    {2, 2, 0, 3, 4, 5, 9, 7, 8},
                    {4, 0, 2, 6, 3, 4, 0, 5, 7},
                    {3, 4, 8, 3, 5, 7, 0, 4, 1},
                    {4, 5, 1, 4, 7, 9, 0, 5, 7},
                    {1, 1, 5, 0, 0, 0, 8, 8, 6} };
```

και θέλουμε να ελέγξουμε αν οι (τετραγωγικοί) υποπίνακες από `p3[0][0]` μέχρι `p3[2][2]` και από `p3[2][3]` μέχρι `p3[5][6]` είναι συμμετρικοί. Θα μπορούσαμε να ρωτήσουμε *isSymmetric*`(&p3[0][0], 3)` και *isSymmetric*`(&p3[2][3], 4)`; Ούτε για αστείο!

Αν μας ενδιαφέρουν τέτοιες επεξεργασίες ξαναγράφουμε την *isSymmetric* ώστε να τηρούμε αυτό που είπαμε και πιο πριν:

- ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Στο παράδειγμά μας η *isSymmetric* θα πρέπει να ξαναγραφεί ως εξής:

```
bool isPartSymmetric( const double* a, int nRow, int nCol,
                      int rUL, int cUL, int rLR, int cLR )
```

Οι παράμετροι της πρώτης γραμμής περνούν στη συνάρτηση τον πίνακα: αρχή, γραμμές, στήλες. Οι παράμετροι της δεύτερης γραμμής περνούν την περιοχή επεξεργασίας που τη θεωρούμε ως ορθογώνιο και περνούμε: τη γραμμή (*rUL*) και τη στήλη (*cUL*) πάνω αριστε-

[4] Αργότερα θα μάθουμε τους δυναμικούς πίνακες που έχουν πιο «ορθόδοξο» χειρισμό.

ρής κορυφής και τη γραμμή (*rLR*) και τη στήλη (*cLR*) κάτω δεξιάς κορυφής. Για τα παραδείγματα που είπαμε πιο πάνω θα καλούμε ως εξής:

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          0, 0, 2, 2 ) . . .
```

και

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          2, 3, 5, 6 ) . . .
```

12.7 Οι Παράμετροι της `main`

Ας πούμε ότι έχουμε ένα πρόγραμμα φυλαγμένο στο αρχείο mainargs.exe. Για να ζητησουμε την εκτέλεσή του από γραμμή εντολών δίνουμε:

```
D:\>mainargs<enter>
```

Όπως θα έχεις δει όμως, μερικές φορές εκτός από το όνομα του αρχείου γράφουμε και διάφορες παραμέτρους, οι οποίες περνούν στο πρόγραμμα και επηρεάζουν την εκτέλεσή του. Μπορούμε να γράψουμε τέτοια προγράμματα; Ναι! Ας πούμε ότι το αρχικό πρόγραμμα αυτού που έχουμε στο mainargs.exe είναι το παρακάτω:

```
#include <iostream>
using namespace std;
int main( int argc, char* argv[] )
{
   cout << "Η τιμή της argc είναι " << argc << endl;
   cout << "Αυτά είναι τα ορίσματα που πέρασαν στην int main:"
        << endl;

   for ( int k(0); k <= argc; ++k )
      cout << "   argv[" << k << "]: " << argv[k] << endl;
} // main
```

Η διαφορά του από αυτά που είδαμε μέχρι τώρα βρίσκεται στις παραμέτρους που υπάρχουν στην επικεφαλίδα της **main**. Η δεύτερη παράμετρος είναι ένας πίνακας που κάθε του στοιχείο είναι ένα «C-style string»· μπορεί να τη δεις και ως `char** argv`. Η πρώτη παράμετρος μας λέει πόσα στοιχεία έχει πίνακας (+1)· για την ακρίβεια μας λέει τον δείκτη του τελευταίου στοιχείου.

Τι μπορεί να είναι αυτό το "`char* argv[]`"; Όπως είναι γραμμένο, μας λέει ότι έχουμε πίνακα που το κάθε στοιχείο του είναι τύπου "`char*`". Δηλαδή, μπορεί να είναι δισδιάστατος πίνακας με στοιχεία τύπου `char`; Ναι, αλλά κάπως διαφορετικός από αυτούς που είδαμε. Παρ' όλα αυτά μπορούμε να τον αξιοποιήσουμε με αυτά που μάθαμε.

Όπως βλέπεις, αυτό που κάνει το πρόγραμμά μας είναι να τυπώνει τις τιμές των παραμέτρων. Να ένα παράδειγμα εκτέλεσης: Ζητούμε να εκτελεστεί ως εξής:

```
E:\cpp2bk\progs>mainargs όρισμα1 a\b "όρισμα 3" 4 5.5 c\d?'ef<enter>
```

και παίρνουμε:

```
Η τιμή της argc είναι 7
Αυτά είναι τα ορίσματα που πέρασαν στην int main:
   argv[0]: mainargs
   argv[1]: όρισμα1
   argv[2]: a\b
   argv[3]: όρισμα 3
   argv[4]: 4
   argv[5]: 5.5
   argv[6]: c\d?'ef
   argv[7]:
```

Ας δούμε λοιπόν τις τιμές των στοιχείων του argv:

- Το `argv[0]` είναι το όνομα του αρχείου που περιέχει το (εκτελέσιμο) πρόγραμμα (μπορεί να δεις και την πλήρη διαδρομή (path) προς αυτό).

- Το `argv[1]` δείχνει τον πρώτο ορμαθό χαρακτήρων μετά το όνομα του αρχείου, δηλαδή το πρώτο όρισμα (στην περίπτωσή μας: `όρισμα1`).
- Το `argv[`*k*`]` δείχνει τον *k*-στό ορμαθό χαρακτήρων μετά το όνομα του αρχείου.
- Το `argv[argc-1]` δείχνει το τελευταίο όρισμα (ορμαθό χαρακτήρων).
- Το `argv[argc]` είναι φρουρός: περιέχει το (βέλος) 0.

Όπως βλέπεις, τα ορίσματα-ορμαθοί ξεχωρίζουν μεταξύ τους με ένα διάστημα. Στο τρίτο όρισμα θέλαμε να περάσουμε την τιμή `όρισμα 3`, που περιέχει διάστημα. Γι' αυτό υποχρεωθήκαμε να το γράψουμε: `"όρισμα 3"`.

12.8 Τελικώς ...

Τα βέλη είναι εργαλεία με τα οποία μπορείς να χειρίζεσαι πίνακες. Η χρήση βελών (υποτίθεται ότι) κάνει τα προγράμματά σου πιο γρήγορα αλλά η απλή κωδικοποίησή τους με δείκτες τα κάνει πιο ευανάγνωστα (και πιο σίγουρα). Προτίμησε λοιπόν τους δείκτες και χρησιμοποίησε βέλη μόνον όταν δεν μπορείς να τα αποφύγεις.

Δεν μπορείς να τα αποφύγεις (προς το παρόν) όταν περνάς πολυδιάστατο πίνακα σε συνάρτηση.

Όταν περνάς πίνακα σε συνάρτηση μην προσπαθείς να ξεγελάσεις τον μεταγλωττιστή με τεχνάσματα. Επαναλαμβάνουμε για τρίτη φορά τη συμβουλή που δώσαμε:

♦ ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Ασκήσεις

Α Ομάδα

12-1 Τι δίνει το παρακάτω πρόγραμμα; (εκτέλεσέ το *εσύ, όχι ο υπολογιστής*)

```
#include <iostream>
using namespace std;

int qaw( int* p )
{
   *p = 5;
   return *p;
} // qaw

int main()
{
   int x( 0 );
   cout << " the value of x is " << x << endl;
   int y( qaw(&x) );
   cout << " the new value of x is " << x << endl;
   cout << " the value of y is " << y << endl;
} // main
```

12-2 Τα ίδια για το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   char a[] = "abcdefgh";
   for ( char* p(a); *p != 0; ++p )  cout << p << endl;
} // main
```

12-3 Έστω ότι έχουμε έναν τετραγωνικό πίνακα πραγματικών αριθμών. Το άθροισμα των στοιχείων της (κύριας) διαγωνίου ονομάζεται **ίχνος** (trace) του πίνακα. Γράψε συνάρτηση *trace* που θα τροφοδοτείται με τον πίνακα και θα υπολογίζει και θα επιστρέφει το ίχνος.

12-4 Γράψε συνάρτηση *trace2* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της *δεύτερης* διαγωνίου.

12-5 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων δύο στηλών *c1, c2* ενός δισδιάστατου πίνακα. Το ίδιο για τις τιμές των στοιχείων δύο γραμμών *r1, r2*.

12-6 Γράψε συνάρτηση *rawSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο *r1* και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής *r1*.

Γράψε συνάρτηση *colSum()* που θα κάνει το ίδιο για τα στοιχεία της στήλης *c1*.

12-7 Γράψε συνάρτηση *isDiagonal()* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα επιστρέφει τιμή `true` αν και μόνον αν ο πίνακας είναι διαγώνιος (όλα τα στοιχεία εκτος της *πρώτης* διαγωνίου είναι μηδέν).

Β Ομάδα

12-8 Τροποποίησε το πρόγραμμα της Άσκ. 9-2 ώστε να δίνει:

```
h
gh
fgh
efgh
defgh
cdefgh
bcdefgh
abcdefgh
```

Υπόδ.: Αν δεν μπορείς να το κάνεις με βέλη, κάνε το πρώτα όπως μπορείς και μετά μετάτρεψέ το ώστε να χρησιμοποιεί μόνο βέλη.

12-9 Γράψε συνάρτηση *periphSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών με *nR* γραμμές και *nC* στήλες και θα υπολογίζει και θα επιστρέφει το άθροισμα των περιφερειακών στοιχείων (των γραμμών 0 και *nR*-1 και των στηλών 0 και *nC*-1.)

12-10 Γράψε συνάρτηση *matrix2Sum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε συνάρτηση *matrix3Sum()* που θα τροφοδοτείται με τρισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε πρόγραμμα που θα δείχνει τον τρόπο χρήσης των δύο συναρτήσεων.

12-11 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο τις (πραγματικές) τιμές των στοιχείων ενός δισδιάστατου πίνακα και θα υπολογίζει και θα μας δίνει:

- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *γραμμών* του,
- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *στηλών* του.

Γ Ομάδα

12-12 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης ενός τετραγωνικού πίνακα. Δοκίμασε να κάνεις το ίδιο για τα στοιχεία της *r* γραμμής και της *c* στήλης όταν $r \neq c$.

12-13 Γράψε την *isPartSymmetric()* που προδιαγράψαμε στην§12.6.1.

κεφάλαιο

13

Συναρτήσεις ΙΙ – Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:

Αφού έμαθες –πολύ νωρίς– να γράφεις συναρτήσεις που σου χρειάζονται και δεν υπάρχουν στις βιβλιοθήκες της C++ τώρα *θα μάθεις να γράφεις δικές σου εντολές* ή αλλιώς συναρτήσεις `void`. Θα μάθεις ακόμη *να περνάς τιμές από μια συνάρτηση προς μια άλλη που την κάλεσε* μέσω παραμέτρων.

Προσδοκώμενα αποτελέσματα:

Με τα εργαλεία που σου δίνουμε μπορείς πια να γράψεις πρόγραμμα για να λύσεις μη τετριμμένα προβλήματα.

Έννοιες κλειδιά:

- *συναρτήσεις* `void`
- *παράμετρος-αναφοράς*
- *παράμετρος-βέλος* (*pointer*)
- *τύπος-αναφοράς*
- *παράμετρος-ρεύμα από/προς αρχείο*
- *δομημένος προγραμματισμός*
- *βήμα-προς-βήμα ανάλυση*
- *καθολικές μεταβλητές*
- *στατικές μεταβλητές*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Έλυσες την Άσκ. 9-1; Ας τη δούμε μαζί:

```
#include <iostream>
using namespace std;

int qaw( int* p )
{
   *p = 5;
   return *p;
} // qaw

int main()
{
   int x( 0 );
   cout << " the value of x is " << x << endl;
   int y( qaw(&x) );
   cout << " the new value of x is " << x << endl;
   cout << " the value of y is " << y << endl;
} // main
```

Η *qaw* έχει παράμετρο βέλος προς αντικείμενο τύπου `int`. Στη μοναδική κλήση της *qaw* –"`int y( qaw(&x) )`"– περνούμε ως όρισμα βέλος προς τη *x* και αυτό γίνεται τιμή του *p*. Άρα το `*`*p* –στο οποίο δίνουμε την τιμή 5– είναι ακριβώς ή *x*.

Έτσι, το πρόγραμμα θα μας δώσει:

```
the value of x is 0
the new value of x is 5
the value of y is 5
```

Ενώ μέχρι τώρα ξέραμε ότι ο μόνος τρόπος να πάρουμε τιμή από μια συνάρτηση ήταν το όνομά της με τα ορίσματα, τώρα βλέπουμε ότι μπορούμε να παίρνουμε τιμές και από τις παραμέτρους, αν αυτές είναι βέλη.

Αυτός είναι ο πάγιος τρόπος της C. Η C++ τον έχει κληρονομήσει αλλά μας δίνει και μια σαφώς πιο εύχρηστη παραλλαγή του. Αυτά και άλλα παρεμφερή θα δούμε σε αυτό το κεφάλαιο.

Ακόμη, στο κεφάλαιο αυτό θα γράψουμε και πρόγραμμα διαφορετικό από τα προηγούμενα. Μέχρι τώρα γράφαμε προγράμματα για να δείχνουμε μια νέα έννοια ή μια νέα τεχνική. Τώρα θα γράψουμε πρόγραμμα που θα λύνει ένα πρόβλημα.

13.1 Ένα Παλιό Πρόβλημα Ξανά

Στην §2.7 είχαμε λύσει το εξής πρόβλημα:

> *Από ύψος h αφήνεται να πέσει προς τη γή ένα σώμα. Να γραφεί πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του h και θα υπολογίζει και θα γράφει:*
> *α) τον χρόνο που θα κάνει το σώμα μέχρι να φτάσει στην επιφάνεια της γής*
> *β) η ταχύτητά του τη στιγμή της πρόσκρουσης.*
> *Να αγνοηθεί η αντίσταση του αέρα. Επιτάχυνση βαρύτητας: g = 9.81 m/sec².*

Είχαμε κάνει ένα σχέδιο για τη λύση:

Διάβασε το *h*
Υπολόγισε τα *tP*, *vP*
Τύπωσε τα *tP*, *vP*

και γράψαμε το πρόγραμμα.

Τώρα θέλουμε κάτι άλλο: Να διαχωρίσουμε τα δύο τελευταία βήματα, να τα βγάλουμε από τη **main** και να τα «κρύψουμε» σε ξεχωριστές συναρτήσεις, Γιατί; Διότι θέλουμε να έχουμε τη δυνατότητα να αλλάζουμε την υλοποίησή τους όποτε μας χρειαστεί.

Ας ξεκινήσουμε από το τελευταίο. Θέλουμε να γράψουμε μια συνάρτηση που θα τροφοδοτείται με τα *tP* και *vP* και θα γράφει στην οθόνη τις τιμές τους (μαζί και την αρχική τιμή του ύψους). Εύκολο φαίνεται, αλλά υπάρχει ένα ερώτημα: Τι τιμή θα επιστρέφει αυτή η συνάρτηση, αφού δεν υπάρχει κάτι που να υπολογίζει;

Θα μπορούσαμε να βάλουμε μια άχρηστη, επιστρεφόμενη τιμή, π.χ.:

```
int displayResults( double h, double t, double v )
{
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << t << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << v << " m/sec";
   return 0;
} // displayResults
```

που είναι μια χαρά. Η C++ όμως μας δίνει τη δυνατότητα να γράψουμε μια *συνάρτηση χωρίς τύπο*:

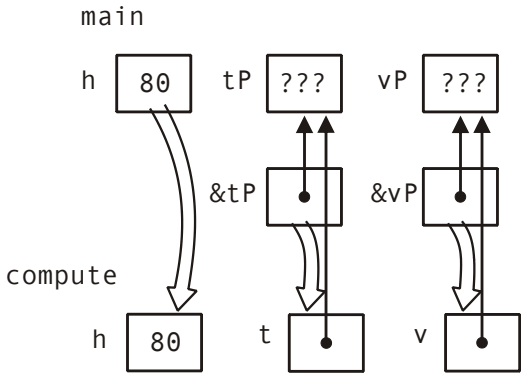
```
void displayResults( double h, double t, double v )
{
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << t << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << v << " m/sec" << endl;
} // displayResults
```

Το `void` στην αρχή δείχνει ότι *η συνάρτηση δεν επιστρέφει τιμή*.[1]

Πώς την καλούμε; Με το όνομά της και τα ορίσματά της, αλλά όχι μέσα σε παράσταση:

```
int main()
{
   const double  g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας

   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if (h >= 0)  {  // Υπολόγισε τα tP, vP
   // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
      tP = sqrt((2/g)*h);
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))

      displayResults( h, tP, vP );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

[1] Το «συνάρτηση χωρίς τύπο» δεν είναι και τόσο σωστό, τουλάχιστον κατ' αρχήν. Για τη C++ *ο* "`void`" *είναι τύπος με κενό σύνολο τιμών*.



Σχ. 13-1 Η κλήση της *compute* από τη **main**. Η τιμή της *h* της **main** αντιγράφεται ως τιμή της *h* της *compute()*. Στη μεταβλητή-βέλος *t* αντιγράφεται η τιμή-βέλος **&***tP*. Έτσι η *t* δείχνει τη μεταβλητή *tP* της **main**, δηλαδή ***t* είναι η *tP*, και αν δώσουμε με κάποιον τρόπο μια τιμή στην ***t* αυτή γίνεται τιμή της *tP*. Τα ίδια ισχύουν και για τις ***v* και *vP*.

13.2 Επιστροφή Τιμών από τη Συνάρτηση I

Τώρα, ας προσπαθήσουμε να γράψουμε μια συνάρτηση, ας την πούμε *compute*, που θα τροφοδοτείται με το ύψος (*h*) και θα υπολογίζει και θα επιστρέφει τα *tP*, *vP*.

Θα πεις: «αφού ξέρουμε ότι μια συνάρτηση επιστρέφει μια τιμή, ας γράψουμε *δύο* συναρτήσεις!» Όχι! Οι προδιαγραφές μας λένε *μία* συνάρτηση· μπορεί να γίνει κάτι;

Μήπως μπορούμε να παίρνουμε αποτελέσματα μέσω των παραμέτρων; Πώς; Στις «Εισαγωγικές Παρατηρήσεις» είδαμε τη λύση: θα βάλουμε παραμέτρους-βέλη! Δηλαδή, γράφουμε μια:

```
void compute( double h, double* t, double* v )
```

και την καλούμε ως εξής:

```
compute( h, &tP, &vP );
```

Και γιατί βάλαμε "`void`"; Διότι *η συνάρτηση δεν επιστρέφει τιμή* μέσα σε μια παράσταση που υπάρχει το όνομά της αλλά μόνο με τις παραμέτρους της. Τα λέμε πιο κάτω.

Ας δούμε, με τη βοήθεια του Σχ. 13-1, τι θα γίνει κατά την κλήση: η τιμή της *h* της **main** θα αντιγραφεί στην παράμετρο *h* της *compute()*. Η τιμή της παράστασης `&tP` θα αντιγραφεί στην παράμετρο *t* της *compute()*. Αλλά τι είναι η τιμή της `&tP`; Είναι ένα βέλος που δείχνει τη μεταβλητή *tP* της **main**. Άρα, μετά την αντιγραφή, η *t* της *compute()*, που είναι μεταβλητή βέλος (`double* t`), θα δείχνει επίσης τη μεταβλητή *tP* της **main**. Τα ίδια θα γίνουν και με τις *vP* και *v*. Αν λοιπόν, μέσα στην *compute()*, αλλάξω τις τιμές των ***t* και ***v* αλλάζω τις τιμές των *tP* και *vP* αντίστοιχα! Αυτό ήτανε λοιπόν:

```
void compute( double h, double* t, double* v )
{
   const double  g = 9.81; // m/sec², επιτάχυνση της βαρύτητας

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   *t = sqrt((2/g)*h);
   *v = -g*(*t);
// (*t ≈ √(2h/g)) && (*v ≈ -√(2hg))
} // compute
```

Και να πώς γίνεται τώρα πια η **main**:

```
int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
```

```
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   {
      compute( h, &tP, &vP );
      displayResults( h, tP, vP );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως η *displayResults()*, έτσι και η *compute* είναι μια συνάρτηση χωρίς τύπο (**void**). Δηλαδή δεν την καλούμε μέσα σε μια παράσταση, όπου θα μας επιστρέψει κάποια τιμή. Πώς παίρνουμε τα αποτελέσματα των υπολογισμών που κάνει; Μέσω των παραμέτρων: για την ακρίβεια, μέσω των παραμέτρων γνωστοποιούμε στη συνάρτηση τις θέσεις της μνήμης (**&tP** και **&vP**) όπου θέλουμε να αποθηκευτούν τα αποτελέσματα και αφού τελειώσει τη δουλειά της τα παίρνουμε και τα χρησιμοποιούμε.

Ας γράψουμε με τον ίδιο τρόπο και μια συνάρτηση, με όνομα *inputH()*, που θα διαβάζει από το πληκτρολόγιο το ύψος και θα το φέρνει στη **main**, όταν την καλεί.[2] Αν ο χρήστης δώσει αρνητικό αριθμό θα του δίνει *μία* ευκαιρία να διορθώσει το λάθος του.

```
   void inputH( double* h )
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> *h;
      if ( *h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> *h;  }
   } // inputH
```

Και να πώς γίνεται τελικώς το πρόγραμμά μας:

```
#include <iostream>
#include <cmath>
using namespace std;

   void inputH( double* h )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   void compute( double h, double* t, double* v )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   void displayResults( double h, double tP, double vP )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   inputH( &h );
   if ( h >= 0 )
   {
      compute( h, &tP, &vP );
      displayResults( h, tP, vP );
```

[2] Βέβαια, αφού η *inputH()* επιστρέφει μια τιμή θα μπορούσαμε να γράψουμε συνάρτηση με τύπο:

```
   double inputH()
   {
      double locH;
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> locH;
      if ( locH < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> locH;  }
      return locH;
   } // inputH
```

Ναι, θα μπορούσαμε, αλλά... διάβασε παρακάτω.

```
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως βλέπεις, όλες οι δουλειές γίνονται στις συναρτήσεις και η **main** παίζει πια ρόλο «συντονιστικό». Και μπορεί αυτό να μη λέει και πολλά πράγματα σε ένα τόσο μικρό πρόγραμμα, αλλά τα μεγάλα προγράμματα δεν είναι δυνατό να γραφούν διαφορετικά.

Πόσο μεγάλες είναι αυτές οι συναρτήσεις; Από παλιά, οι προγραμματιστές έχουν έναν εμπειρικό κανόνα που λέει: *καμιά συνάρτηση του προγράμματος δεν πιάνει περισσότερο από μια σελίδα εκτύπωσης* (αυτό σημαίνει 60 γραμμές περίπου). Πάντως, ένας πιο καλός κανόνας είναι ο εξής: *κάθε συνάρτηση ανταποκρίνεται σε έναν συγκεκριμένο υπολογιστικό στόχο*. Φυσικά, κάθε υπολογιστικός στόχος μπορεί να διασπασθεί σε μικρότερους. Αυτό αντιστοιχεί σε μια συνάρτηση που καλεί άλλες συναρτήσεις. Στη συνέχεια θα δούμε αυτά τα πράγματα με παράδειγμα.

Όπως είπαμε, αυτός ο τρόπος για να βγάζουμε τιμές από μια συνάρτηση μέσω των παραμέτρων είναι κληρονομιά από τη C. Στην επόμενη παράγραφο θα μάθουμε έναν άλλον τρόπο που μας δίνει η C++. Πάντως, πολλοί προγραμματιστές προτιμούν τον τρόπο της C για λόγους που θα εξηγήσουμε αργότερα.

13.3 Επιστροφή Τιμών από τη Συνάρτηση II

Θα ξεκινήσουμε με έναν κανόνα, που θα εξηγηθεί αργότερα:

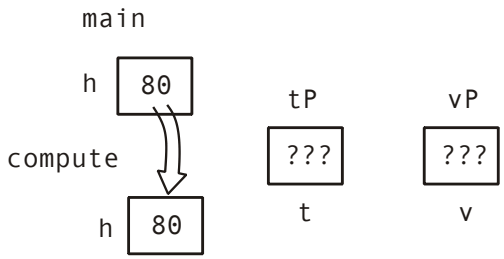
- ***Αν μετά τον τύπο της παραμέτρου μιας συνάρτησης βάλεις το σύμβολο "&" τότε η παράμετρος είναι «διπλής κατεύθυνσης»· δηλαδή οι αλλαγές της τιμής της τυπικής παραμέτρου, που γίνονται μέσα στη συνάρτηση, γίνονται αλλαγές τιμής της αντίστοιχης πραγματικής παραμέτρου.***

Αυτές οι παράμετροι λέγονται **παράμετροι αναφοράς** (reference parameters). Όπως καταλαβαίνεις,

- ***Η πραγματική παράμετρος που αντιστοιχεί σε μια παράμετρο αναφοράς δεν μπορεί να είναι σταθερά ή παράσταση, αλλά κάτι που να μπορεί να αλλάξει η τιμή του δηλαδή τροποποιήσιμη τιμή-l, π.χ. μια μεταβλητή ή ένα στοιχείο πίνακα.***

Ξαναγράφουμε ολόκληρο το πρόγραμμα της προηγούμενης παραγράφου χρησιμοποιώντας, αντί για βέλη, παραμέτρους αναφοράς:

```
#include <iostream>
#include <cmath>
using namespace std;

   void inputH( double& h )
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
      if ( h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> h;  }
   } // inputH

   void compute( double h, double& t, double& v )
   {
      const double  g = 9.81; // m/sec2, επιτάχυνση της βαρύτητας

   // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
      t = sqrt( (2/g)*h );
      v = -g*t;
   // (t ≈ √(2h/g)) && (v ≈ -√(2hg))
   } // compute
```



Σχ. 13-2 Όταν καλείται η *compute()*, η πραγματική παράμετρος τιμής (*h* της **main**) αντιγράφεται στην τυπική (*h* της *compute*). Για τις παραμέτρους αναφοράς τα πράγματα είναι αλλιώς: οι δύο συναρτήσεις «κατέχουν από κοινού» τις ίδιες θέσεις της μνήμης, τις *tP*, *vP* της **main**. Η καλούμενη συνάρτηση (*compute*) δίνει δικά της ονόματα στις δύο θέσεις και τις χειρίζεται με αυτά.

```
   void displayResults(double h, double tP, double vP)
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   inputH( h );
   if ( h >= 0 )
   {
      compute( h, tP, vP );
      displayResults( h, tP, vP  );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Πρόσεξε την επικεφαλίδα της *compute()*:

```
   void compute( double h, double& t, double& v )
```

Ο τύπος των *t* και *v* είναι **double&**. Δες και την κλήση της:

```
      compute( h, tP, vP );
```

Στις δύο τελευταίες παραμέτρους βάζουμε απλώς τα αναγνωριστικά των αντίστοιχων μεταβλητών: **tP**, **vP**.

Πρόσεξε τώρα το σώμα της *compute()*: Οι εκχωρήσεις γίνονται στις *t* και *v* και όχι στις ***t** και ***v** που είχαμε στην προηγούμενη παράγραφο. Παρόμοιες διαφορές βλέπεις και στην *inputH()*.

Όπως βλέπεις, οι παράμετροι αναφοράς σου δίνουν τη δυνατότητα να γράφεις το πρόγραμμά σου πιο απλά. Και, παρ' όλο που οι δύο τρόποι είναι κατά βάση ίδιοι, δες και ένα σενάριο κλήσης και για να τις σκέφτεσαι πιο απλά.

Όταν καλείται η συνάρτηση, στην περίπτωσή μας η *compute()*, της παραχωρείται μνήμη για τις παραμέτρους τιμής (όπως είναι η *h*) και τα τοπικά της αντικείμενα (όπως είναι η σταθερά *g*) αλλά όχι για τις παραμέτρους αναφοράς. Για αυτές γίνεται το εξής: της γνωστοποιούνται οι θέσεις τους στη μνήμη και η συνάρτηση τους δίνει δικά της ονόματα. Όσο εκτελείται η *compute()* μπορείς να σκέφτεσαι ότι η *tP* έχει δύο ονόματα: *tP* για τη **main** και *t* για την *compute()*. Παρομοίως, η *vP* της **main** έχει το όνομα *v* για την *compute()*. Δηλαδή, η συνάρτηση που καλεί και η συνάρτηση που καλείται «κατέχουν από κοινού» τις παραμέτρους αναφοράς. Αυτό προσπαθεί να δείξει και το Σχ. 13-2.

Το σενάριό μας δεν απέχει πολύ από την αλήθεια: Αν ζητήσεις να δεις τις διευθύνσεις των *tP* και *vP* της **main** και αυτές των *t* και *v* της *compute()* θα τις βρεις ίδιες.

Αλλά τώρα προσοχή:

Δηλαδή, αν έχεις τη

```
void inputH( double& h )
```

και τις:

```
int i;
double d;
```

μπορείς να βάλεις:

```
inputH( d );
```

αφού η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, *d*, είναι τύπου `double`.

Δεν μπορείς να βάλεις:

```
inputH( i );
```

διότι η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, *i*, είναι τύπου `int`.

Δεν μπορείς να βάλεις:

```
inputH( 1.0 + sqrt(d) );
```

διότι η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, `1.0 + sqrt(d)`, είναι παράσταση και όχι μεταβλητή.

Αν έχεις καταλάβει τον τρόπο αντιστοίχισης, που περιγράψαμε πιο πάνω, αυτά είναι αυτονόητα.

Στη συνέχεια θα προτιμήσουμε τις παραμέτρους αναφοράς από τις παραμέτρους-βέλη. Έτσι:

- όταν θέλουμε παράμετρο «μονής κατεύθυνσης», που θα μεταβιβάζει στοιχεία από την καλούσα συνάρτηση προς την καλούμενη θα χρησιμοποιούμε παράμετρο τιμής,
- όταν θέλουμε παράμετρο «διπλής κατεύθυνσης», που θα μεταβιβάζει στοιχεία και από την καλούσα προς την καλούμενη και αντίθετα, από την καλούμενη προς την καλούσα, θα χρησιμοποιούμε παράμετρο αναφοράς.

13.3.1 Παράμετρος `unsigned`; (ξανά)

Τώρα, θα ξαναδιατυπώσουμε, πιο προσεκτικά, τον κανόνα που δώσαμε στην §7.6:

♦ ***Μην βάζεις στις συναρτήσεις σου παραμέτρους τιμής τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

Για τις παραμέτρους αναφοράς αυτά δεν ισχύουν· αυτό που ισχύει είναι: *Ο τύπος της πραγματικής παραμέτρου θα πρέπει να είναι ίδιος με αυτόν της τυπικής παραμέτρου* (αλλιώς δεν περνάει από τον μεταγλωττιστή).

13.4 Τύποι Αναφοράς

Εκτός από τη λίστα παραμέτρων, μπορούμε να βάλουμε **τύπο αναφοράς** (reference type) και σε δηλώσεις μεταβλητών μέσα στη συνάρτηση (ή σε καθολικές). Ας πούμε ότι δηλώνουμε:

```
int x;
int& r( x );
```

Τι καταφέραμε; *Τα αναγνωριστικά x και r καθορίζουν την ίδια θέση της μνήμης*. Έτσι, αν δώσουμε τις εντολές:

```
r = 10;
cout << " x = " << x << "   r = " << r << endl;
x = 15;
```

```
   cout << " x = " << x << "   r = " << r << endl;
```

θα πάρουμε:

```
 x = 10   r = 10
 x = 15   r = 15
```

Δηλαδή, εκείνο το "`r(x)`" στη δήλωση της *r* της δίνει όχι την τιμή της *x* αλλά τη θέση.

Μα κάτι τέτοιο δεν κάνουμε και με τα βέλη; Περίπου! Να πώς γίνονται τα ίδια πράγματα με βέλη:

```
   int* p( &x );
   *p = 10;
   cout << " x = " << x << "   *p = " << *p << endl;
   x = 15;
   cout << " x = " << x << "   *p = " << *p << endl;
```

Αποτέλεσμα:

```
 x = 10   *p = 10
 x = 15   *p = 15
```

Τι διαφορές έχουμε;

- Ως αρχική τιμή στην *p* δίνουμε την `&x` και όχι τη `x`.
- Ακόμη, κάθε φορά που χρησιμοποιούμε τη μεταβλητή μέσω της *p* πρέπει να κάνουμε αποπαραπομπή (`*p`).
- Τέλος, *στη δήλωση μιας μεταβλητής αναφοράς πρέπει να δίνουμε οπωσδήποτε αρχική τιμή*, ενώ με τα βέλη αυτό δεν είναι υποχρεωτικό.

Μια συνάρτηση με τύπο μπορεί να έχει ως τύπο επιστροφής έναν τύπο αναφοράς. Ας δούμε ένα

Παράδειγμα

Αντιγράφουμε και αλλάζουμε λίγο την *maxNdx* από την §9.3:

```
   int& maxElmn( int x[], int n )
   {
      int mxP( 0 );

      for ( int m(1); m < n; ++m )
      {
         if ( x[m] > x[mxP] )  mxP = m;
      } // for (m …

      return x[mxP];
   } // maxElmn
```

Η *maxElmn* επιστρέφει τη μέγιστη τιμή από αυτές των στοιχείων του πίνακα και όχι τη θέση του.

Πρόσεξε ότι:

- Ως τύπο επιστροφής βάλαμε `int&` και όχι `int`. Στη `return` όμως επιστρέφουμε την x[*mxP*], δηλαδή μια τιμή τύπου `int`.
- Η πρώτη παράμετρος είναι `int x[]` και όχι `const int x[]`. Διάβασε παρακάτω τα σχετικά.

Πώς καλούμε τη συνάρτηση; Όπως ξέρουμε. Αν έχουμε δηλώσει:

```
   int zm, z[] = { 12, 4, -3, 24, 33, 2, 4, 7 };
```

μπορούμε να γράψουμε:

```
   zm = maxElmn( z, 8 );
   cout << zm << "   " << maxElmnD(z, 8) << endl;
```

που θα μας δώσουν:

```
33   33
```

Δηλαδή τα πάντα δουλεύουν σαν να είχαμε τύπο επιστροφής `int` και όχι `int&`.

Γιατί βγάλαμε το "`const`"; Διότι δεν το δέχεται ο μεταγλωττιστής και να το πρόβλημα που έχει: Οι εντολές

```
for ( int k(0); k < 8; ++k )  cout << z[k] << "  ";
cout << endl;
maxElmn( z, 8 ) = 17;
for ( int k(0); k < 8; ++k )  cout << z[k] << "  ";
cout << endl;
```

θα δώσουν:

```
12  4  -3  24  33  2  4  7
12  4  -3  24  17  2  4  7
```

Δηλαδή, η "`maxElmn(z, 8) = 17`" όχι μόνον είναι δεκτή αλλά μας επιτρέπει να αλλάξουμε την τιμή του στοιχείου `z[4]`! Βάζοντας ως τύπο επιστροφής της συνάρτησης "`int&`" η συνάρτηση επιστρέφει *το στοιχείο* με τη μέγιστη τιμή (τιμή-*l*) και όχι απλώς την τιμή του. Παρακάτω θα διαβάσεις πώς γίνεται αυτό. Πώς διορθώνεται όμως; Έτσι:

```
const int& maxElmn( const int x[], int n )
```

και η παράξενη εκχώριση απαγορεύεται!

Και πού θα μας χρειασθούν αυτά τα μπερδεμένα πράγματα; Ας δούμε τι ακριβώς γίνεται και μετά θα απαντήσουμε. Όταν εκτελείται η κλήση της συνάρτησης, στην εντολή "`zm = maxElmn(z, 8)`", γίνονται τα εξής:

- υπολογίζεται η *mxP* και
- η `return x[mxP]` *αντιγράφει*, σε κάποια θέση της μνήμης, *ένα βέλος* προς το στοιχείο x[*mxP*].

Όταν στη συνέχεια εκτελείται η εκχώρηση, *στην zm αντιγράφεται η τιμή που δείχνει το βέλος*.

Αν είχαμε βάλει ως τύπο επιστροφής `int` αντί για `int&` τι θα γινόταν; Θα αντιγραφόταν, από τη `return`, η τιμή του x[*mxP*] και στη συνέχεια θα είχαμε άλλη μια αντιγραφή αυτής της τιμής κατά την εκχώρηση. Δηλαδή:

- αν έχουμε τύπο επιστροφής τον *T*`&` έχουμε μια αντιγραφή βέλους και μια αντιγραφή τιμής τύπου *T* ενώ
- αν έχουμε τύπο επιστροφής τον *T* έχουμε δύο αντιγραφές τιμών τύπου *T*.

«Σιγά τη διαφορά!» θα πεις και θα έχεις δίκιο αν *T* είναι ο `int`. Αν όμως οι τιμές τύπου *T* πιάνουν μερικά *kB* η κάθε μια τότε τα πράγματα αλλάζουν: βάζοντας ως τύπο της συνάρτησης τον *T*`&` έχουμε πιο γρήγορη εκτέλεση! Θα δούμε τέτοιους «μεγάλους» τύπους στη συνέχεια.

Τώρα όμως προσοχή! Θα μπορούσαμε να αλλάξουμε την *maxNdx* σε:

```
int& maxPosD( const int x[], int n )
{
   int mxP( 0 );

   for ( int m(1); m < n; ++m )
   {
      if ( x[m] > x[mxP] )  mxP = m;
   } // for (m …

   return mxP;
} // maxNdx
```

Όχι! Και να γιατί: Ας πούμε ότι είχαμε την εντολή:

```
p = maxNdx( z, 8 );
```

Η εκτέλεση της `return mxP` θα αντέγραφε ένα βέλος προς την *mxP*, *που είναι μια μεταβλητή τοπική στη συνάρτηση*. Όταν θα έφθανε η ώρα να εκτελεσθεί η αντιγραφή της τιμής στην *p*, η εκτέλεση της συνάρτησης θα είχε τελειώσει και *η mxP δεν θα υπήρχε*! Δίδαγμα:

- ***Αν ο τύπος επιστροφής μιας συνάρτησης είναι τύπος αναφοράς η συνάρτηση δεν θα πρέπει να επιστρέφει ως τιμή κάποιο τοπικό αντικείμενο.***

13.5 Η Εντολή `return` (ξανά)

Είπαμε στην §7.2 ότι: η **`return`** Π «*επιστρέφει την τιμή της Π αφού τη μετατρέψει στον τύπο της συνάρτησης*. Αλλά η **`return`** κάνει και κάτι άλλο: *τελειώνει την εκτέλεση της συνάρτησης στην οποία υπάρχει* και η εκτέλεση συνεχίζεται στο σημείο που κλήθηκε· οι εντολές που τυχόν την ακολουθούν δεν θα εκτελεσθούν.»

Δηλαδή, σε μια συνάρτηση χωρίς τύπο, που οι επιστροφές τιμών γίνονται μέσω των παραμέτρων, δεν μας χρειάζεται **`return`**; Ναι, και γι' αυτό δεν τη χρησιμοποιήσαμε. Πάντως υπάρχει και μορφή –χωρίς παράσταση– που απλώς τελειώνει την εκτέλεση της συνάρτησης· αυτή μπορεί να χρησιμοποιηθεί.

Παράδειγμα

Η παρακάτω συνάρτηση μας επιστρέφει την ελάχιστη (*mnxy*) και τη μέγιστη (*mxxy*) από δύο ακέραιες τιμές:

```
void minmax(int x, int y, int& mnxy, int& mxxy)
{
   if (x < y)  {  mnxy = x;  mxxy = y;  return;  }
   mnxy = y;  mxxy = x;
} // minmax
```

Αν ισχύει η συνθήκη $x < y$ θα εκτελεσθούν οι "**`mnxy = x; mxxy = y;`**" και στη συνέχεια, με την εκτέλεση της **`return`** τελειώνει η εκτέλεση της συνάρτησης· οι "**`mnxy = y; mxxy = x;`**", που ακολουθούν, δεν εκτελούνται. Αν δεν ισχύει η συνθήκη θα εκτελεσθούν μόνον οι "**`mnxy = y; mxxy = x;`**" και θα τελειώσει η εκτέλεση της συνάρτησης.

Όπως βλέπεις όμως, η χρήση της **`return`**, στις συναρτήσεις χωρίς τύπο, σημαίνει παράβαση του κανόνα «*μία είσοδος και μία έξοδος*». Για τον λόγο αυτόν θα αποφεύγουμε τη χρήση της.

13.6 Εμβέλεια και Χρόνος Ζωής Μεταβλητών

Όπως λέγαμε και στο Κεφ. 7,

> *«Ένα πρόγραμμα της C++ είναι ένα σύνολο από συναρτήσεις. Μια από αυτές τις συναρτήσεις έχει το όνομα* **`main`** *και από αυτήν αρχίζει η εκτέλεση του προγράμματος.*
>
> *Κάθε συνάρτηση είναι μια σύνθετη εντολή (σώμα της συνάρτησης) με μια επικεφαλίδα. Στην επικεφαλίδα, δηλώνεται και ένα όνομα που χαρακτηρίζει τη συνάρτηση. Η σύνθετη εντολή αποτελείται από τις δηλώσεις των σταθερών, των τύπων και των μεταβλητών και τις άλλες εντολές. Στις εντολές αυτές μπορεί να περιλαμβάνονται άλλες σύνθετες εντολές με το περιεχόμενο που περιγράφουμε (δηλ. δηλώσεις κλπ).»*

Τώρα θα σταθούμε σε δύο σημεία:

- Δηλώσεις μπορεί να υπάρχουν και έξω από οποιαδήποτε συνάρτηση (καθολικές).
- Δηλώσεις μπορεί να υπάρχουν και μέσα σε οποιαδήποτε σύνθετη εντολή (τοπικές στη σύνθετη εντολή).

Ξεκινούμε με την πρώτη περίπτωση ξαναγράφοντας το παράδειγμα που είδαμε πιο πριν κάπως διαφορετικά.

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

   const double  g = 9.81; // m/sec2, επιτάχυνση της βαρύτητας

   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   void inputH()
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
      if ( h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> h;  }
   } // inputH

   void compute()
   {
   // (g == 9.81) && (0 <= h <= DBL_MAX)
      tP = sqrt( (2/g)*h );
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   } // compute

   void displayResults()
   {
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   } // displayResults

int main()
{
   inputH();
   if ( h >= 0 )
   {
      compute();
      displayResults();
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως βλέπεις:

- οι συναρτήσεις δεν έχουν παραμέτρους και
- οι μεταβλητές έχουν δηλωθεί έξω από τις συναρτήσεις.

Λέμε ότι, στην περίπτωση αυτή, οι μεταβλητές είναι **καθολικές** (global)· *μπορεί να τις χρησιμοποιεί και να αλλάζει τις τιμές τους οποιαδήποτε συνάρτηση.*

Οποιαδήποτε; Και αν έχω, ας πούμε, μια καθολική μεταβλητή *x* και μια μεταβλητή *x* τοπική σε κάποια συνάρτηση τότε τι γίνεται; Όσο εκτελείται η συνάρτηση, με το όνομα *x*, ξέρει την τοπική μεταβλητή. Μπορεί όμως να δει και την καθολική *x* ως "`::x`". Αν έχουμε για παράδειγμα:

```
   int x;

   void q( double x )
   {
      cout << x << "  " << ::x << endl;
//       . . .
   } // q

int main()
{
// . . .
```

```
   x = 4;  q( 2.5 );
// . . .
}
```

Στη **main**, η *καθολική* μεταβλητή *x* παίρνει τιμή 4 και στη συνέχεια καλείται η *q* με πραγματική παράμετρο 2.5. Αυτή γίνεται τιμή της τυπικής παραμέτρου *x*, που είναι *τοπική* στην *q*. Όταν εκτελεσθεί η εντολή `cout << x << "  " << ::x << endl` θα καταλάβει ως *x* την τοπική παράμετρο, ενώ θα μας δώσει και την τιμή της καθολικής από την `::x`. Έτσι, θα πάρουμε:

```
2.5  4
```

Εκτός από αυτό το χαρακτηριστικό, της «καθολικής ορατότητας», οι καθολικές μεταβλητές έχουν και ένα άλλο ενδιαφέρον χαρακτηριστικό: είναι **στατικές**, δηλαδή ζουν –άρα κρατούν την τιμή τους– από την αρχή μέχρι το τέλος της εκτέλεσης του προγράμματος.

Τώρα θα ξαναδούμε τα περί τοπικών μεταβλητών. Αλλά πριν προχωρήσεις ξαναδιάβασε την §11.1.

Για τη C++, μια **ομάδα** (block) είναι μια **σύνθετη εντολή** (compound statement), δηλαδή ό,τι υπάρχει ανάμεσα σε ένα ζευγάρι "`{`" και "`}`". Όπως έχουμε δει, μπορεί να υπάρχει *ομάδα μέσα σε ομάδα*.

Σε ένα πρόγραμμα C++:

- Μπορείς να δηλώσεις μια μεταβλητή σε οποιοδήποτε σημείο, αλλά πριν τη χρησιμοποιήσεις.
- Η εμβέλειά της ξεκινάει από το σημείο που έγινε η δήλωση και τελειώνει στο τέλος της πιο εσωτερικής ομάδας που περιλαμβάνει τη δήλωση.

Ας δούμε τι ακριβώς συμβαίνει με ένα παράδειγμα. Δες το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   { // ομάδα 1.1
      int x( 5 );
      cout << "p2: x = " << x << endl;
   }
   cout << "p3: x = " << x << endl;
// . . .
```

Αν προσπαθήσεις να το μεταγλωττίσεις θα πάρεις μήνυμα:

```
Error … line 9: Undefined symbol 'x' in function main()
```

Τι συμβαίνει; Η γραμμή 9 (`cout << "p3: x = " << x << endl`) βρίσκεται στην ομάδα 1, όπου δεν έχει δηλωθεί μεταβλητή. Η *x* έχει δηλωθεί στην ομάδα 1.1, είναι γνωστή μέσα σ' αυτήν –έτσι δεν υπάρχει πρόβλημα με την εντολή "`cout << "p2: x = " << x << endl`"– αλλά δεν είναι γνωστή στη γραμμή 9 που βρίσκεται έξω από την ομάδα (που τελειώνει στη γραμμή 8).

Οι εντολές μιας ομάδας μπορεί να βλέπουν αντικείμενα που δηλώνονται σε ομάδες που την περιβάλλουν. Έστω π.χ. το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   int y;

   { // ομάδα 1.1
      y = 19;
      cout << "p1: y = " << y << endl;
   }
   cout << "p2: y = " << y << endl;
   . . .
```

που δίνει:

Πλαίσιο 13.1

Κανόνας Εμβέλειας

Κάθε δήλωση ισχύει στη σύνθετη εντολή όπου έγινε και στις σύνθετες εντολές που είναι εσωτερικές σε αυτήν. Δεν ισχύει στις εσωτερικές σύνθετες εντολές που υπάρχει άλλος ορισμός (άλλη δήλωση) για το ίδιο όνομα. Οι παράμετροι μιας συνάρτησης είναι σαν να δηλώνονται στην ομάδα που ακολουθεί την επικεφαλίδα.

Το ίδιο πράγμα με άλλα λόγια:

Σε κάθε σύνθετη εντολή είναι «ορατά» όλα τα αντικείμενα που δηλώνονται σε αυτήν –τοπικά (local)– και ακόμη όλα αυτά που είναι «ορατά» στην ομάδα που την περιβάλλει –καθολικά (global). Από αυτά που έρχονται από το περιβάλλον δεν είναι «ορατά» αυτά που έχουν το ίδιο όνομα με κάποιο τοπικό αντικείμενο.

Κανόνας Διάρκειας Ζωής

Κάθε αντικείμενο του προγράμματός μας υπάρχει όσο εκτελείται η σύνθετη εντολή όπου έχει δηλωθεί. Τα καθολικά και αυτά που έχουν δηλωθεί με προδιαγραφή `static` ***υπάρχουν από την αρχή μέχρι το τέλος της εκτέλεσης του προγράμματος.***

```
p1: y = 19
p2: y = 19
```

Και οι δύο εντολές της ομάδας 1.1 διαχειρίζονται μια μεταβλητή, την *y*, που δηλώνεται στην ομάδα 1. Η δεύτερη γραμμή των αποτελεσμάτων μας (`p2: y = 19`) δείχνει ότι η τιμή που εκχωρήθηκε στην *y* στην ομάδα 1.1 διατηρείται και όταν τελειώσει η εκτέλεση αυτής της ομάδας.

Τι γίνεται όμως όταν ένα όνομα, ας πούμε *x*, χρησιμοποιείται σε κάποια ομάδα και έξω από αυτήν; Οι εντολές που υπάρχουν στην ομάδα, με το *x* καταλαβαίνουν το αντικείμενο που έχει δηλωθεί *μέσα* στην ομάδα. Για παράδειγμα, το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   int x( 1 );

   cout << "p1: x = " << x << endl;
   { // ομάδα 1.1
      const int x( 5 );
      cout << "p2: x = " << x << endl;
   }
   cout << "p3: x = " << x << endl;
}
```

μας δίνει:

```
p1: x = 1
p2: x = 5
p3: x = 1
```

Η γραμμή "`p1: x = 1`" προέρχεται από την εντολή εκτύπωσης: "`cout << "p1: x = " << x << endl`", που «βλέπει» την αρχική δήλωση "`int x = 1`", της ομάδας 1.

Η "`p2: x = 5`" προέρχεται από τη "`cout << "p1: x = " << x << endl`" που υπάρχει στην ομάδα 1.1. Η σταθερά *x* αυτής της ομάδας δεν έχει σχέση με τη μεταβλητή *x* που δηλώσαμε στην ομάδα 1. Οι εντολές της ομάδας 1.1 «βλέπουν» μόνον τη «δική» τους *x* ενώ δεν έχουν δυνατότητα να «δουν» τη *x* της ομάδας 1.

Η γραμμή "`p3: x = 1`" προέρχεται από την εντολή εκτύπωσης: "`cout << "p3: x = " << x << endl`", που ανήκει στην ομάδα 1 και «βλέπει» την αρχική δήλωση "`int x = 1`", της ομάδας 1.

Και τι συμβαίνει με τη «ζωή» των αντικειμένων: Τα αντικείμενα που δηλώνονται σε μια ομάδα «ζουν», υπάρχουν, όσο εκτελούνται οι εντολές της ομάδας αυτής. Π.χ. η μεταβλητή *x* που δηλώνεται στην ομάδα 1 ζει όσο εκτελείται η ομάδα της **main**. Η σταθερά *x* που δηλώνεται στην ομάδα 1.1 ζει όσο εκτελείται η ομάδα 1.1 ή –αφού πρόκειται για έναν ορισμό– αυτός ο ορισμός ισχύει μόνον όσο εκτελείται η ομάδα 1.1. Όσο εκτελείται η ομάδα 1.1 το πρόγραμμά μας έχει στη διάθεσή του μια θέση μνήμης για τη μεταβλητή *x* της ομάδας 1 και μια θέση για τη σταθερά της ομάδας 1.1. Όταν έρχεται η ώρα να εκτελεστεί η εντολή "`cout << "p3: x = " << x << endl`", που βρίσκεται μετά το τέλος της ομάδας 1.1, η σταθερά *x* δεν υπάρχει· έχουμε μείνει μόνον με τη μεταβλητή που είχαμε αρχικά.

Όλα αυτά υπάρχουν μαζεμένα στο Πλ. 13.1.

Υπάρχουν και δύο ειδικές περιπτώσεις:

1. Οι παράμετροι μιας συνάρτησης έχουν εμβέλεια όλο το σώμα της συνάρτησης και χρόνο ζωής το χρόνο εκτέλεσης της συνάρτησης.
2. Όπως είδαμε, μπορούμε να δηλώσουμε μεταβλητές στην αρχική εντολή της **for**. Στην περίπτωση αυτή η εμβέλεια των μεταβλητών εκτείνεται μέχρι και την τελευταία επαναλαμβανόμενη εντολή και χρόνος ζωής μέχρι το τέλος της εκτέλεσης της επαναλαμβανόμενης εντολής.

Μπορούμε να έχουμε τοπικές μεταβλητές που ζούν σε όλη τη διάρκεια της εκτέλεσης του προγράμματος; Ναι! Διάβασε την επόμενη παράγραφο.

13.6.1 * Στατικές Μεταβλητές

Αν θέλεις «να κρατάς πάντοτε στη ζωή» κάποιο αντικείμενο μιας συνάρτησης μπορείς να το δηλώσεις ως `static`, όπως δηλώνουμε την τοπική μεταβλητή *a* στην παρακάτω συνάρτηση[3]:

```
double rnd01()
{
   static double a( 0.13579246801357924680 );

   cout << " rnd01 arxh: a = " << a << endl;
   a = 37*a;
   a = a - static_cast<int>(a);
   cout << " rnd01 telos: a = " << a << endl;
   return a;
} // rnd01
```

Ας πούμε ότι έχουμε και την:

```
double f( double x )
{
   double z;

   cout << " f arxh: z = " << z << endl;
   z = pow( x, 0.25 );
   cout << " f telos: z = " << z << endl;
   return x + z;
} // f
```

που κι αυτή έχει μια τοπική μεταβλητή, τη *z*, που όμως δεν δηλώνεται `static`. Και στις δύο συναρτήσεις έχουμε βάλει εντολές που τυπώνουν τις τιμές των τοπικών μεταβλητών στην αρχή και στο τέλος της εκτέλεσής τους. Ζητούμε το εξής:

```
   for ( int k(1); k <= 3; ++k )
```

[3] Πρόκειται για μια απλοϊκή μέθοδο για την παραγωγή ψευδοτυχαίων αριθμών στο διάστημα (0,1).

```
        cout << k << "  " << f(rnd01()) << endl;
```

και να τι παίρνουμε:

```
 rnd01 arxh: a = 0.135792
 rnd01 telos: a = 0.0243213
 f arxh: z = 4.97035e-219
 f telos: z = 0.394909
1  0.41923
 rnd01 arxh: a = 0.0243213
 rnd01 telos: a = 0.899889
 f arxh: z = 4.97035e-219
 f telos: z = 0.973974
2  1.87386
 rnd01 arxh: a = 0.899889
 rnd01 telos: a = 0.295882
 f arxh: z = 4.97035e-219
 f telos: z = 0.73753
3  1.03341
```

Βλέπουμε λοιπόν ότι, κατά την πρώτη κλήση της *rnd*01, η *a* έχει στο τέλος τιμή 0.0243213 και αυτή ακριβώς είναι η τιμή της στην αρχή της δεύτερης κλήσης. Κατά την πρώτη κλήση της *f*, στο τέλος, η *z* έχει τιμή 0.394909 αλλά στην αρχή της δεύτερης κλήσης έχει τιμή 4.97035*e*–219.

Πώς εξηγούνται αυτά; Η *z* είναι μια **αυτόματη** (automatic) μεταβλητή που δημιουργείται όταν ενεργοποιείται η *f* και εξαφανίζεται όταν τελειώνει τη δουλειά της. Η *a* ως στατική συνεχίζει να υπάρχει και μετά το τέλος της εκτέλεσης της *rnd*01 και έτσι δεν χάνει την τιμή της.

Οι στατικές μεταβλητές έχουν και μια άλλη ιδιότητα: αν δεν τους δοθεί αρχική τιμή με τη δήλωση παίρνουν την *ερήμην καθορισμένη* (default) τιμή του τύπου τους. Για όλους τους αριθμητικούς τύπους της C++ και για τα βέλη είναι η τιμή 0 (μηδέν).

13.6.2 Καθολικά Αντικείμενα και Τεκμηρίωση

Απόφευγε όσο μπορείς τις καθολικές μεταβλητές. Αν τις χρησιμοποιείς θα γράφεις προγράμματα που δεν είναι ευκολο να

- Επαληθευθούν
- Τροποποιηθούν.

Πάντως αυτό δεν θα είναι πάντοτε εύκολο. Διάβασε λοιπόν αυτά που λέμε παρακάτω για τις συναρτήσεις και τις μεταβλητές.

Οι συναρτήσεις είναι *καθολικά* αντικείμενα. Έτσι, κάθε συνάρτηση μπορεί να καλεί άλλες συναρτήσεις και να καλείται από άλλες συναρτήσεις. Σκέψου λοιπόν το πρόβλημα που έχεις όταν χρειάζεται να τροποποιήσεις κάποια συνάρτηση. Θα πρέπει να πας σε όλες τις συναρτήσεις που την καλούν και να δεις τι πρόβλημα θα δημιουργηθεί.

Καταλαβαίνεις λοιπόν ότι: ουσιώδες τμήμα της τεκμηρίωσης ενός προγράμματος είναι η καταγραφή των αλληλεξαρτήσεων των συναρτήσεων.

♦ ***Για κάθε συνάρτηση θα πρέπει να είναι καταγραμμένο: από ποιες καλείται και ποιες καλεί.***

Αν έλθουμε τώρα στις καθολικές μεταβλητές, ο αντίστοιχος κανόνας τεκμηρίωσης είναι ο εξής:

♦ ***Για κάθε καθολική μεταβλητή θα πρέπει να είναι καταγραμμένο: ποιες συναρτήσεις τη χρησιμοποιούν και ποιες συναρτήσεις αλλάζουν την τιμή της.***

13.7 * Οι Συναρτήσεις στις Αποδείξεις (ξανά)

Στην §7.9 είχαμε δει το τι και πώς πρέπει να αποδείξουμε όταν έχουμε μια συνάρτηση με τύπο με παραμέτρους τιμής μόνον. Τώρα θα δούμε τα ίδια πράγματα για την περίπτωση που έχουμε συνάρτηση χωρίς τύπο (`void`) με παραμέτρους τιμής και αναφοράς.

Ξεκινούμε με δύο παραδείγματα:

Παράδειγμα 1

Ας πάρουμε τη *minmax()* που τροφοδοτείται με δύο ακέραιους *x*, *y* και μας επιστρέφει τον ελάχιστο και τον μέγιστο από τους δύο:

```
   void minmax( int x, int y, int& mnxy, int& mxxy )
   {
      if ( x < y )
      {
         mnxy = x;  mxxy = y;
      }
      else
      {
         mnxy = y;  mxxy = x;
      }
   } // minmax
```

Αν τώρα βάλουμε σε μια συνάρτηση την εντολή:

```
   minmax( a, b, p, q );
```

τι περιμένουμε; Όποιες και να είναι οι *a*, *b* αρχικά, θα πρέπει μετά από αυτήν να έχουμε την τιμή της μικρότερης από αυτές στην *p* και την τιμή της μεγαλύτερης στην *q*. Να λοιπόν οι προδιαγραφές της *minmax()*:

```
// true
   minmax( a, b, p, q );
// ((p == a && q == b) || (p == b && q == a)) &&
// (p <= a && p <= b) && (q >= a && q >= b)
```

Όπως ξέρουμε, στις τυπικές παραμέτρους τιμής, *x*, *y*, θα αντιγραφούν οι τιμές των πραγματικών παραμέτρων *a*, *b*, ενώ οι *mnxy*, *mxxy* αναφέρονται στις ίδιες θέσεις της μνήμης με τις *p*, *q*. Άρα, μέσα στη συνάρτηση θα πρέπει να αποδείξουμε ότι:

```
// true
   if (x < y)
   {
      mnxy = x;  mxxy = y;
   }
   else
   {
      mnxy = y;  mxxy = x;
   }
// ((mnxy == x && mxxy == y) || (mnxy == y && mxxy == x)) &&
// (mnxy <= x && mnxy <= y) && (mxxy >= x && mxxy >= y)
```

Αυτό δεν διαφέρει και πολύ από αυτό που είχαμε να αποδείξουμε όταν είχαμε συνάρτηση με τύπο. Απλώς εδώ υπολογίζουμε δύο τιμές, αντί για μία.

Παράδειγμα 2

Ας δούμε τώρα τη *swap()*, που δέχεται δύο ορίσματα (τη γράφουμε για τον τύπο `int`) και αντιμεταθέτει τις τιμές τους:

```
   void swap( int& x, int& y )
   {
      int s( x );
      x = y;  y = s;
   } // swap
```

Ποιες είναι οι προδιαγραφές της *swap()*; Μπορούμε να τις διατυπώσουμε εύκολα (ρίξε μια ματιά και στο Κεφ. 3):

```
// a == a0 && b == b0
   swap( a, b );
// a == b0 && b == a0
```

Επομένως στη συνάρτηση θα πρέπει να αποδείξουμε:

```
// x == a0 && y == b0
   int s( x );  x = y;  y = s;
// x == b0 && y == a0
```

☜☜☜

Συγκρίνοντας τα δύο παραδείγματα βλέπουμε ότι όταν έχουμε εισερχόμενες και εξερχόμενες τιμές μπορούμε να γράψουμε προδιαγραφές με τα ονόματα των παραμέτρων μόνον. Όταν όμως έχουμε μεταβαλόμενες τιμές θα πρέπει να χρησιμοποιήσουμε στις προδιαγραφές μας και τις αρχικές και τελικές τιμές των παραμέτρων αναφοράς.

Ας υποθέσουμε ότι έχουμε μια συνάρτηση, `void p`, με *n* παραμέτρους τιμής, *v1*, *v2*,... *vn* και *m* παραμέτρους αναφοράς *r1*, *r2*,... *rm*. Αν ονομάσουμε *r1α*, *r2α*,... *rmα* τις αρχικές τιμές των παραμέτρων αναφοράς και *r1τ*, *r2τ*,... *rmτ* τις τελικές τιμές τους, οι προδιαγραφές της *p* μπορεί να διατυπωθούν ως εξής:

```
// Pd(p1, p2,... pn, q1, q2,... qm) &&
// (q1 == r1α) && (q2 == r2α) && ... (qm == rmα)
   p(v1, v2,... vn, r1, r2,... rm);
// Qd(p1, p2,... pn, r1α, r2α,... rmα, q1τ, q2τ,... qmτ)
```

Όταν την καλούμε, την τροφοδοτούμε με τις τιμές που έχουν οι παράμετροι τιμής και πιθανότατα κάποιες από τις παραμέτρους αναφοράς. Αν υποθέσουμε ότι οι τιμές των παραμέτρων τιμής δεν αλλάζουν, μπορούμε να γράψουμε:

```
   void p(T1 v1, T2 v2,... Tn vn, Tr1& r1, Tr2& r2,... Trm& rm)
   {
   // Pd(v1, v2,... vn, r1, r2,... rm) &&
   // (r1 == r1α) && (r2 == r2α) && ... (rm == rmα)
         :
   // Qd(v1, v2,... vn, r1α, r2α,... rmα, r1, r2,... rm)
   } // p
```

13.8 Ορμαθοί C και Αριθμοί (ξανά)

Πριν προχωρήσουμε παρακάτω, θα κάνουμε μια παρένθεση για να «εξοφλήσουμε κάποια παλιά χρέη». Στην §10.13.1 είχαμε πει ότι για μετατροπές ορμαθών σε αριθμούς «*πιο πλήρη δουλειά, σε περίπτωση λάθους, κάνουν οι strtod() και strtol() που θα δούμε αργότερα.*» Τώρα μπορούμε να τις δούμε.

Ας πούμε ότι δίνουμε:

```
   d = strtod( s3, &p );
```

όπου *s3* αριθμητικός ορμαθός και "`char* p`". Αν ολόκληρος ο ορμαθός *s3* μετατραπεί σε ακέραιο, αυτός θα γίνει τιμή της *d* και η τιμή της *p* θα είναι (βέλος) 0. Αν βρεθούν απαράδεκτοι χαρακτήρες το βέλος *p* θα δείχνει έναν (υπο) ορμαθό που ξεκινάει με τον πρώτο από αυτούς.

Ας πούμε ότι έχουμε δηλώσει:

```
   double d;
   char  s1[] = "12345", s2[] = "1.23456789e10",
         s3[] = "1.23ab45678";
   char* p;
```

Οι παρακάτω εντολές:

```
   d = strtod( s2, &p );
   if ( p != 0 )  cout << d << "  " << p << endl;
            else  cout << d << endl;
   d = strtod( s3, &p );
   if ( p != 0 )  cout << d << "  " << p << endl;
```

```
            else  cout << d << endl;
```

θα δώσουν:

```
1.23457e+010
1.23  ab45678
```

Πώς είναι δηλωμένη η *strtod()* (στο **cstdlib**); Έτσι:

```
double strtod( const char* s, char** endptr );
```

Δεν έχεις πρόβλημα να καταλάβεις την πρώτη παράμετρο: Βέλος προς πίνακα χαρακτήρων. Η δεύτερη όμως; Μέσω της δεύτερης η συνάρτηση θα επιστρέψει ένα βέλος, είτε 0 (**NULL**) είτε προς τον πρώτο παράνομο χαρακτήρα. Στην §13.2 είδαμε πώς επιστρέφει η C τιμές μέσω παραμέτρων: αν η τιμή είναι τύπου *T*, βάζουμε μια παράμετρο τύπου *T** και εκεί –όταν καλούμε τη συνάρτηση– αντιστοιχίζουμε μια διεύθυνση της μνήμης όπου θα αποθηκευτεί η επιστρεφόμενη τιμή. Στην περίπτωσή μας θα επιστραφεί μια τιμή τύπου **char***· θα πρέπει λοιπόν να βάλουμε μια παράμετρο τύπου **(char*)***. Όταν καλούμε τη συνάρτηση θέλουμε να αποθηκεύσουμε θέλουμε η επιστρεφόμενη τιμή να αποθηκευτεί στην *p* (που είναι τύπου **char***). Δίνουμε λοιπόν την διεύθυνση της *p*, δηλαδή **&p**.

Αν κατάλαβες αυτό το σημείο, τα υπόλοιπα είναι απλά.

Σαν τη *strtod()* είναι και οι:

```
float strtof( const char* s, char** endptr );
long double strtold( const char* s, char** endptr );
```

Μπορείς να βλέπεις την *atof()* (§10.13.1) ως

```
double atof( const char* s )
{  return strtod( s, NULL );  }
```

Παρομοίως δουλεύει και η *strtol()*, που είναι δηλωμένη ως εξής:

```
long strtol( const char* s, char** endptr, int radix )
```

Όπως βλέπεις, αυτή έχει και μια τρίτη παράμετρο όπου πρέπει να βάλουμε τη βάση του αριθμητικού συστήματος στο οποίο είναι γραμμένος ο ακέραιος που παριστανει ο ορμαθός. Π.χ. οι:

```
   l = strtol( "11", &p,  2 );   cout << l << endl;
   l = strtol( "11", &p,  8 );   cout << l << endl;
   l = strtol( "11", &p, 10 );   cout << l << endl;
   l = strtol( "11", &p, 16 );   cout << l << endl;
```

δίνουν:

```
3
9
11
17
```

(3 = 2 + 1, 9 = 8 + 1, 11 = 10 + 1, 17 = 16 + 1)

Ακόμη, οι:

```
   long int l( strtol(s1, &p, 10) );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
   l = strtol( s2, &p, 10 );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
```

δίνουν:

```
12345
1  .23456789e10
```

Τα παραπάνω σου δείχνουν και τον τρόπο χρήσης των δύο συναρτήσεων: αφού τις καλέσεις ελέγχεις το *p*:

- Αν είναι 0 όλα πήγαν καλά, δηλαδή ολόκληρος ο ορμαθός μετατράπηκε στην αριθμητική τιμή που σου επίστρεψε τη συνάρτηση και μπορείς να τη χρησιμοποιήσεις.

- Αν δεν είναι 0 τότε δεν έγινε πλήρης μετατροπή διότι υπήρχε ένας τουλάχιστον παράνομος χαρακτήρας. Το *p* σου δείχνει τον πρώτο (`*p`).

Όλα καλά εκτός από την περίπτωση που έχουμε **υπερχείλιση** (overflow)! Τι γίνεται στην περίπτωση αυτή; Σου έρχεται ένα μήνυμα από την τιμή που επιστρέφει η συνάρτηση και την τιμή μιας καθολικής μεταβλητής με το όνομα "`errno`". Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cerrno>`".

- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μεγαλύτερη από `DBL_MAX` αυτή επιστρέφει τιμή `HUGE_VAL` (ορίζεται στο `cmath`) και βάζει στην *errno* τιμή `ERANGE` (ορίζεται στο `cerrno`).
- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μικρότερη από `-DBL_MAX` αυτή επιστρέφει τιμή `-HUGE_VAL` και βάζει στην *errno* τιμή `ERANGE`.[4]

Αν λοιπόν μετά τις

```
errno = 0;
d = strtod( s2, &p );
```

ισχύει η συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

δεν έχει νόημα να χρησιμοποιήσεις την τιμή της *d*.

Παρομοίως, αν μετά τις

```
errno = 0;
l = strtol(s2, &p, 10)
```

η συνθήκη

```
(l == LONG_MAX || l == LONG_MIN) && errno == ERANGE
```

έχει τιμή `true` μην χρησιμοποιήσεις την τιμής *l*.

Για την *errno* θα τα ξαναπούμε.

Παρόμοιες με τη *strtol()* είναι και οι

```
long long int strtoll( const char* s, char** endptr, int base );
unsigned long int strtoul( const char* s, char** endptr,
                                                    int base );
unsigned long long int strtoull( const char* s, char** endptr,
                                                    int base );
```

ενώ οι *atol()* (§10.13.1), *atoi()*, *atoll()* είναι ισοδύναμες με

```
long int atol( const char* s )
{  return strtol( s, NULL, 10 );  }
int atoi( const char* s )
{  return int( strtol(s, NULL, 10) );  }
long long int atoll( const char* s )
{  return strtoll( s, NULL, 10 );  }
```

13.9 Πώς Επιλέγουμε το Είδος της Συνάρτησης

Μάθαμε λοιπόν για συναρτήσεις με τύπο και χωρίς τύπο, για παραμέτρους βέλη, τιμής και αναφοράς, για καθολικές μεταβλητές... Πώς τα χρησιμοποιούμε όλα αυτά;

Παρακάτω δίνουμε μερικούς κανόνες για το πώς επιλέγουμε το είδος της συνάρτησης. Θα πρέπει να τηρείς αυτούς τους κανόνες μέχρι να γίνεις μεγάλος(-η) προγραμματιστής(-στρια)· τότε θα ξέρεις πότε και πώς να τους παραβιάζεις.

Πριν από όλα λοιπόν θα βάλουμε τον εξής κανόνα:

♦ ***Όποτε μπορούμε να γράψουμε συνάρτηση με τύπο θα γράφουμε συνάρτηση με τύπο.***

[4] Σε περίπτωση υπερχείλισης η *strtof* επιστρέφει *HUGE_VALF* ή *–HUGE_VALF* ενώ η *strtold* θα δώσει *HUGE_VALL* ή *–HUGE_VALL*.

Αυτός ο κανόνας μπορεί να θεωρηθεί ως ειδική περίπτωση ενός γενικότερου κανόνα που λέει:

- ***Κάθε συνάρτηση θα πρέπει να έχει τον ελάχιστο καλά καθορισμένο υπολογιστικό στόχο***

Δηλαδή, όχι «αφού γράφουμε μια συνάρτηση να τα βάλουμε όλα μέσα!» Στον κανόνα αυτόν θα επανέλθουμε αργότερα με παραδείγματα.

1ος Κανόνας:

- ***Κάθε συνάρτηση επικοινωνεί –με τη συνάρτηση που την καλεί– μόνο με το όνομά της και με τις παραμέτρους της.***

Με άλλα λόγια: *μη γράφεις συναρτήσεις που να επικοινωνούν μέσω καθολικών μεταβλητών*.

Όταν πρόκειται να γράψεις μια συνάρτηση θα πρέπει, πριν από όλα, να καταγράψεις:

- τις τιμές που θα εισάγονται σε αυτήν (εισερχόμενες),
- τις τιμές που θα εξάγονται από αυτήν (εξερχόμενες),
- τις τιμές που θα μεταβάλλονται από αυτήν (διπλής κατεύθυνσης).

Με βάση αυτήν την καταγραφή αποφασίζεις το είδος της συνάρτησης που θα γράψεις:

2ος Κανόνας:

- ***Αν η συνάρτηση δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή τότε γράφεις συνάρτηση με τύπο.***

Στην περίπτωση αυτήν η εξερχόμενη τιμή θα αποδίδεται με την κλήση (όνομα) της συνάρτησης και τύπος της συνάρτησης είναι ο τύπος της εξερχόμενης τιμής. Είναι φανερό ότι οποιαδήποτε συνάρτηση με τύπο μπορείς να την υλοποιήσεις και ως συνάρτηση χωρίς τύπο με μια εξερχόμενη τιμή. Αλλά η συνάρτηση με τύπο είναι σαφώς πιο εύχρηστη και επομένως προτιμότερη.

3ος Κανόνας:

- ***Αν η συνάρτηση έχει μια ή περισσότερες μεταβαλλόμενες τιμές ή περισσότερες από μία εξερχόμενες τιμές τότε γράφεις συνάρτηση χωρίς τύπο*** `(void)`.

Στην περίπτωση αυτή οι μεταβαλλόμενες τιμές επιστρέφουν μέσω παραμέτρων. Ο κανόνας αυτός είναι συμπλήρωση του 2ου κανόνα· μια διαφορετική διατύπωση είναι η εξής:

- ***Μη γράφεις συνάρτηση με τύπο με παραμέτρους αναφοράς που να υλοποιούν επικοινωνία για μεταβαλλόμενες τιμές ή εξερχόμενες τιμές.***

13.9.1 Περί Παραμέτρων

Ας έλθουμε τώρα στις παραμέτρους. Άλλες γλώσσες προγραμματισμού επιτρέπουν προσδιορισμό των παραμέτρων σχετικό με τη χρήση τους. Η Ada, για παράδειγμα, έχει παραμέτρους, όπως τις δώσαμε πιο πάνω:

- **in** για εισαγωγή δεδομένων στη συνάρτηση,
- **out** για εξαγωγή τιμών από τη συνάρτηση και
- **in out** για τιμές που μεταβάλλονται από τη συνάρτηση.

Αξίζει να επισημάνουμε εδώ ότι η Ada στα υποπρογράμματα που αντιστοιχούν στις συναρτήσεις με τύπο της C++ επιτρέπει παραμέτρους **in** μόνον![5]

[5] Σου θυμίζει αυτό κάτι από τους κανόνες μας;...

Η C έχει ένα είδος παραμέτρων: *παραμέτρους τιμής*. Με αυτές περνάς δεδομένα προς τη συνάρτηση. Αν τώρα θέλεις να πάρεις τιμές από τη συνάρτηση μέσω παραμέτρων εισάγεις προς αυτήν ένα βέλος που δείχνει τη διεύθυνση όπου περιμένεις το «εξαγόμενο».

Η C++ έκανε ένα βήμα σε σχέση με τη C αλλά όχι μεγάλο: μετονόμασε το βέλος σε *αναφορά* και μας έδωσε την ευκολία να το χειριζόμαστε χωρίς να κάνουμε αποπαραπομπή, που γίνεται αυτομάτως. Παρ' όλα αυτά ο προσδιορισμός των παραμέτρων έχει σχέση με την υλοποίηση:

1. **Παράμετροι τιμής** (value parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση χωρίς να επιστρέφουν τις οποιεσδήποτε αλλαγές στη συνάρτηση που καλεί. Με αυτές υλοποιούμε παραμέτρους για εισερχόμενες τιμές (**in**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο τιμής μπορεί να είναι γενικά μια παράσταση.
2. **Παράμετροι αναφοράς** (reference parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση και γνωστοποιούν στη συνάρτηση που καλεί τις όποιες αλλαγές έγιναν στην καλούμενη συνάρτηση. Με αυτές υλοποιούμε παραμέτρους για εξερχόμενες (**out**) και μεταβαλλόμενες τιμές (**in out**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο αναφοράς μπορεί να είναι μεταβλητή ή στοιχείο πίνακα ή πίνακας (αν η τυπική παράμετρος είναι πίνακας).
3. **Παράμετροι-πίνακες** που είναι μεν παράμετροι τιμής αλλά είναι παράμετροι-βέλη. Επομένως, όπως έχουμε δει στην §9.4, είναι παράμετροι **in out**. Τις βάζουμε χωριστά διότι μπορείς να τις χειρίζεσαι μέσα στη συνάρτηση ως πίνακες χωρίς αποπαραπομπές. Αν θέλεις να έχεις έναν πίνακα ως παράμετρο **in** προτάσσεις τον περιορισμό `const`.

Προς το παρόν τα μόνα «μεγάλα» αντικείμενα που έχουμε δει είναι οι πίνακες και ξέρουμε πώς να τους χειριστούμε. Στη συνέχεια, όταν μάθουμε για κλάσεις, θα δούμε και άλλα. Αν θέλουμε να περάσουμε ένα μεγάλο αντικείμενο (ας πούμε τύπου *T*) ως παράμετρο **in**, η παράμετρος τιμής δεν είναι καλή λύση· η διαδικασία της αντιγραφής καθυστερεί την εκτέλεση του προγράμματος και φορτώνει μια περιοχή της μνήμης (τη στοίβα) που είναι μάλλον περιορισμένη. Στην περίπτωση αυτήν προτιμούμε να χρησιμοποιήσουμε παράμετρο αναφοράς (*T*`&`) με τον περιορισμό `const` (`const` *T*`&`).

Οι προγραμματιστές που έγραφαν παλιότερα C προτιμούν να χρησιμοποιούν αντί για παραμέτρους αναφοράς **παραμέτρους βέλη** (pointers). Ένας λόγος για την προτίμησή τους είναι ότι όταν διαβάζει κάποιος μια *κλήση συνάρτησης* καταλαβαίνει αμέσως ποιες παράμετροι είναι εισερχόμενες και ποιες διπλής κατεύθυνσης.

13.9.2 Παράμετρος – Ρεύμα

Είπαμε πριν (§11.6) ότι οι εντολές εισόδου/εξόδου είναι εντολές-παραστάσεις (πράξεις). Ας προσπαθήσουμε τώρα να το καταλάβουμε.

Κατ' αρχάς να υπενθυμίσουμε ότι, όπως μάθαμε στο Μέρος Α (§8.2), ένα ρεύμα έχει έναν **ενταμιευτή** (buffer), δηλαδή μια περιοχή μνήμης, όπου γίνεται ενδιάμεση αποθήκευση των δεδομένων του αρχείου που διαβάζουμε (ή γράφουμε). Φυσικά, χρειαζόμαστε και έναν δείκτη που δείχνει σε ποια θέση του ενταμιευτή διαβάζουμε, άλλους δείκτες που μας λένε ποια περιοχή του αρχείου έχουμε στον ενταμιευτή κλπ. Κάθε πράξη εισόδου/εξόδου μεταβάλλει την κατάσταση του ενταμιευτή και ολόκληρου του ρεύματος και η επόμενη πράξη εισόδου/εξόδου γίνεται πάνω σε αυτήν τη νέα κατάσταση.

Δες πώς αντανακλώνται αυτά τα πράγματα στο πρόγραμμα με ένα παράδειγμα σε γνωστά πράγματα. Ας πούμε ότι έχουμε την εντολή (δήλωσε τις *x*, *r* όπως θέλεις και δώσε όποιες τιμές θέλεις):

```
cout << " x = " << x << "   r = " << r << endl;
```

Δοκιμάζουμε να τη γράψουμε με διαφορετικό τρόπο:

```
(((((cout << " x = ") << x) << "   r = ") << r) << endl);
```

Βλέπουμε ότι γίνεται δεκτή από τον μεταγλωττιστή και βγάζει το ίδιο αποτέλεσμα με την αρχική.

Για να δούμε όμως τι γίνεται εδώ: Ξέρουμε ότι η "`cout << " x = "`" σημαίνει «*στείλε στο ρεύμα cout την τιμή της παράστασης* `" x = "`.» Ωραία! Τότε όμως τι σημαίνει η:

`(cout << " x = ") << x`

Στείλε στο ρεύμα `(cout << " x = ")` *την τιμή της x;*

Ναι, ακριβώς αυτό σημαίνει. *Η "*`cout << " x = "`*" είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.

Όπως καταλαβαίνεις αυτό είναι απαραίτητο για να μπορούμε να γράφουμε σε μια εντολή εξόδου πολλές εξερχόμενες τιμές (παραστάσεις).

Και κάτι ακόμη: Το «*Στείλε στο ρεύμα* `(cout << " x = ")` *την τιμή της x*» πρέπει να διατυπωθεί ακριβέστερα: *Στείλε στο ρεύμα "*`cout << " x = "`*" όπως έχει γίνει μετά την έξοδο της* `" x = "` *, την τιμή της x.*

Παίρνοντας υπόψη μας τα παραπάνω ας σκεφτούμε την εξής περίπτωση: Έχουμε δηλώσει στη **main**

```
ofstream tout( "afile.txt" );
```

και στη συνέχεια καλούμε μια συνάρτηση με παράμετρο *tout*:

```
afunction( tout, . . . );
```

Αν η *tout* περάσει ως παράμετρος τιμής, στο εσωτερικό της συνάρτησης θα δουλεύουμε με ένα αντίγραφο του ρεύματος ενώ η κατάσταση του *tout* της **main** δεν μεταβάλλεται. Όταν ο έλεγχος της εκτέλεσης επιστρέψει στη **main** το *tout* δεν θα μάθει αυτά που έγιναν στην *afunction*!

Επομένως; Η λύση είναι μία:

- ***Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* in out *(αναφοράς).***

  Στην περίπτωσή μας θα πρέπει να έχουμε:

```
void afunction( ofstream& tout, . . . )  { . . . }
```

  Και κάτι ακόμη:

- Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο *cout* βάζε παράμετρο τύπου `ostream` και όχι `ofstream`.
- Παρομοίως, αν θέλεις να περνάς σε μια συνάρτηση είτε το *cin* είτε κάποιο ρεύμα κλάσης `ifstream` βάζε παράμετρο κλάσης *istream*.

Γιατί; Ξαναδιάβασε την §8.1 και πάρε υπόψη σου ότι –όπως θα μάθουμε αργότερα– όπου μπορεί να εμφανιστεί αντικείμενο μιας κλάσης μπορεί να εμφανιστεί και αντικείμενο των κληρονόμων της.

Με βάση αυτά που είπαμε, μια συνάρτηση για είσοδο ή έξοδο στοιχείων έχει πάντοτε μια παράμετρο για μεταβαλλόμενη τιμή: το ρεύμα. Θα πρέπει λοιπόν να είναι συνάρτηση `void`.

13.9.3 Παραδείγματα

Ας δούμε μερικά παραδείγματα εφαρμογής των παραπάνω κανόνων ξεκινώντας από αυτά της προηγούμενης παραγράφου.

Παράδειγμα 1

Θέλουμε μια συνάρτηση *minmax* που θα τροφοδοτείται με δύο ακέραιους *x*, *y* και θα μας επιστρέφει τον ελάχιστο και τον μέγιστο από τους δύο.

Η συνάρτησή μας:

- θα παίρνει (**in**) δύο τιμές, τις *x*, *y* και

- θα επιστρέφει (**out**) δύο τιμές, τις *mnxy*, *mxxy*.

Αφού η συνάρτησή μας έχει δύο εξερχόμενες τιμές, σύμφωνα με τον Κανόνα 3 θα πρέπει να είναι συνάρτηση χωρίς τύπο.

Τι παραμέτρους θα έχει;

- δύο παραμέτρους τιμής, τις *x*, *y* και
- δύο παραμέτρους αναφοράς, τις *mnxy*, *mxxy*.

Να λοιπόν η συνάρτηση της:

```
void minmax( int x, int y, int& mnxy, int& mxxy )
{
   if ( x < y )
   {  mnxy = x;  mxxy = y;  }
   else // x >= y
   {  mnxy = y;  mxxy = x;  }
} // minmax
```

Ας πούμε τώρα ότι δεν μας ζητούν να γράψουμε τη συγκεκριμένη συνάρτηση αλλά να επιλέξουμε τι είδους συνάρτηση ή συναρτήσεις θα γραφούν.

Πρέπει να επιλέξουμε μεταξύ της *minmax()* και των:

```
int min( int x, int y )
{
   int fv( x );
   if ( x > y )  fv = y;
   return fv;
} // min
int max( int x, int y )
{
   int fv( x );
   if ( y > x )  fv = y;
   return fv;
} // max
```

Η *minmax()* προφανώς παραβιάζει τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά αν το πρόγραμμά μας χρειάζεται να υπολογιζει κατ' επανάληψη και το ελάχιστο και το μέγιστο τότε κάθε φορά έχουμε:

- Η *minmax()* θα καλείται ως εξής:

  ```
  minmax( a, b, mnab, mxab );
  ```

  και θα έχουμε:
  - μια κλήση συνάρτησης,
  - μια σύγκριση,
  - δύο εκχωρήσεις.
- Οι *min()* και *max()* θα καλούνται ως εξής:

  ```
  mnab = min( a, b );  mxab = max( a, b );
  ```

  και θα έχουμε:
  - δύο κλήσεις συναρτήσεων,
  - δύο συγκρίσεις,
  - δύο δηλώσεις μεταβλητών με αρχική τιμή (ορισμούς),
  - μια εκχώρηση,
  - δύο επιστροφές τιμών.

Προφανώς, για την περίπτωση αυτή, η *minmax()* είναι προτιμότερη.

Παράδειγμα 2

Θέλουμε μια συνάρτηση (με όνομα *swap*) που θα παίρνει δύο μεταβλητές τύπου `int` και θα αντιμεταθέτει τις τιμές τους.

Στην περίπτωση αυτή θέλουμε μια συνάρτηση που θα *μεταβάλλει* τις τιμές δύο μεταβλητών (**in out**). Κατά τον 3ο Κανόνα θα γράψουμε μια συνάρτηση χωρίς τύπο.

```
void swap( int& x, int& y )
{
   int s( x );
   x = y;  y = s;
} // swap
```

Παράδειγμα 3

Θέλουμε μια συνάρτηση, με όνομα *cntInt()*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου **double** και το πλήθος *n* των στοιχείων του και θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή.

Η *cntInt()*:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και
- με «*το πλήθος n των στοιχείων του*»,
- «*θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή*».

Δηλαδή, τροφοδοτείται με έναν πίνακα –βέλος και πλήθος στοιχείων– (**in**) και επιστρέφει μία ακέραιη τιμή.

Αφού η συνάρτησή μας «*δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή*» θα γράψουμε *συνάρτηση με τύπο*. Γράφουμε λοιπόν:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
```

Η εξερχόμενη τιμή θα επιστρέφει μέσω του ονόματος της συνάρτησης στην κλήση της.

Ολόκληρη η συνάρτηση[6]:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
{
   int m( 0 );

   for ( int k(0); k < n; ++k )
      if (a[k] == static_cast<long int>(a[k]))  ++m;
   return m;
} // cntInt
```

Παράδειγμα 4

Θέλουμε μια συνάρτηση, με όνομα *to1Dgt*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου **double**, και το πλήθος *n* των στοιχείων του και θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους.

Η συνάρτηση:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και «*θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους*»· δηλαδή οι τιμές των στοιχείων του πίνακα θα *μεταβάλλονται* από τη συνάρτηση.

  Ακόμη, η συνάρτηση

- «[*θα τροφοδοτείται*] *με το πλήθος n των στοιχείων του* [*πίνακα*]». Η τιμή της *n* είναι εισερχόμενη στη συνάρτηση.

Αφού η συνάρτησή μας «*έχει μεταβαλλόμενες τιμές*» (πίνακας a) θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο (**void**). Γράφουμε λοιπόν:

```
void to1Dgt( double a[],     // μεταβαλλόμενη παράμετρος
```

[6] Προφανώς, ο έλεγχος `a[k] == static_cast<long int>(a[k])` για ακέραιη τιμή δεν δουλεύει όταν η a[*k*] έχει πολύ μεγάλη τιμή.

```
                int n )          // εισερχόμενη παράμετρος
```

Τα στοιχεία του a έχουν τις μεταβαλλόμενες τιμές.

Ολόκληρη η συνάρτηση:

```
void to1Dgt( double a[], int n )
{
   for (int k(0); k < n; ++k )
      a[k] = (static_cast<long int>(a[k]*10))/10.0;
} // to1Dgt
```

&

Παράδειγμα 5

Στο Παράδ. 2 της §7.7 γράψαμε δύο συναρτήσεις την *gcd()* –που υπολογίζει τον ΜΚΔ δύο ακεραίων– και την *lcm()* που υπολογίζει το ΕΚΠ. Η *lcm()* καλεί την *gcd()*.

Αν έχουμε να επιλέξουμε ποιες και τι είδους συναρτήσεις θα γράψουμε:

- Σε πλήρη συμφωνία με όλους τους κανόνες μας θα γράψουμε την

```
unsigned int gcd( int x, int y )
```

- Η *lcm()* που γράψαμε μπορεί να γραφεί ως εξής:

```
void gcdLcm( int x, int y, int& gcdP, int& lcmP )
{                                 // ΕΚΠ(x,y)*ΜΚΔ(x,y) = x*y
   if ( (x == 0 && y == 0) || x < 0 || y < 0 )
   {
      cout << " η gcdLcm κλήθηκε με "
           << x << ", " << y << endl;
      exit( EXIT_FAILURE );
   }
// (x != 0 || y != 0) && x >= 0 && y >= 0
   gcdP = gcd(x, y);
   lcmP = x * y / gcdP;
} // gcdLcm
```

Εδώ, προφανώς, παραβιάζουμε τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά δεν κάνουμε τους ίδιους υπολογισμούς δύο φορές.

Πρόσεξε ότι ο ΜΚΔ υπολογίζεται σε ένα μέρος μόνον: στη *gcd()*. Για βελτιώσεις ή προβλήματα που μπορεί να παρουσιασθούν παρεμβαίνουμε μόνον εκεί.

- Και την *lcm()*, θα την πετάξουμε; Ε, μη την πετάξεις. Αν έχεις κάποια εφαρμογή που θέλει μόνον το ΕΚΠ χρησιμοποίησέ την.

Μια παρόμοια περίπτωση είναι και αυτή του Παραδ. 2 της §9.4: Οι συναρτήσεις

```
double vectorAvrg( const double x[], int n, int from, int upto)
double stdDev( const double x[], int n, int from, int upto )
```

υπολογίζουν τη μέση τιμή και την τυπική απόκλιση των `x[`*from*`]..x[`*upto*`]` αντιστοίχως. Η *stdDev()* καλεί τη *vectorAvrg()*.

Εκεί όμως έχουμε και κάτι άλλο: μπορούμε να γράψουμε μια:

```
void simpleStat ( const double x[], int n, int from, int upto,
                  double& avrgP, double& stdDevP )
```

που υπολογίζει και τα δύο μεγέθη με πιο αποδοτικό τρόπο.

Το τι γράφουμε, τι όχι και πότε τα χρησιμοποιούμε τα αφήνουμε ως άσκηση για σένα.

&

Παράδειγμα 6

Στο Παράδ. 4 της §12.4 (πολλαπλασιασμός πινάκων) γράφουμε δυο φορές τις ίδιες εντολές για να διαβάσουμε τους δυο πίνακες και τρεις φορές τις ίδιες εντολές για να γράψουμε τους τρεις πίνακες. Δεν θα μπορούσαμε να ευκολύνουμε τη ζωή μας με δύο συναρτήσεις; Φυσικά!

Κατ' αρχάς, αφού οι δύο συναρτήσεις κάνουν είσοδο/έξοδο στοιχείων θα πρέπει να είναι `void`:

```
void input2DAr( istream& tin, . . .
```

```
void output2DAr( ostream& tout, . . .
```

Και οι δύο συναρτήσεις έχουν ως παράμετρο τον πίνακα: η πρώτη ως παράμετρο **out** και η δεύτερη ως παράμετρο **in**. Για τη δεύτερη τα πράγματα είναι απλά:

```
void output2DAr( ostream& tout,
                 const int* a, int nRow, int nCol )
```

Για την πρώτη πώς θα είναι οι παράμετροι; Έτσι:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
```

Οι διαστάσεις των πινάκων δεν διαβάζονται από το αρχείο· είναι γνωστές στο πρόγραμμα που καλεί τη συνάρτηση γι' αυτό και δεν βάζουμε "`int& nRow, int& nCol`". Αργότερα θα δούμε και περιπτώσεις όπου τα πάντα είναι **out**.

Ολόκληρη η *input2DAr*:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
         tin >> a[r*nCol+c];
//          tin >> a[r][c];
   }
} // input2DAr
```

Για τούς πίνακες:

```
int  a[ l ][ m ],  b[ m ][ n ];
```

την καλούμε ως εξής:

```
ifstream atx( "arr.txt" );
input2DAr( atx, &a[0][0], l, m );
input2DAr( atx, &b[0][0], m, n );
atx.close();
```

Αφήνουμε ως άσκηση την *output2DAr*.

Παρατήρηση: ▶

Ωραίες οι συνταγές, αλλά εδώ βλέπουμε συναρτήσεις της πάγιας βιβλιοθήκης της γλώσσας που δεν συμμορφώνονται με αυτές.

- Στην προηγούμενη παράγραφο οι *strtod()* και *strtol()* επιστρέφουν τιμή (`double` ή `long int` αντιστοίχως) αλλά βγάζουν και ένα βέλος μέσω της παραμέτρου *endptr*.
- Στο προηγούμενο κεφάλαιο προσπαθώντας να μιμηθούμε τη λειτουργία των *strlen()* και *strcat()* είδαμε ότι η δεύτερη:
  - αλλάζει την πρώτη παράμετρο,
  - επιστρέφει και τιμή-βέλος.

Τι συμβαίνει με αυτές τις παρανομίες;

Οι απαντήσεις στα ερωτήματα αυτά είναι οι εξής:

- Οι συναρτήσεις αυτές έχουν σχεδιασθεί πολύ παλιά (έρχονται από τη C). Ο τρόπος που σκέφτονταν τότε οι προγραμματιστές ήταν αρκετά διαφορετικός και το βασικό κριτήριο για να πάρουν τις αποφάσεις τους είχε να κάνει με τους περιορισμούς που έβαζαν οι ΗΥ εκείνου του καιρού. Τώρα τα πράγματα είναι αρκετά διαφορετικά.
- Και εσυ, στο μέλλον, πιθανότατα, θα παραβείς αυτούς τους κανόνες. Αλλά να το κάνεις όταν θα έχεις γίνει προγραμματιστής(-στρια) σαν τον D.M. Ritchie ή τον B.W. Kernighan, γιατί τότε θα ξέρεις πότε και πώς να τους παραβείς. Μέχρι τότε, το καλύτερο που έχεις να κάνεις είναι να τους τηρείς.◀

13.10 Υποδείγματα Συναρτήσεων

Έστω ότι θέλουμε να γράψουμε ένα πρόγραμμα που i) θα διαβάζει από το πληκτρολόγιο τις τιμές των στοιχείων ενός μονοδιάστατου πίνακα (με 5 στοιχεία), ii) θα βρίσκει το πλήθος των στοιχείων του με ακέραιη τιμή, iii) θα αποκόπτει τις τιμές τους στο 1ο δεκαδικό ψηφίο και iv) θα τις γράφει στην οθόνη.

Μπορούμε να το γράψουμε χρησιμοποιώντας τις συναρτήσεις που είδαμε στα παραδ. 3 και 4 της προηγούμενης παραγράφου:

```
#include <iostream>
using namespace std;

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main
```

Οι συναρτήσεις έχουν γραφεί έτσι ώστε να μην προκαλούν προβλήματα κατά τη μεταγλώττιση. Δηλαδή, γράψαμε τελευταία τη **main** που έχει τις αναφορές στις δύο συναρτήσεις, ώστε όταν ο μεταγλωττιστής βρει τα ονόματα των συναρτήσεων να τις ξέρει ήδη.

Όταν γράφεις ένα μεγάλο πρόγραμμα κάτι τέτοιο δεν είναι πάντοτε εφικτό ούτε βολικό. Πολύ συχνά το πρόγραμμά μας χρησιμοποιεί συναρτήσεις που δεν γράφονται μαζί με αυτό αλλά υπάρχουν σε διάφορες βιβλιοθήκες και συνδέονται στο πρόγραμμα μετά τη μεταγλώττισή του. Όπως λέγαμε και στην §7.4, η C++ μας επιτρέπει να γράφουμε τις συναρτήσεις σε οποιαδήποτε σειρά (ή και να μην τις γράφουμε καθόλου) αρκεί να βάζουμε πιο πριν **υποδείγματα** (prototypes) των συναρτήσεων.

Το υπόδειγμα μιας συνάρτησης περιέχει τον τύπο του αποτελέσματος, το όνομα της συνάρτησης και –μέσα σε παρενθέσεις– τους τύπους των τυπικών παραμέτρων. Πάντως μπορείς να χρησιμοποιήσεις και την επικεφαλίδα της τερματιζόμενη με ένα ";". Για τις συναρτήσεις του παραπάνω παραδείγματος θα μπορούσαμε να έχουμε:

```
   int cntInt( const double[], int );
   void to1Dgt( double[], int );
```

ή

```
   int cntInt( const double*, int );
   void to1Dgt( double*, int );
```

ή

```
   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );
```

Έτσι, το πρόγραμμά μας θα μπορούσε να γραφεί ως εξής:

```
#include <iostream>
using namespace std;

   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt
```

Πολύ συχνά, κυρίως όταν έχουμε μεγάλα προγράμματα, οι επικεφαλίδες μαζί με διάφορες δηλώσεις μπαίνουν σε ένα ξεχωριστό αρχείο που περιλαμβάνεται (**#include**) στην αρχή του αρχείου του προγράμματος. Π.χ. αν το πρόγραμμά μας βρίσκεται στο αρχείο **truncarr.cpp** θα βάλουμε, κατά τη συνήθεια που επικρατεί, τις επικεφαλίδες σε ένα αρχείο με όνομα: **truncarr.h** και η αρχή του προγράμματός μας θα είναι:

```
#include <iostream>
#include <cmath>

#include "truncarr.h"

using namespace std;

int main()
   . . .
```

13.11 Ένα Δύσκολο Πρόβλημα!

Και τώρα θα γράψουμε πρόγραμμα! Μάθαμε αρκετά ώστε να μπορούμε να αντιμετωπίσουμε ένα πρόβλημα που θα μπορούσε να υπάρχει στον πραγματικό κόσμο.

Ένα συνεργείο έκανε μέτρηση ροής οχημάτων σε κάποιο δρόμο. Σε αρχείο, text – με το όνομα στο δίσκο **autoflow.txt***– καταγράφηκαν οι τιμές ροής σε οχήματα/min κάθε λεπτό. Το πλήθος των τιμών στο αρχείο είναι άγνωστο, αλλά σίγουρα θετικό.*

Οι πρώτες τρεις γραμμές του αρχείου έχουν την «ταυτότητα» της μέτρησης ως εξής:

```
Υπεύθυνος:\t<όνομα>\t<επώνυμο>
Σημείο Μετρήσεων:\t<οδός-αριθμός>\t<περιοχή>\t<δήμος>
Αρχή:\tdd.mm.yyyy\thh:mm
```

Στις υπόλοιπες γραμμές δίνονται οι τιμές από τη μέτρηση, μια σε κάθε γραμμή. Κάθε τιμή είναι γραμμένη στις πέντε πρώτες θέσεις. Πέρα από τη δέκατη θέση μπορεί να υπάρχουν σχόλια που περιγράφουν συμβάντα κατά τη διάρκεια της μέτρησης. Να ένα παράδειγμα:

```
Υπεύθυνος:   Ανδρέας      Νικολόπουλος
Σημείο Μετρήσεων: Τζαβέλλα 48    Νεάπολις     Αθήνα
Αρχή:  15.06.2009   05:00
   26
   30
    8
   28
. . .
   17
   19
    4
   12     / Αρχή λειτουργίας σηματοδότησης
   28
   17
. . .
   17
   31     / Βλάβη σηματοδότη Τζαβέλλα & Γιωργάκου
   23
   24
. . .
```

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει:

1. Το πλήθος τιμών του αρχείου καθώς και τη διάρκεια των μετρήσεων σε h και min.

2. Το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων.

3. Τη μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min.

Όλα αυτά θα γράφονται στην τελική έκθεση, που θα γραφεί σε αρχείο με το όνομα `report.txt`*. Στις πρώτες τρεις γραμμές του αρχείου θα αντιγραφούν οι τρεις πρώτες γραμμές του* `autoflow.txt`*.*

Ένα από τα ζητούμενα της δουλειάς είναι και η μελέτη της μεταβολής της ροής οχημάτων. Ως πρώτο βήμα μας ζητείται να δημιουργήσουμε ένα άλλο αρχείο με τις μεταβολές ροής ανά min.

Σαν πολλά δε ζητάει; Μπα, τα περισσότερα τα ξέρουμε! Ας ξεκινήσουμε με τα αρχεία. Οι προδιαγραφές λένε ότι θα διαβάζουμε ένα αρχείο και θα γράφουμε δύο. Όλα τα αρχεία θα είναι text.

Το πρώτο που θα κάνουμε είναι να διαμορφώσουμε τις προδιαγραφές. Αν $numberPerMin_k$, k: $1..n$ είναι η ακολουθία τιμών που διαβάζουμε από το `autoflow.txt` τότε στο ένα από τα αρχεία θα γράψουμε τις τιμές μιας νέας ακολουθίας, της

$$difference_k = numberPerMin_k - numberPerMin_{k-1},\ k: 2..n$$

«*Το πλήθος τιμών του αρχείου*» είναι το *n*. Αφού οι μετρήσεις λαμβάνονται ανά *min*, η «*διάρκεια των μετρήσεων*» σε *min* είναι *n*. Για να τη μετατρέψουμε «*σε h και min*» παίρνουμε το πηλίκο και το υπόλοιπο της *n* δια 60:

$$hours = n \ / \ 60, \quad minutes = n \ \% \ 60$$

Αφού κάθε $numberPerMin_k$ που διαβάζουμε είναι το πλήθος των αυτοκινήτων που περνούν σε ένα *min*, «*το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων*» είναι προφανώς το άθροισμα των τιμών που διαβάζονται:

$$numberOfCars = \sum_{k=1}^{n} numberPerMin_k .$$

Η «*μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min*» είναι η

$$averageFlow = \sum_{k=1}^{n} numberPerMin_k / n$$

Αυτή μπορεί να υπολογισθεί μόνον αν $n > 0$, αν δηλαδή το αρχείο που μας δίνεται έχει μια τουλάχιστον μέτρηση.

Να λοιπόν οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Απαίτηση: Στο αρχείο `differences.txt` γράφονται τα μέλη της ακολουθίας $difference_k$, k: `2..`n

Στο αρχείο `report.txt` γράφονται:

- οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των *n*, *hours* (`==` n `/ 60`), *minutes* (`==` n `%` 60), *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (`==` *numberOfCars* / n).

Ας κάνουμε τώρα ένα σχέδιο για το πρόγραμμά μας.

Στην αρχή θα πρέπει να ανοιχθούν οπωσδήποτε τα αρχεία `autoflow.txt`, για διάβασμα και το `report.txt` για να γράψουμε τις τρεις πρώτες γραμμές. Στο `differences.txt` θα γράψουμε αν από το `autoflow.txt` διαβάσουμε τουλάχιστον δύο τιμές ροής. Θα πρέπει λοιπόν να περιμένουμε και να το ανοίξουμε μόνον αν βρούμε δεύτερη τιμή; Όχι! Αφού μας ζητείται αρχείο διαφορών θα το δημιουργήσουμε ακόμη και αν το αφήσουμε κενό.

Στη συνέχεια θα αντιγράψουμε στο `report.txt` τις τρεις πρώτες γραμμές.

Μετά θα διαβάζουμε από το `autoflow.txt`, θα υπολογίζουμε και θα γράφουμε στο `differences.txt` μέχρι να τελειώσει το αρχείο. Αλλά εδώ χρειάζεται προσοχή: για την πρώτη τιμή δεν μπορούμε να υπολογίσουμε διαφορά· επομένως χρειάζεται ειδικό χειρισμό.

Όσο διαβάζουμε και υπολογίζουμε και γράφουμε διαφορές θα πρέπει να υπολογίζουμε το *n* (να μετρούμε) και το *numberOfCars* (να αθροίζουμε) ώστε να υπολογίσουμε και τη *averageFlow*.

Τέλος, υπολογίζουμε τη μέση τιμή, γράφουμε όσα χρειάζεται στο `report.txt` και κλείνουμε τα αρχεία.

Να λοιπόν ένα πρώτο σχέδιο του προγράμματός μας:

```
Άνοιξε τα ρεύματα των αρχείων
Επεξεργασία
Κλείσε τα ρεύματα
```

Αλλά, σαν να ξεχάσαμε κάτι εδώ: Θα προχωρήσουμε στις επεξεργασίες αν ανοίξουν τα ρεύματα, αλλιώς τι να επεξεργαστούμε; Το σχέδιο πρέπει να αλλάξει λίγο:

```
Άνοιξε τα ρεύματα των αρχείων
if ( άνοιξαν τα ρεύματα )
{
   Επεξεργασία
   Κλείσε τα ρεύματα
}
```

Στην §0.5 λέγαμε ότι «*το αρχικό πρόβλημα διασπάται σε δύο ή περισσότερα υποπροβλήματα*» από τα οποία το καθένα «*έχει προϋπόθεση την απαίτηση του προηγούμενου*». Ας βάλουμε λοιπόν και εδώ τις προδιαγραφές στα προβλήματα που προκύπτουν από τη διάσπαση:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

```
Άνοιξε τα ρεύματα των αρχείων
```

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής n και το πλήθος των οχημάτων *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$).

Υπολογίστηκαν τα *hours* (== n / 60), *minutes* (== n % 60).

Αν $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση) υπολογίστηκε η *averageFlow* (== *numberOfCars* / n).

Γράφηκαν στο `report.txt` τα n, *hours*, *minutes*, *numberOfCars*, *averageFlow*.

`Κλείσε τα ρεύματα`

Απαίτηση: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n

Στο αρχείο `report.txt` έχουν γραφεί:

- οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των n, *hours* (== n / 60), *minutes* (== n % 60), *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (== *numberOfCars* / n).

Κάπως έτσι γίνεται η ανάλυση των προδιαγραφών μέσα στο σχέδιό μας. Πρόσεξε το "$n > 0$" που μεταφέρεται από βήμα σε βήμα μέχρι το «`Τελικοί υπολογισμοί`» όπου και χρειάζεται για να μπορέσουμε να υπολογίσουμε τη μέση ροή.

13.11.1 «Άνοιξε τα ρεύματα των αρχείων»

Ξεκινούμε από την "`Άνοιξε τα ρεύματα των αρχείων`" και το πρώτο που πρέπει να δούμε είναι το τι μπορεί να συμβεί και να μην ανοίξουμε τα ρεύματα. Θα πεις «Για ποια ρεύματα συζητούμε; Το μόνο πρόβλημα που μπορεί να έχουμε είναι: το ρεύμα από το `autoflow.txt`, να μη βρει το αρχείο!». Ας δούμε μιαν άλλη λεπτομέρεια. Μέχρι τώρα ξέρουμε ότι αν απόπειραθούμε να δημιουργήσουμε ένα αρχείο που ήδη υπάρχει στον δίσκο, αυτομάτως σβήνεται το παλιό και δημιουργείται το καινούριο. Ε, λοιπόν: δεν θέλουμε –το πρόγραμμα που θα γράψουμε– να σβήσει, χωρίς προειδοποίηση, κάποιο αρχείο που ήδη υπάρχει με όνομα `report.txt` ή `differences.txt`.

Θα ανοίξουμε αυτά τα αρχεία με την παρακάτω συνάρτηση:

```
void openWrNoReplace( ofstream& newStream, string flNm, bool& ok )
{
   ifstream test( flNm.c_str() );           // άνοιξε για διάβασμα
   if ( test.fail() )                 // ok, δεν υπάρχει το αρχείο
   {
      newStream.open( flNm.c_str() );        // άνοιξε για γράψιμο
      ok = !newStream.fail();
   }
   else // υπάρχει το αρχείο, κλείσε το και μην το πειράξεις
   {
      test.close();
      ok = false;
   }
// ok = (δεν υπήρχε το αρχείο) && (άνοιξε για γράψιμο)
} // openWrNoReplace
```

Ας πούμε ότι έχουμε δηλώσει:

```
ofstream report;
bool     ok;
```

και στη συνέχεια το ανοίγουμε ως εξής:

```
openWrNoReplace( "report.txt", report, bool& ok );
```

Τι θα γίνει; Η συνάρτηση θα προσπαθήσει να δημιουργήσει ένα τοπικό ρεύμα (*test*) *για διάβασμα* από το αρχείο `report.txt`.

- Αν το *test* δημιουργηθεί τότε το κλείνουμε και δεν το πειράζουμε.
- Αποτυχία της απόπειρας (`test.fail()`) σημαίνει ότι το αρχείο δεν υπάρχει και έτσι προσπαθούμε να το ανοίξουμε για γράψιμο.

Με αυτόν τον τρόπο θα ανοίξουμε τα δύο εξερχόμενα ρεύματα.

Τώρα μπορούμε να προχωρήσουμε στο επόμενο επίπεδο ανάλυσης. Η "`Άνοιξε τα ρεύματα των αρχείων`" αναλύεται ως εξής:

```
Άνοιξε το autoflow για διάβασμα
if (δεν άνοιξε το autoflow)
{
   Μην προχωρείς
}
else    // το autoflow ανοικτό
{
   Άνοιξε το differences για γράψιμο
   if (δεν άνοιξε το differences)
   {
      κλείσε το autoflow
      Μην προχωρείς
   }
   else    // τα autoflow, differences ανοικτά
   {
      Άνοιξε το report για γράψιμο
      if (δεν άνοιξε το report)
      {
         κλείσε τα autoflow, differences
         Μην προχωρείς
      }
      else  //τα autoflow, differences, report ανοικτά
      {
         Προχώρησε στην επεξεργασία
      } // if (δεν άνοιξε το report)...
   } // if (δεν άνοιξε το differences)...
} // if (δεν άνοιξε το autoflow)...
```

Όλα αυτά θα τα βάλουμε σε μια συνάρτηση, ας την πούμε *openFiles*. Να δούμε τι είδους θα είναι αυτή η συνάρτηση και τι παραμέτρους θα έχει.

Η *openFiles* θα ανοίγει τρία ρεύματα από και προς αρχεία ή –με άλλα λόγια– θα δίνει τιμές σε τρία ρεύματα. Θα έχει λοιπόν τρεις *εξερχόμενες* παραμέτρους· θα είναι λοιπόν συνάρτηση *χωρίς τύπο* (`void`). Ναι, αλλά εκείνα τα «`Μην προχωρείς`» και «`Προχώρησε στην επεξεργασία`» πώς θα τα βγάζει προς τα έξω; Χρειαζόμαστε άλλη μια εξερχόμενη παράμετρο ας την πούμε *ok* που θα περνάει προς τα έξω αυτά τα μηνύματα. Αν πάρουμε υπόψη μας ότι τα δύο μηνύματα είναι αμοιβαίως αποκλειόμενα μπορούμε να δηλώσουμε την *ok* τύπου `bool`:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm,
                bool& ok )
```

Παίρνουμε ολόκληρη την *openFiles* μεταφράζοντας σε C++ το σχέδιο που δώσαμε πιο πάνω:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
```

```
                ofstream& report, string reprtFlNm,
                bool& ok )
{
   ok = true;
   autoflow.open( "autoflow.txt" );
   if ( autoflow.fail() )
      ok = false;
   else  // το autoflow ανοικτό
   {
      openWrNoReplace( "differences.txt", differences, ok );
      if ( !ok )
         autoflow.close();
      else    // τα autoflow, differences ανοικτά
      {
         openWrNoReplace( "report.txt", report, ok );
         if ( !ok )
         {
            autoflow.close();
            differences.close();
         } // if (δεν άνοιξε το report)...
      } // if (δεν άνοιξε το differences)...
   } // if (δεν άνοιξε το autoflow)...
} // openFiles
```

Εδώ, ας κάνουμε και μια σύγκριση με τις προδιαγραφές μας: Η τιμή της *ok* είναι η τιμή της «*Το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο.*»

Τώρα, ας αρχίσουμε να γράφουμε τη **main**, μεταφράζοντας σε C++ το σχέδιό μας:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      Επεξεργάσου
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.2 «Επεξεργασία»

Ας έρθουμε τώρα στην «`Επεξεργασία`» που έχει προδιαγραφές:

Προϋπόθεση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο.

Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής *n* και το πλήθος των οχημάτων *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMn_k$).

$n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Η συνθήκη «*το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο*» υπάρχει και στην προϋπόθεση και στην απαίτηση, παραμένει δηλαδή *αναλλοίωτη* από την "`Επεξεργασία`". Δηλαδή, δεν θα βάλουμε πουθενά εντολή "*close*". Αλλά, δεν ξέρεις τι άλλο μπορεί να συμβεί...

Η "$n > 0$" υπάρχει και στην προϋπόθεση και στην απαίτηση· και αυτή αναλλοίωτη; Για να κάνει το πρόγραμμα όλα όσα ζητούνται θα πρέπει να ισχύει αυτή η συνθήκη[7]. Αλλά αυτό δεν το ξέρουμε με βεβαιότητα παρά μόνον αφού προχωρήσουμε στο διάβασμα του αρχείου `autoflow.txt`. Να λοιπόν μια πιθανότητα να υπάρχει κάποιο πρόβλημα:

- Να βρούμε ένα αρχείο με λιγότερες από τέσσερις γραμμές· στην περίπτωση αυτή το *n* είναι 0. Μπορεί οι προδιαγραφές των στοιχείων εισόδου να λένε ότι έχει τουλάχιστον μια τιμή, αλλά ένα καλό πρόγραμμα θα ελέγξει αν πληρούνται οι προδιαγραφές των στοιχείων εισόδου! Θα πρέπει λοιπόν να είμαστε προσεκτικοί όταν έλθει η ώρα να υπολογίσουμε τη μέση τιμή της ροής.

Υπάρχει πιθανότητα να βρούμε άλλο λάθος κατά την επεξεργασία; Φυσικά υπάρχει:

- Αφού το `autoflow.txt` είναι text υπάρχει μεγαλύτερη πιθανότητα να υπάρχουν λάθη μέσα στο αρχείο. Πιθανές προσπάθειες παρεμβάσεων με κάποιον κειμενογράφο μπορεί να έχουν αφήσει διάφορα σφάλματα. Κάποιος έλεγχος εγκυρότητας των στοιχείων είναι αναγκαίος· θέλοντας όμως να κρατήσουμε το παράδειγμά μας απλό δεν θα τον κάνουμε προς το παρόν.

Οι τέσσερις πρώτες γραμμές, ο «τίτλος», θα πρέπει να διαβαστούν –και να αντιγραφούν– ως κείμενο. Από εκεί και πέρα τα ξέρουμε (σχεδόν): Μηδενίζουμε την *n* και τη *numberOfCars* κλπ. Για να είναι τα πράγματα εντάξει θα πρέπει να φτάσουμε να διαβάσουμε μέχρι και μια, τουλάχιστον, τιμή ροής.

```
Αντίγραψε τίτλο
if (εντάξει)
{
   Μηδένισε τους μετρητές
   Διάβασε και επεξεργάσου τις τιμές
   Τελικοί υπολογισμοί
}
```

Όπως κάναμε και πιο πριν, θα πρέπει να διατυπώσουμε προδιαγραφές για τη νέα διάσπαση. Αυτό σου το αφήνουμε ως άσκηση. Στη συνέχεια θα κάνουμε το ίδιο αλλά με αρκετά «ελεύθερο» τρόπο.

Η αντιγραφή του «τίτλου» μπορεί να γίνει όπως ξέρουμε (αντιγραφή αρχείου text), αλλά θα πρέπει να αντιγράψουμε τέσσερις γραμμές μόνον.

Θα γράψουμε μια συνάρτηση *copyTitle* που θα της περνούμε τα ρεύματα *autoflow* και *report* και αυτή θα αντιγράφει από το πρώτο στο δεύτερο τρεις γραμμές. Μέσω μιας παραμέτρου (`bool ok`) θα γνωστοποιεί αν τα κατάφερε.

Τι είδους θα είναι η συνάρτηση; Αφού η συνάρτηση θα εκτελεί αντιγραφή από το ένα αρχείο στο άλλο (είσοδο και έξοδο στοιχείων) η συνάρτηση θα είναι χωρίς τύπο:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
```

Πρόσεξε ότι τα ρεύματα περνούν (πάντοτε) ως παράμετροι αναφοράς.

---

[7] Και για να υπάρχει μια τουλάχιστον διαφορά θα πρέπει να έχουμε "$n \geq 2$".

Θα μπορούσαμε, όπως είπαμε, να γράψουμε τη συνάρτηση όπως μάθαμε στην §8.11, αντιγράφοντας χαρακτήρα προς χαρακτήρα αλλά καλύτερα να χρησιμοποιήσουμε τη *getline()* που μάθαμε στην §10.4. Αφού δηλώσουμε:

```
   int  lineCount;   // μετρητής γραμμών
```

αντιγράφουμε και μετρούμε ως εξής:

```
   lineCount = 0;
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
```

Στο τέλος, αν αντιγράψαμε τέσσερις γραμμές όλα πήγαν εντάξει:

```
   ok = ( lineCount == 3 );
```

Να ολόκληρη η συνάρτηση:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
{
   int lineCount( 0 );   // μετρητής γραμμών
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
   ok = (lineCount == 3);
} // copyTitle
```

Ας έρθουμε τώρα στη "**Μηδένισε τους μετρητές**". Τι σημαίνει; Δύο εντολές:

```
   n = 0;  numberOfCars = 0;
```

Θα μπορούσαμε να τις βάλουμε έτσι και ούτε γάτα ούτε ζημιά.

Εμείς όμως θα τις βάλουμε σε μια συνάρτηση με όνομα *initialize*. Γιατί; Σε κάθε πρόγραμμα, συνήθως, υπάρχει μια συνάρτηση (ή και περισσότερες) που διαμορφώνει μια αρχική κατάσταση (κάνει κάποια αναλλοίωτη να ισχύει): δίνει αρχικές τιμές σε μεταβλητές, ανοίγει αρχεία κλπ. Θα τη δεις συνήθως με το όνομα *initialize* ή κάτι παρόμοιο[8]. Καλό είναι να αρχίσεις να συνηθίζεις κάτι τέτοιες πάγιες πρακτικές.

Τι τύπο θα δηλώσουμε για τις *n* και *numberOfCars*; Θα πεις «θέλει και ρώτημα; `int` βέβαια!» Για την *n* δεν υπάρχει πρόβλημα. Για την *numberOfCars* όμως, αν έχουμε `INT_MAX == 32767` και μετρήσεις από έναν πολυσύχναστο δρόμο για μεγάλο χρονικό διάστημα, ο `int` δεν είναι αρκετός και καλύτερα να βάλουμε `long`.

```
void initialize( int& n, long& numberOfCars )
{
   n = 0;
   numberOfCars = 0;
} // initialize
```

Και τώρα ερχόμαστε στο «ψητό»: "**Διάβασε και επεξεργάσου τις τιμές**". «Σιγά το ψητό» θα πεις «αθροίσματα και μέσες τιμές υπολογίζουμε συνέχεια.» Σωστό! Αρκεί να προσέξουμε την πρώτη τιμή.

Ας δούμε τώρα πώς θα γίνεται η επεξεργασία. Αφού θέλουμε να υπολογίσουμε μέση τιμή αντιγράφουμε ένα γνωστό «πατρόν» επεξεργασίας από το *Μέση Τιμή 5* (§8.5):

```
   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
```

[8] Στην πραγματικότητα η *openFiles* είναι μια τέτοια συνάρτηση για ολόκληρο το πρόγραμμα. Η *initialize()* είναι η αντίστοιχη για την επεξεργασία.

```
a.open( "exp4.txt" );
a >> x;
while ( !a.eof() )
{
   n = n + 1;               // Αυτά γίνονται για
   sum = sum + x;           // όλους τους αριθμούς
   if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
   {
      selSum = selSum + x;       // Αυτά γίνονται μόνο για
      selN = selN + 1;           // τους επιλεγόμενους αριθμούς
   } // if
   a >> x;
} // while
```

Εδώ:

- το ρεύμα που διαβάζουμε (*autoflow*) είναι ήδη ανοικτό,
- αντί για την *sum* έχουμε την *numberOfCars*,
- αντί για την *x* έχουμε την *numberPerMin*,
- μετά την ανάγνωση μιας τιμής *numberPerMin*, θα πρέπει να πηγαίνουμε στην αρχή της επόμενης γραμμής,
- δεν έχουμε επιλεκτική επεξεργασία.

Άρα παίρνουμε:

```
initialize( n, numberOfCars );
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

έχοντας δηλώσει:

```
string aLine;
char*  p;
```

Πρόσεξε πώς γίνεται η ανάγνωση μιας τιμής από το *autoflow*: αφού σε κάθε γραμμή υπάρχει μια τιμή στην αρχή της, διαβάζουμε ολόκληρη τη γραμμή:

```
getline( autoflow, aLine, '\n' );
```

και μετά, με τη *strtol*, μετατρέπουμε σε **long int** τα ψηφία που υπάρχουν στην αρχή της. Διαστήματα που μπορεί να υπάρχουν πριν από τα ψηφία αγνοούνται από τη *strtol*. Αυτή θα σταματήσει τη μετατροπή όταν βρει τον πρώτο χαρακτήρα που δεν είναι ψηφίο. Την *p* δεν τη χρησιμοποιούμε; Όχι, αλλά θα σημειώσουμε ότι εκεί βρίσκεται ένα από τα μεγαλύτερο πλεονεκτήματα αυτού του τρόπου ανάγνωσης· μπορούμε να χρησιμοποιήσουμε την πληροφορία που μας δίνει για να κάνουμε έλεγχο εγκυρότητας των στοιχείων.

Χρειαζόμαστε όμως ακόμη κάτι: τον υπολογισμό των διαφορών. Ας πούμε ότι κρατούμε την προηγούμενη τιμή της *numberPerMin* στην *previous*. Τότε θα έχουμε:

```
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   numberOfCars = numberOfCars + numberPerMin;
   difference = numberPerMin - previous;
   differences << difference << endl;
   previous = numberPerMin;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

Γιατί βγάλαμε την κλήση προς την *initialize*; Διότι έχουμε και την χωριστή επεξεργασία της πρώτης τιμής που πρέπει να μεσολαβήσει.

Την πρώτη τιμή:

- θα τη μετρήσουμε,
- θα την προσθέσουμε,
- *δεν θα υπολογίσουμε διαφορά*, αφού δεν έχουμε προηγούμενη,
- *ούτε θα γράψουμε κάτι στο αρχείο διαφορών*, αλλά
- θα την βάλουμε στην *previous*, για να χρησιμοποιηθεί από τη δεύτερη τιμή.

Να λοιπόν τι πρέπει να γίνει πριν από τη **while**:

```
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );    // 1η τιμή
   if ( !autoflow.eof() )
   {
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      previous = numberPerMin;
      getline( autoflow, aLine, '\n' );
      numberPerMin = strtol( aLine.c_str(), &p, 10 ); // 2η τιμή
      while ( !autoflow.eof() )
      {
         . . .
      } // while
   } // if (!autoflow.eof())
```

Και τώρα μπορούμε να γράψουμε την *processData* αφού αποφασίσουμε για το είδος και τις παραμέτρους της συνάρτησης. Η συνάρτηση διαβάζει από αρχείο και γράφει σε αρχείο. Θα είναι λοιπόν χωρίς τύπο. Οι παράμετροί του θα είναι τα ρεύματα των δύο αρχείων και οι *n* και *numberOfCars* (εξερχόμενες):

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
```

Ολόκληρη η *processData*:

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
{
   int numberPerMin;          // μια τιμή ροής από το autoflow.txt
   string aLine;
   char*  p;
// επεξεργασία 1ης τιμής
   getline( autoflow, aLine, '\n' );
   if ( !autoflow.eof() )
   {
      numberPerMin = strtol( aLine.c_str(), &p, 10 );   // 1η τιμή
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      int previous( numberPerMin );     // η προηγούμενη τιμή ροής
      getline( autoflow, aLine, '\n' );
      while ( !autoflow.eof() )
      {
         numberPerMin = strtol( aLine.c_str(), &p, 10 );// 2η τιμή
         ++n;
         numberOfCars = numberOfCars + numberPerMin;
         int difference( numberPerMin - previous );
                                 // η διαφορά των δύο προγούμενων
         differences << difference << endl;
         previous = numberPerMin;
         getline( autoflow, aLine, '\n' );
      } // while
   } // if (!autoflow.eof())
} // processData
```

Εδώ πρόσεξε: Οι προδιαγραφές μας λένε ότι τα αρχεία θα είναι τελικώς ανοικτά. Τι γίνεται όμως με το ρεύμα *autoflow*; Σταματάμε να ασχολούμαστε με αυτό όταν βρούμε `autoflow.eof()`. Όπως ξέρουμε όμως, στην περίπτωση αυτή, το ρεύμα δεν κλείνει βέβαια αλλά έχουμε αναστολή της λειτουργίας του.

Μετά την επεξεργασία των τιμών έχουμε τους τελικούς υπολογισμούς: από τις *n* και *numberOfCars* υπολογίζουμε

- τις ώρες και τα λεπτά καθώς και
- τη μέση τιμή ροής αν $n > 0$. Εδώ ακριβώς γίνεται ο έλεγχος προϋπόθεσης.

Όλα αυτά γράφονται στο `report.txt`:

```
void finish( ofstream& report, int n, long numberOfCars )
{
   int    hours( n / 60 ),    // διάρκεια μέτρησης σε h . . .
          minutes( n % 60 );  // . . . και min
// υπολογισμοί και γράψιμο στο report.txt
   report << endl;
   report << " Η μέτρηση κράτησε ";
   report.width(2);  report << hours << " h και ";
   report.width(2);  report << minutes << " min." << endl;
   report << " Διάβασα " << n << " τιμές." << endl;
   report << " Μέτρησα " << numberOfCars << " οχήματα συνολικά"
          << endl;
   if ( n > 0 )
   {
      double averageFlow( static_cast<double>(numberOfCars) / n );
                                        // μέση τιμή ροής οχημάτων
      report << " Μέση Τιμή Ροής: ";
      report.setf( ios::fixed, ios::floatfield );
      report.precision(1);
      report.width(5);
      report << averageFlow << " οχήματα/min" << endl;
   }
} // finish
```

Πρόσεξε ότι υπολογίζουμε και γράφουμε τη μέση τιμή ροής αφού προηγουμένως εξασφαλίσουμε ότι $n > 0$.

Τώρα, μπορούμε να γράψουμε την *processing()*. Θα είναι και αυτή `void` και θα μας δίνει μέσω μιας παραμέτρου (`bool ok`) την πληροφορία για το τι κατάφερε να κάνει:

```
void processing( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   int n;             // πλήθος τιμών ροής στο autoflow.txt,
   long numberOfCars; // το άθροισμα των τιμών ροής.
   bool ok;

   copyTitle( autoflow, report, ok );
   if ( ok )
   {
      initialize( n, numberOfCars );
      processData( autoflow, differences, n, numberOfCars );
      finish( report, n, numberOfCars );
   } // if (ok)...
} // processing
```

Προχωρώντας στο γράψιμο της **main**, έχουμε:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
```

```
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.3 «Κλείσε τα Ρεύματα»

Η *closeFiles* κλείνει τα τρία ρεύματα με τα οποία δουλέψαμε:

```
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   autoflow.close();

   differences.close();
   if ( differences.fail() )
   {
      cout << "Τις χάσαμε τις διαφορές!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = true;

   report.close();
   if ( report.fail() )
   {
      cout << "Γράψε την έκθεση με το χέρι!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = ok && true; // άχρηστη
} // closeFiles
```

Μπορεί να αποτύχει το κλείσιμο των αρχείων; Βεβαιότατα! Αν πάρουμε υπόψη μας ότι με το κλείσιμο αντιγράφεται στο αρχείο το όποιο περιεχόμενο του ενταμιευτή, σκέψου την εξής περίπτωση: ας πούμε ότι γράφεις σε αφαιρούμενο δίσκο –π.χ. σε δισκέτα– και βιάζεσαι να την αφαιρέσεις πριν εκτελεσθεί η *close*.

Παρατήρηση: ▶
Βεβαίως εδώ τελειώνουμε όλες τις γραμμές με "`endl`"· έτσι ο ενταμιευτής είναι σχεδόν μονίμως άδειος.◀

13.11.4 Ολόκληρο το Πρόγραμμα

Ολόκληρο το πρόγραμμα θα είναι κάπως έτσι:

```
#include <iostream>
#include <fstream>

using namespace std;

void openWrNoReplace( string flNm, ofstream& newStream,
                      bool& ok );
```

```
void openFiles( ifstream& autoflow,
                ofstream& differences, ofstream& report,
                bool& ok );
void process( ifstream& autoflow,
              ofstream& differences, ofstream& report,
              bool& ok );
void copyTitle( ifstream& autoflow, ofstream& report,
                bool& ok );
void initialize( int& n, long& numberOfVehicles );
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfVehicles );
void finish( ofstream& report,
             int n, long numberOfVehicles );
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok );

int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      if ( !ok )
         cout << "Δεν βρήκα ούτε μια τιμή ροής" << endl;
      closeFiles( autoflow, differences, report, ok );
      if ( ok ) // γράψιμο και κλείσιμο επιτυχώς
         cout << "Τέλος καλό, όλα καλά..." << endl;
   }
   else
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
} // main
```

13.12 Δυο Λόγια για το Παράδειγμά μας

Έχοντας δει αρκετά στοιχεία προγραμματισμού, θελήσαμε να σου δώσουμε ένα παράδειγμα ανάπτυξης ενός μη-τετριμμένου προγράμματος. Η ανάπτυξη βασίστηκε στην αρχή «διαίρει και βασίλευε» (divide and conquer). Δηλαδή, ενώ το αρχικό πρόβλημα έμοιαζε κάπως δύσκολο, το διασπάσαμε σε μικρότερα και μικρότερα, μέχρι που φτάσαμε σε προβλήματα με σχετικά απλή λύση. Όπως είχαμε πει στην §0.5, αυτή η διαδικασία λέγεται και **βήμα προς βήμα ανάλυση** (step by step refinement). Τώρα την είδαμε στην πράξη.

Ξεκινήσαμε από το αρχικό πρόβλημα και πήγαμε προς τις λεπτομέρειες, **από πάνω προς τα κάτω** (top - down). Σε αυτήν την πορεία, δημιουργούσαμε δικές μας εντολές, π.χ.: *openFiles()*, *initialize()*, *finish()* κλπ, που στη συνέχεια τις υλοποιούσαμε ως συναρτήσεις. Ένας τρόπος απεικόνισης αυτής της διαδικασίας είναι το **ιεραρχικό διάγραμμα** που βλέπεις στο Σχ. 13-3. Σε αυτό βλέπεις τα διαδοχικά επίπεδα ανάλυσης, ενώ οι συναρτήσεις που καλούνται από κάθε συνάρτηση φαίνονται σαν κλαδιά της. Το διάγραμμα αυτό είναι ένας τρόπος καταγραφής των αλλεξαρτήσεων των συναρτήσεων (για τις οποίες μιλούσαμε στην §13.6.2.)

Στο διάγραμμα αυτό δεν παρατίθενται συνήθως συναρτήσεις όπως η *sqrt()*, η *strcpy()* και άλλες τέτοιες από τις βιβλιοθήκες της C++ ή άλλες βιβλιοθήκες συναρτήσεων. Έτσι, πρέπει να βλέπεις και την *openWrNoReplace()*. Αυτή θα τη βάλουμε αργότερα σε μια δική μας βιβλιοθήκη.

Ένα άλλο σημείο που πρέπει να προσέξεις, είναι το πώς αποφασίσαμε το ποιες συναρτήσεις θα γράψουμε. Ούτε για μια στιγμή δεν είχαμε κάποια αμφιβολία. Οι συναρτήσεις «ξεπήδησαν», μπορούμε να πούμε, αυθόρμητα. Αυτό *δεν σημαίνει ότι η ανάλυση είναι μοναδική*.

Ενδιαφέρον έχει και το λογικό δέσιμο των συναρτήσεων. Κάθε μια προετοιμάζει την κατάσταση για την επόμενη (ή τις επόμενες) –όπως π.χ. η *openFiles()* για όλες τις επόμενες, η *initialize()* για την *processData()*– και κάθε μια συνεχίζει τη δουλειά από εκεί που την άφησε η προηγούμενη –όπως π.χ. οι *copyTitle()*, *processData()*.

Αυτή είναι η μέθοδος του **δομημένου προγραμματισμού** (structured programming) και είναι η μόνη μέθοδος που μπορείς να χρησιμοποιήσεις προς το παρόν. Αργότερα θα δεις και άλλες μεθόδους σχεδίασης λογισμικού και (γενικότερα) συστημάτων. Αλλά, παντού θα βλέπεις αυτές εδώ τις αρχές. Θα δεις ακόμη ότι έχει γίνει πολλή δουλειά στο δέσιμο αυτής της μεθόδου με τη μαθηματική λογική.

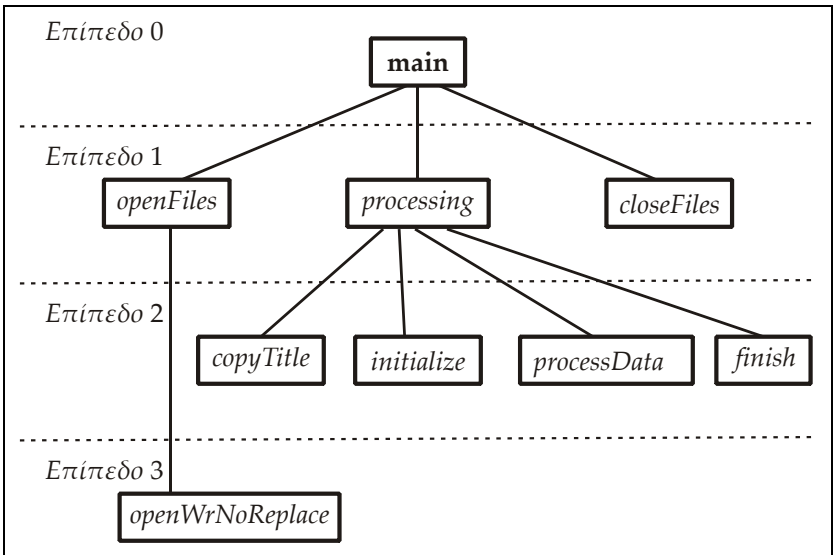
Πρόσεξε ακόμη τη σειρά υλοποίησης των συναρτήσεων. Αρχίσαμε από το τελευταίο επίπεδο, υλοποιώντας συναρτήσεις που δεν καλούν άλλες και προχωρήσαμε προς τα πάνω. Τελευταία υλοποιήθηκε η **main**. Αυτή η διαδικασία λέγεται **υλοποίηση από κάτω προς τα πάνω** (bottom-up implementation) και συνήθως πάει μαζί με τη **σχεδίαση από πάνω προς τα κάτω** (top-down design, bottom-up implementation).

Παρατηρήσεις: ▶

Κατά τ' άλλα, το πρόγραμμα που γράψαμε θέλει κι άλλη δουλειά, κυρίως στον τομέα της ασφάλειας.

1. Ένα πράγμα που δεν μας απασχόλησε καθόλου είναι ο **έλεγχος εγκυρότητας** των στοιχείων (data validation) που διαβάσαμε. Για παράδειγμα, δεν είναι δυνατόν να έχουμε *αρνητικές* τιμές ροής αλλά και μια τιμή 5000 *οχήματα/min* είναι απαράδεκτη. Ας πούμε ότι ο έλεγχος εγκυρότητας των στοιχείων έχει γίνει πιο πριν, από άλλο πρόγραμμα.

2. Ας έλθουμε στο άνοιγμα των αρχείων: Το πρόγραμμά μας είναι εξαιρετικώς ανελαστικό· ένα καλύτερο πρόγραμμα, κάθε φορά που θα εύρισκε πρόβλημα με κάποιο αρχείο, θα έπρεπε να ζητάει νέο όνομα αρχείου. ◀



Σχ. 13-3 Ιεραρχικό διάγραμμα της εφαρμογής. Η *openWrNoReplace* (επομένως και το Επίπεδο 3) θα ήταν καλό να μην υπάρχει στο διάγραμμα.

Ασκήσεις

Α Ομάδα

Για κάθε συνάρτηση που ζητείται στις Ασκ. 13-1..13-11 θα πρέπει να δικαιολογήσεις το είδος της συνάρτησης (με τύπο ή `void`*) και το είδος και τον τύπο κάθε παραμέτρου. Γράψε πρόγραμμα που θα δοκιμάζει τη συνάρτηση.*

13-1 Γράψε τη συνάρτηση *output2Dar()*, όπως την προδιαγράψαμε στο Παράδ. 6 της §13.9.3.

13-2 (συνέχεια της Άσκ. 12-5) Γράψε συνάρτηση που θα τροφοδοτείται με έναν δισδιάστατο πίνακα και δύο φυσικούς *c*1, *c*2 και θα αντιμεταθέτει τις τιμές των στοιχείων των στηλών *c*1, *c*2.

13-3 (συνέχεια της Άσκ. 12-12) Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και έναν φυσικό *k* και θα αντιμεταθέτει τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης του πίνακα.

13-4 Γράψε συνάρτηση με όνομα *h3*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με τρεις πραγματικές τιμές, ας τις πούμε χ_1, χ_2, χ_3, και έναν ακέραιο *n* και θα επιστρέφει τη διαφορά $\chi_1 - \chi_2$ αν *n* == 0, τη $\chi_2 - \chi_3$ αν *n* == 1 και τη $\chi_3 - \chi_1$ αν *n* == 2.

13-5 Μια συνάρτηση ορίζεται στο {1,2,3}×ℝ×ℝ ως εξής:

$$g(n, x, y) = \begin{cases} x + y, & n = 1 \\ x\eta\mu y, & n = 2 \\ x - 1/y, (y \neq 0) & n = 3 \end{cases}$$

Πώς θα υλοποιήσουμε τη *g()* στη C++;

13-6 Θέλουμε μια συνάρτηση, με όνομα *q1*, που θα τροφοδοτείται μέσω των παραμέτρων της, με τρεις πραγματικούς αριθμούς, ας τους πούμε *x*, *y*, *z*, και θα υπολογίζει και θα μας επιστρέφει δύο τιμές, των παραστάσεων p_1 και p_2, όπου:

αν $z > 0$ τότε $p_1 = \dfrac{x^z + y^z}{z}$ και $p_2 = z^{y+x}$,

αν $z < 0$ τότε $p_1 = \dfrac{x^{-z} - y^{-z}}{z}$ και $p_2 = (-z)^{y-x}$.

Δεν επιτρέπεται να κληθεί η *q1()* με τιμή της παραμέτρου *z* ίση με 0 (μηδέν).

13-7 Γράψε συνάρτηση που θα τροφοδοτείται, μέσω των παραμέτρων της, με δύο μονοδιάστατους πίνακες x, y –με το ίδιο πλήθος *n* στοιχείων τύπου `double`–και με έναν πραγματικό *w*. Η συνάρτηση θα κάνει το εξής: θα αλλάζει τις τιμές των στοιχείων των x και y ως εξής: αν αρχικώς x_k == *a* και y_k == *b* τότε τελικά θα πρέπει να έχουμε: x_k == *a* – *wb* και y_k == *a* + *wb*, για όλα τα *k* από 0 μέχρι *n* – 1.

Β Ομάδα

13-8 Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει τον **ανάστροφό** του (τον πίνακα που έχει ως γραμμές τις στήλες του αρχικού). Γράψε συνάρτηση που θα κάνει το ίδιο για τυχαίο δισδιάστατο πίνακα (όχι κατ' ανάγκη τετραγωνικό).

13-9 Κάθε τετραγωνικός πίνακας M (*n*×*n*) μπορεί να γραφεί ως άθροισμα δύο τετραγωνικών πινάκων *n*×*n*· ενός συμμετρικού S και ενός αντισυμμετρικού A. Τα στοιχεία τους:

$$s_{rc} = \frac{m_{rc} + m_{cr}}{2} (= s_{cr}) \text{ και } a_{rc} = \frac{m_{rc} - m_{cr}}{2} (= -a_{cr})$$

Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει το συμμετρικό και το αντισυμμετρικό μέρος του.

13-10 Με βάση το προγράμματα που έγραψες για τις Άσκ. 5-7, 5-8 γράψε μια συνάρτηση, με όνομα *trinomial()*, που:

- Θα τροφοδοτείται με τους (πραγματικούς) συντελεστές *a*, *b*, *c* ενός τριωνύμου $ax^2 + bx + c$.
- Θα υπολογίζει και θα επιστρέφει τέσσερις πραγματικές τιμές *reX1*, *imX1*, *reX2*, *imX2*, τα πραγματικά και φανταστικά μέρη των ριζών της εξίσωσης $ax^2 + bx + c = 0$.
- Θα επιστρέφει και μια ακέραιη τιμή, *n*, που θα είναι:
  - −1, αν η εξίσωση είναι αδύνατη,
  - 0 αν δεν έχει πραγματικές ρίζες,
  - 1, αν είναι πρώτου βαθμού με μία ρίζα (στη *reX1*),
  - 2, αν έχει δύο πραγματικές ρίζες (οι *imX1*, *imX2* θα είναι 0),
  - *INT_MAX*, αν είναι αόριστη.

Γράψε πρόγραμμα που θα διαβάζει τρεις πραγματικούς *a*, *b*, *c* από το πληκτρολόγιο και, καλώντας την *trinomial()*, θα λύνει την $ax^2 + bx + c = 0$.

13-11 Γράψε μια συνάρτηση, με όνομα *addPrev*, που θα τροφοδοτείται με ένα μονοδιάστατο πίνακα a με στοιχεία τύπου `double` και θα αντικαθιστά την τιμή του κάθε στοιχείου (εκτός από το a[0]) με το άθροισμά της παλιάς και της τιμής του προηγούμενου στοιχείου, δηλαδή: $a[k]_{\text{νέα}} = a[k]_{\text{παλιά}} + a[k-1]_{\text{παλιά}}$, k: 1..n−1. (Προσοχή! Θα πας από την αρχή προς το τέλος ή από το τέλος προς την αρχή;).

13-12 Κάνε αυτό που σου αφήσαμε ως άσκηση στην §13.11.2: διατύπωσε προδιαγραφές για τα επί μέρους βήματα της διάσπασης του βήματος "`Επεξεργασία`" του αρχικού προγράμματος. Μετά έλεγξε τις συναρτήσεις που γράψαμε (δηλαδή: κάνε μια άτυπη επαλήθευση). Συμμορφώνονται με τις προδιαγραφές;

13-13 α) Γράψε συνάρτηση, ας την πούμε *toSec*, που θα παίρνει *h* (0..23), *min*, *sec* μιας χρονικής στιγμής και θα τα μετατρέπει σε *sec* από τα τελευταία μεσάνυκτα.

β) Γράψε συνάρτηση, ας την πούμε *toHms*, αντίστροφη της προηγούμενης: θα παίρνει δευτερόλεπτα από τα τελευταία μεσάνυκτα και θα μας δίνει *h*, *min*, *sec*.

γ) Γράψε συνάρτηση *readHms()* που θα διαβάζει από το πληκτρολόγιο, θα ελέγχει και θα περνάει στο πρόγραμμά μας μια χρονική στιγμή που δίνεται στη μορφή *h*, *min*, *sec*.

Γράψε πρόγραμμα που θα χρησιμοποιεί τις παραπάνω συναρτήσεις για να

- διαβάσει από το πληκτρολόγιο δύο τριάδες ακεραίων h_1, m_1, s_1 και h_2, m_2, s_2 που είναι ώρα, πρώτα λεπτά και δευτερόλεπτα δύο χρονικών στιγμών,
- να ελέγξει αν είναι σωστές: $0 \le h_1, h_2 < 24$, $0 \le m_1, m_2 < 60$, $0 \le s_1, s_2 < 60$,
- να υπολογίσει και θα μας δώσει τη χρονική διαφορά μεταξύ των δύο χρονικών στιγμών σε ώρες, λεπτά και δευτερόλεπτα στη μορφή: *hh*:*mm*:*ss*. Π.χ.: Αν δοθούν: ***15 59 0*** και ***16 1 12*** γράφει: `00:02:12`.

Γ Ομάδα

13-14 Στο παιχνίδι "Φιδάκι" (snakes and ladders) ο κάθε παίκτης έχει να διανύσει έναν δρόμο με 100 βήματα αριθμημένα από 1 μέχρι 100. Ο κάθε παίκτης προχωράει όσα βήματα δείξει το ζάρι, που ρίχνει κάθε φορά που έρχεται η σειρά του. Για να τερματίσει ένας παίκτης, θα πρέπει να φέρει ζαριά που να τον φέρει ακριβώς στο 100. Αν φέρει μεγαλύτερη επιστρέφει πίσω όσα βήματα περισσεύουν (π.χ. αν είναι στο 96 και φέρει 6 θα προχωρήσει μέχρι το 100 με τα τέσσερα και θα επιστρέψει στο 98 με τα δύο που περίσσεψαν). Μια παρτίδα του παιχνιδιού τελειώνει όταν τερματίσει ένας παίκτης.

- Σε όρισμένα βήματα υπάρχουν φίδια· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα το φίδι τον «καταπίνει» και τον βγάζει στο βήμα που είναι η ουρά του (μερικά βήματα πιο πίσω).
- Σε άλλα βήματα υπάρχουν σκάλες· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα «ανεβαίνει» τη σκάλα που τον βγάζει στο βήμα που είναι η άλλη άκρη της (μερικά βήματα πιο μπροστά).

Στους παρακάτω πίνακες βλέπεις πού αρχίζουν και πού καταλήγουν οι σκάλες και τα φίδια:

Σκάλες		**Φίδια**	
αρχή	τέλος	αρχή	τέλος
5	12	10	2
17	30	25	15
40	66	37	20
60	78	50	25
75	96	88	64

Γράψε πρόγραμμα που θα προσομοιώνει μια παρτίδα του παιχνιδιού με τέσσερις παίκτες.

13-15 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο έναν φυσικό αριθμό n και θα μας δίνει όλους τους πρώτους φυσικούς που είναι ≤ n. Ο αλγόριθμος που θα χρησιμοποιήσεις είναι το *Κόσκινο του Ερατοσθένη*, που δουλεύει ως εξής:

1. Γράψε όλους τους φυσικούς από 1 μέχρι n
2. Βάλε $p = 2$
3. while ($p^2 \le n$) {

3.1 Διάγραψε όλα τα πολλαπλάσια του p

3.2 Βάλε p = επόμενος αριθμός που δεν έχεις διαγράψει

}

Παράδειγμα: (n = 22)

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
p = 2
2 3   5   7   9    11    13    15    17    19    21
p = 3
2 3   5   7        11    13          17    19
```

κεφάλαιο

14

Συναρτήσεις III

Ο στόχος μας σε αυτό το κεφάλαιο:

Να καλύψουμε μερικά υπόλοιπα σχετικά με συναρτήσεις. Μερικά από αυτά είναι πολύ σημαντικά:

- Οι συναρτήσεις *ανάκλησης* (callback).
- Η *επιφόρτωση* συναρτήσεων και τελεστών.
- Τα *περιγράμματα* (templates) συναρτήσεων.
- Οι *εξαιρέσεις* (exceptions) και η διαχείρισή τους.

Προσδοκώμενα αποτελέσματα:

Η ενσωμάτωση της χρήσης των παραπάνω τεχνικών στην ανάπτυξη λογισμικού που κάνεις θα πάρει καιρό· η πλήρης αξιοποίησή τους έρχεται με τη σχετική ωριμότητα. Είναι όμως ένα σημαντικό βήμα προς την επαγγελματική διαδικασία ανάπτυξης λογισμικού. Ακόμη, σε προετοιμάζουν για το επόμενο σχετικό βήμα που θα δούμε στο τελευταίο κεφάλαιο του *Μέρους Β*.

Έννοιες κλειδιά:

- *συναρτήσεις* `inline`
- *προκαθορισμένες τιμές παραμέτρων*
- *παράμετρος-συνάρτηση*
- *βέλος προς συνάρτηση*
- *συνάρτηση ανάκλησης* (*callback*)
- *επιφόοτωση συναρτήσεων*
- *επιφόοτωση τελεστών*
- *γενικές συναρτήσεις*
- *περιγράμματα συναρτήσεων*
- *παράμετροι της* ***main***
- *στοίβα* (*stack*)
- *εξαιρέσεις*
- *αναδρομή*

Περιεχόμενα:

14.1 Συναρτήσεις "inline"

Η κλήση μιας συνάρτησης χρειάζεται κάποιο χρόνο εκτέλεσης (πέρα από το χρόνο εκτέλεσης των εντολών που έχει στο σώμα της). Μια παλιά (και παλιομοδίτικη) προγραμματιστική συνταγή λέει: αν μια συνάρτηση καλείται πολύ συχνά μη τη γράφεις ως χωριστή συνάρτηση αλλά όπου υπάρχει η κλήση της αντίγραψε όλες τις εντολές της. Κάτι τέτοιο βέβαια «φουσκώνει» πολύ το πρόγραμμα και δεν είναι εύκολα διαχειρίσιμο. Το πρόβλημα λύθηκε, από πολύ παλιά, με τις **μακροσυναρτήσεις** (macros): πρόκειται για συναρτήσεις που γράφονται μεν χωριστά αλλά ο μεταγλωττιστής αντικαθιστά τις κλήσεις τους με τις (μεταγλωττισμένες) εντολές τους.

Αν και σήμερα το πρόβλημα με τις κλήσεις συναρτήσεων δεν είναι και τόσο σοβαρό, η C++[1] μας δίνει τη δυνατότητα να ζητήσουμε από τον μεταγλωττιστή να χειριστεί κάποια συνάρτηση ως μακροσυνάρτηση. Αν γράψεις, π.χ.:

```
inline int max( int x, int y )
{
   int fvx;

   if ( x > y )  fvx = x;
          else  fvx = y;
   return fvx;
} // max int
```

ζητάς από τον μεταγλωττιστή να χειριστεί τη συνάρτηση ως μακροσυνάρτηση. Δεν είναι καθόλου σίγουρο ότι ο μεταγλωττιστής θα σε υπακούσει: υπακούει όταν η συνάρτηση είναι αρκετά απλή. Στην περίπτωσή μας η **if** μπορεί να τον αποτρέψει. Αν όμως γράψεις:

```
inline int max( int x, int y )  {  return (x > y) ? x : y;  }
```

είναι σίγουρο ότι θα σε υπακούσει.

14.2 Προκαθορισμένες Τιμές Παραμέτρων

Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση, ας την πούμε *inc*, με τις εξής προδιαγραφές:

```
void inc( int& x, int s )
```

[1] Η προδιαγραφή "`inline`" έχει περιληφθεί και στην τυποποίηση της C του 1999 (ISO/ IEC 1999).

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }
{ x == x_0 } `inc( x );` { x == $x_0 + 1$ }
Δηλαδή, η *inc* θα μπορεί να καλείται άλλοτε με ένα όρισμα και άλλοτε με δύο![2]

Ας πάρουμε μόνον την:

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }

Αυτήν μπορούμε να τη γράψουμε; Εύκολα:

```
void inc( int& x, int s )
{
   x += s;
} // inc
```

Με μια μικρή μετατροπή μπορούμε να έχουμε αυτό που μας ενδιαφέρει:

```
void inc( int& x, int s = 1 )
{
   x += s;
} // inc
```

Τι λέει η νέα επικεφαλίδα; Αν δεν δοθεί δεύτερο όρισμα –αντίστοιχο της *s*– η *s* να θεωρηθεί ότι είναι 1.

Έτσι, αν στο πρόγραμμά μας δώσουμε:

```
   int p;

   p = 5;   inc( p, 2 );   cout << p << endl;
   p = 5;   p += 2;   cout << p << endl;
   p = 5;   inc( p );   cout << p << endl;
   p = 5;   ++p;   cout << p << endl;
```

θα πάρουμε:

```
7
7
6
6
```

Αν λοιπόν γράψουμε καταλλήλως μια συνάρτηση μπορούμε να την καλούμε παραλείποντας μερικά ή όλα τα ορίσματα.

Είναι φανερό ότι,

- ***Στην κλήση μιας συνάρτησης, δεν μπορείς να παραλείψεις ορίσματα που αντιστοιχούν σε παραμέτρους αναφοράς.***

Ακόμη:

- ***Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.***

Φυσικά, η συνάρτηση θα πρέπει να είναι γραμμένη καταλλήλως. Ας δούμε τι σημαίνει αυτό με ένα παράδειγμα. *Δεν επιτρέπεται να γράψεις*:

```
int f( double x, int y = 0, float z )
```

ενώ *μπορείς να γράψεις*:

```
int f( double x, int y = 0, float z = 1.5 )
```

Η κλήση της *f* μπορεί να είναι:

```
f( 1.56 )    f( a+b, a, b/2 )   f( p+0.5, n+1 )
```

- Στην πρώτη περίπτωση το 1.56 θα γίνει τιμή της *x*. Οι *y* και *z* θα έχουν τις ερήμην καθορισμένες τιμές τους 0 και 1.5 αντιστοίχως.
- Στη δεύτερη περίπτωση η τιμή της *a+b* θα γίνει τιμή της *x*, η τιμή της *a* θα γίνει τιμή της *y* και η τιμή της *b*/2 θα γίνει τιμή της *z*.

[2] Δηλαδή θα δουλεύει άλλοτε ως "`x += a`" και άλλοτε ως "`++x`" ή "`x += 1`".

- Στην τρίτη περίπτωση οι τιμές των p+0.5 και n+1 θα γίνουν τιμές των x και y αντίστοιχα. Η z θα ξεκινήσει με την ερήμην καθορισμένη τιμή της 1.5.

14.3 Παράμετρος – Συνάρτηση

Θέλουμε να γράψουμε μια συνάρτηση που θα υπολογίζει, προσεγγιστικά, μια λύση της εξίσωσης $f(x) = 0$ στο διάστημα $[a_0,b_0]$. Μια πολύ απλή και σίγουρη μέθοδος είναι αυτή της **διχοτόμησης**[3] (bisection).

Η μέθοδος αυτή προϋποθέτει ότι η f είναι συνεχής στο διάστημα $[a_0,b_0]$ και ότι η $f(a_0)f(b_0) \le 0$. Αν m_0 είναι το μέσο του διαστήματος $[a_0,b_0]$, ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
αλλιώς
   ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

Τα παραπάνω γράφονται εύκολα σε C++ ως εξής:

```
a = a0;  b = b0;
m = (a + b) / 2;
if ( f(a)*f(m) <= 0.0 )  b = m;
                    else  a = m;
```

Το «*ψάχνουμε για τη λύση στο διάστημα* $[a_0,m_0]$» το μεταφράσαμε «`b = m`», που βέβαια είναι σωστό. Παρακάτω θα δεις ότι μπορεί να μεταφραστεί και αλλιώς (Άσκ. 14-3).

Στο νέο διάστημα ψάχνουμε με τον ίδιο τρόπο. Πρόσεξε ότι οι τιμές των a, b μεταβάλλονται έτσι που το εύρος του διαστήματος να ελαττώνεται. Πότε σταματούμε; Όταν η τιμή της m προσεγγίσει ικανοποιητικά την τιμή της λύσης. Αλλά πόσο είναι το μέγιστο σφάλμα που μπορεί να έχουμε εδώ; Η απόσταση κάθε σημείου του διαστήματος $[a,b]$ από το μέσο του m είναι $|b - a|/2$ το πολύ. Αν ρ είναι η ακριβής τιμή της λύσης, τότε θα έχουμε και $|m - \rho| \le |b - a|/2$. Αν λοιπόν θέλουμε να υπολογίσουμε τη λύση με ακρίβεια ε, θα μικρύνουμε το διάστημα τόσο[4] ώστε $|b - a|/2 < \varepsilon$.

Ο αλγόριθμός μας θα είναι:

```
a = a0;  b = b0;
while ( fabs(b - a)/2 >= epsilon )
{
   m = (a + b) / 2;
   if (f(a)*f(m) <= 0.0)  b = m;
                     else  a = m;
} // while
root = m;
```

Όλα αυτά θα γίνονται με την προϋπόθεση ότι $f(a_0)f(b_0) \le 0$. Αν δεν ισχύει αυτή η συνθήκη για το αρχικό διάστημα, καλύτερα να μην αποπειραθούμε να προχωρήσουμε, αλλά να στείλουμε στο πρόγραμμα που καλεί τη διαδικασία ένα σήμα λάθους. Αυτό συνήθως γίνεται με μια παράμετρο, ας την πούμε *errCode*, μέσω της οποίας επιστρέφεται τιμή 0 αν όλα πήγαν καλά και 1 αν το αρχικό διάστημα ήταν λάθος.

Τι είδους συνάρτηση θα γράψουμε: με τύπο ή χωρίς τύπο; Η συνάρτησή μας θα τροφοδοτείται με την f και τα a_0, b_0 και ε και θα επιστρέφει την προσέγγιση της ρίζας και την *errCode*. Αφού θα επιστρέφει δύο τιμές θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο:

```
void bisection( ...
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

---

[3] Περισσότερα για τη μέθοδο αυτή σε οποιοδήποτε βιβλίο Αριθμητικής Ανάλυσης αλλά και στο βιβλίο Ανάλυσης της Γ′ Λυκείου. (Κατσαργύρης et al. 1992).
Δες και στο `http://en.wikipedia.org/wiki/Bisection_method`.

[4] Η αλήθεια είναι ότι συνήθως έχουμε πετύχει την ακρίβεια που θέλουμε πιο πριν.

Οι *a*0, *b*0, *epsilon* είναι παράμετροι τιμής, μια και θα έχουν πληροφορίες που δίνονται *προς* τη διαδικασία. Η *root* θα μεταφέρει τη λύση της εξίσωσης *από* τη διαδικασία προς το πρόγραμμα που την κάλεσε. Το ίδιο και η *errCode*. Και οι τελείες στην αρχή; Μα εκεί θα πρέπει να μπει μια παράμετρος που θα μας δίνει την *f* ! Τι παράμετρος θα είναι αυτή; Η C++ λέει: εκεί θα πρέπει να βάλουμε μια παράμετρο *βέλος* προς μια συνάρτηση που επιστρέφει τιμή τύπου **double**. Δηλαδή κάτι σαν:

```
double* f(double)
```

Το "**(double)**" μετά το *f* δείχνει ότι η *f* είναι συνάρτηση με μια παράμετρο, τύπου **double**. Τώρα θα γυρίσεις στον πίνακα προτεραιοτήτων (Παράρ. Ε) και θα δεις ότι πρώτα αναγνωρίζεται το "**f(double)**" και μετά το "*". Αλλά αφού το *f* είναι βέλος, το "**f(double)**" δεν έχει νόημα. Το "***f**" είναι συνάρτηση και νόημα έχει το "**(*f)(double)**". Άρα, το σωστό είναι:

```
double (*f)(double)
```

και η επικεφαλίδα είναι:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
```

Αλλά και *μέσα στο σώμα της συνάρτησης* –κατ' αρχήν– *δεν έχει νόημα να γράφουμε*: "**f(a)**" *αλλά*: "**(*f)(a)**". Να λοιπόν η συνάρτηση:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
{
   double m;

   if ( (*f)(a)*(*f)(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b-a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( (*f)(a)*(*f)(m) <= 0.0 )  b = m;
                                   else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Η C++ σου κάνει ένα δώρο· *μπορείς να γράφεις απλώς*:

```
   if ( f(a)*f)(b) > 0 )
```

και

```
         if ( f(a)*f(m) <= 0.0 )  b = m;
```

αντιστοίχως.

Τώρα να δούμε πώς θα καλέσουμε τη *bisection()*. Ας πούμε ότι θέλουμε λύση της εξίσωσης $x - ln(x) - 2 = 0$ στο διάστημα [0.1, 1]. Κατ' αρχάς, γράφουμε μια συνάρτηση που υπολογίζει την τιμή της $x - ln(x) - 2$:

```
double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και –π.χ. μέσα στη **main**– βάζουμε την εντολή:

```
   bisection( &q, 0.1, 1.0, 1e-5, riza, errCode );
```

Τι σημαίνει **&q**; Ας πούμε ότι είναι είναι ένα βέλος προς τη θέση της μνήμης όπου είναι γραμμένες οι εντολές της *q()*.

Υπάρχει όμως και εδώ το δώρο της C++· σου επιτρέπει να παραλείψεις το **&** και να γράψεις απλούστερα:

```
   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
```

Αν θέλεις να βρείς τη λύση της εξίσωσης συνx = 0 στο διάστημα [0,π] δώσε την εντολή:

```
   bisection( cos, 0.0, 4*atan(1), 1e-5, riza, errCode );
```

Φυσικά, στην περίπτωση αυτή, δεν θα πρέπει να ξεχάσεις να βάλεις "**#include cmath**" όπου υπάρχει το υπόδειγμα των *cos()* και *atan()*.

Να ένα παράδειγμα χρήσης της *bisection()*:

```
#include <iostream>
#include <cmath>

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   const double pi( 4*atan(1.0) );
   double riza;
   int   errCode;

   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
   bisection( cos, 0.0, pi, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
double q( double x )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Στη συνέχεια βλέπεις το αποτέλεσμα του παραπάνω προγράμματος.

```
 Ρίζα = 0.1585983
 Ρίζα = 1.570784
```

Παρατηρήσεις:▶ σχετικώς με τη μέθοδο της διχοτόμησης

1. Πρόσεξε ότι μετά από n επαναλήψεις το σφάλμα είναι το πολύ $|b - a|/2^n$. Επομένως, θα μπορούσαμε ισοδυνάμως αντί για το ε να δίνουμε στη συνάρτηση κάποιο *nMax*.
2. Αν θελήσεις να χρησιμοποιήσεις τη *bisection()* πρόσεξε το εξής: το κριτήριο τερματισμού $|b - a|/2 < \varepsilon$ συχνά δεν είναι αρκετό. Ένα άλλο κριτήριο που μπορεί να χρησιμοποιείται (μαζί με αυτό που έχουμε ή –ισοδυνάμως– μαζί με μέγιστο αριθμό επαναλήψεων) είναι το $|f(x)| < \varepsilon'$.◀

14.4 * Συναρτήσεις και Βέλη – Συναρτήσεις Ανάκλησης

Το είδαμε λοιπόν και αυτό: βέλος προς συνάρτηση! Ας κάνουμε όμως μια αφαίρεση: να δούμε τι πρόβλημα είχαμε να λύσουμε και πώς το λύσαμε.

Πολύ συχνά έχουμε να γράψουμε πρόγραμμα που δεν έχει ορισμένες τις προδιαγραφές του μέχρι την τελευταία λεπτομέρεια. Στην περίπτωσή μας έπρεπε να γράψουμε συνάρτηση για την προσεγγιστική επίλυση της εξίσωσης $f(x) = 0$ χωρίς να ξέρουμε ποια ακριβώς είναι η f. Όταν γράφουμε τη *bisection* το μόνο που ξέρουμε για την f είναι ότι:

f : `double` → `double`

Τι κάνουμε λοιπόν; Δίνουμε λύση στο πρόβλημά μας –δηλαδή γράφουμε συνάρτηση– *με παράμετρο* την *f*. Και το εργαλείο που χρησιμοποιούμε για την υλοποίηση είναι το *βέλος προς τη συνάρτηση f*. Λέμε ότι η παράμετρος *f* περιμένει μια **συνάρτηση ανάκλησης** (callback function)[5]. Στο παράδειγμά μας, τέτοιες είναι η *q* και η *cos*.

Ας δούμε άλλο ένα παράδειγμα από την API των Windows[6]. Οι προγραμματιστές που την έγραψαν είχαν το εξής πρόβλημα: Όταν κάποιος προγραμματιστής (που χρησιμοποιεί την API για να γράψει μια εφαρμογή Windows) θέλει να βγάζει το πρόγραμμά του ένα **πλαίσιο διαλόγου** (dialog box) θα πρέπει να το σχεδιάσει και να γράψει μια συνάρτηση που θα καθορίζει όλη την αλληλεπίδρασή του με τον χρήστη. Αλλά ο προγραμματιστής της εφαρμογής δεν θα πρέπει να ασχολείται με το πώς θα εμφανιστεί το πλαίσιο· αυτό γίνεται με πάγια διαδικασία. Έγραψαν λοιπόν μια συνάρτηση:

```
int DialogBox( HINSTANCE hInstance,
               PCTSTR pTemplate,    // όνομα του πλαισίου διαλόγου
               HWND hWndParent,
               DLGPROC pDialogFunc )    // βέλος προς τη συνάρτηση
                           // διαχείρισης του πλαισίου διαλόγου
```

Στη δεύτερη παράμετρο περιμένει το όνομα του πλαισίου διαλόγου (από αυτό θα βρει το σχέδιο). Στην τέταρτη παράμετρο περιμένει βέλος προς τη συνάρτηση διαχείρισης του πλαισίου διαλόγου που είναι μια συνάρτηση ανάκλησης.

Η *DialogBox*, αφού κάνει αυτά που γίνονται για κάθε πλαίσιο διαλόγου, θα καλέσει τη συνάρτηση ανάκλησης για τη διαχείριση του συγκεκριμένου πλαισίου. Παράδειγμα επικεφαλίδας τέτοιας συνάρτησης:

```
bool CALLBACK about( HWND hDlg, UINT iMessage,
                     WPARAM wParam, LPARAM lParam )
```

ή, απλούστερα:

```
bool about( HWND hDlg, unsigned int iMessage,
            unsigned short int wParam, long int lParam )
```

Και η αντίστοιχη κλήση:

```
        DialogBox( hInstance, "AboutBox", hWnd, &about );
```

ή:

```
        DialogBox( hInstance, "AboutBox", hWnd, about );
```

Τα παραπάνω δείχνουν παραδείγματα που χρησιμοποιείς βέλη προς συναρτήσεις (και δεν μπορείς να αποφύγεις.) Μπορεί να δεις και άλλες «εξωτικές» (ή, για άλλους, «τρομακτικές») περιπτώσεις αλλά να έχεις υπόψη σου ότι αναφέρονται κατά κύριο σε προγραμματισμό στη γλώσσα C. Στη C, με παρόμοιες τεχνικές, υλοποιούνται «γενικές» συναρτήσεις. Η C++ σου δίνει εργαλεία που (θα τα δούμε στη συνέχεια) που σου επιτρέπουν να αποφεύγεις τις «εξωτικές» περιπτώσεις.

Στο (Haendel 2001) θα βρεις πολλά πράγματα για βέλη προς συναρτήσεις, για τα προβλήματα που χρησιμοποιούνται και σχετικά εργαλεία και τεχνικές της C++. Στο –πιο απλό– (Hosey 2007) θα βρεις πολλά πράγματα για βέλη στη C και στη C++.

Εδώ θα δώσουμε μερικά παραδείγματα άλλων συνδυασμών συναρτήσεων και βελών που μπορεί να συναντήσεις.

Ξεκινούμε με την πιο απλή αλλά και πολύ συνηθισμένη περίπτωση: *συνάρτηση που επιστρέφει βέλος*. Τέτοιες συναρτήσεις έχουμε δει ήδη: *strcpy*, *strcat* και άλλες παρόμοιες. Στις περιπτώσεις αυτές όμως το αποτέλεσμα «έβγαινε» από δυο μεριές: ως αποτέλεσμα της συνάρτησης και ως τιμή της πρώτης παραμέτρου. Θα γράψουμε τώρα μια συνάρτηση:

```
const char* myStrLT( const char* s1, const char* s2 )
```

---

[5] Θα διαβάσεις αλλού ότι «συναρτήσεις ανάκλησης είναι αυτές που καλούμε με βέλος».

[6] Windows Application Program Interface (διεπαφή προγράμματος εφαρμογής).

που θα επιστρέφει ως τιμή βέλος προς τον ορμαθό –*s1* ή *s2*– που προηγείται λεξικογραφικώς. Προφανώς, είναι πολύ απλή:

```
const char* myStrLT( const char* s1, const char* s2 )
{
   const char* fv( s1 );
   if ( strcmp(s2, fv) < 0 ) fv = s2;
   return fv;
} // myStrLT
```

Συναρτήσεις που θα βγάζουν βέλη προς πίνακες διαφόρων τύπων –και όχι μόνον `char`– θα γράψουμε αρκετές.

Ας έλθουμε τώρα στο δεύτερο παράδειγμα. Τι είναι εκείνο το "`DLGPROC`"; Αν ψάξεις στο winuser.h θα δεις ότι είναι όνομα τύπου. Σε απλουστευμένη μορφή:

```
typedef int (*DLGPROC)( HWND, unsigned int,
                        unsigned short int, long int );
```

Δηλαδή έχουμε έναν τύπο βέλους προς συνάρτηση! Ο τύπος του `&`*about* είναι ακριβώς *DLGPROC*.

Παρομοίως, στην προηγούμενη παράγραφο, θα μπορούσαμε να ορίσουμε:

```
typedef double Real1( double );
```

Ο *Real1* είναι ο τύπος που περιλαμβάνει τις συναρτήσεις με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας τον θα μπορούσαμε να γράψουμε:

```
void bisection( Real1* f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Θα μπορούσαμε επίσης να ορίσουμε:

```
typedef double (*PReal1)( double );
```

Ο *PReal1* είναι ο τύπος που περιλαμβάνει *βέλη* προς συνάρτηση με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας αυτόν τον τύπο η επικεφαλίδα της *bisection* γράφεται:

```
void bisection( PReal1 f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Να δούμε τώρα *έναν πίνακα συναρτήσεων*! Ας πούμε ότι θέλουμε έναν πίνακα συναρτήσεων[7] τύπου *Real*1. Εδώ υπάρχει ένα πρόβλημα: αν δοκιμάσεις να δηλώσεις:

```
   Real1 funcArr[3];
```

ο μεταγλωττιστής δεν θα το επιτρέψει. Γιατί; Όλα τα στοιχεία του πίνακα πρέπει να έχουν το ίδιο μέγεθος· πόσο θα είναι αυτό; Αν δηλώσεις:

```
   PReal1 funcArr[3];
```

όλα πάνε καλά!

Γυρίζουμε στο παράδειγμα της προηγούμενης παραγράφου και το εμπλουτίζουμε με μια ακόμη συνάρτηση (για τη λύση της εξίσωσης $e^x = x+2$):

```
double em2( double x )
{
   return exp(x)-x-2;
} // em2
```

Έστω ότι θέλουμε να προσεγγίσουμε ρίζες των εξισώσεων:

$$x = lnx + 2 \text{ για } x \in [0.1,1], \quad συνx = 0 \text{ για } x \in [0,\pi], \quad e^x = x+2 \text{ για } x \in [0,2]$$

Μπορούμε να το κάνουμε ως εξής;

```
   const double pi( 4*atan(1.0) );
```

[7] «Τι να τον κάνουμε;» θα ρωτήσεις. Αν γράφεις πρόγραμμα C++ μάλλον δεν θα χρειαστείς τέτοιο πράγμα. Αν γράφεις C...

```
   PReal1 funcArr[3];
   double a[3], b[3];
   double riza;
   int   errCode;

   funcArr[0] = &q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = &cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = &em2;  a[2] = 0.0;  b[2] = 2.0;

   for ( int k(0); k < 3; ++k )
   {
      bisection( funcArr[k], a[k], b[k], 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
   }
```

Μην αμφιβάλλεις ότι μπορούμε να δώσουμε απλούστερα:

```
   funcArr[0] = q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = em2;  a[2] = 0.0;  b[2] = 2.0;
```

Ακόμη, μπορούμε να δηλώσουμε τον πίνακα χωρίς να ορίσουμε τον *PReal*1:

```
   double (*funcArr[3])( double );
```

Τέλος, ας δούμε και μια *συνάρτηση που επιστρέφει (βέλος προς) συνάρτηση*. Γράφουμε την:

```
PReal1 funcSel( PReal1 funcArr[], int n, int sel )
{
   return funcArr[sel];
}
```

Όπως βλέπεις αυτή επιστρέφει βέλος προς συνάρτηση και μπορούμε να γράψουμε (για λόγους επίδειξης και μόνον):

```
   bisection( funcSel(funcArr, 3, 1), a[1], b[1], 1e-5,
              riza, errCode );
```

Βεβαίως, θα μπορούσαμε να παραλείψουμε τους ορσμούς των ενδιάμεσων τύπων και να γράψουμε κατ' ευθείαν:

```
double (*(funcSel(double (*funcArr[])(double),
                  int n,int sel )))(double)
{
   return funcArr[sel];
}
```

Δεν σου αρέσει, ε;! Ας ελπίσουμε ότι δεν θα χρειαστεί να γράψεις τέτοιες συναρτήσεις.

Τελειώσαμε; Προς το παρόν: Ναι! Αλλά, υπάρχουν και χειρότερα που θα τα δεις αργότερα…[8]

14.5 Επιφόρτωση Συναρτήσεων

Στην §13.9.3 είδαμε τη συνάρτηση *swap()* που είναι πολύ χρήσιμη για όλους τους τύπους και όχι μόνον για τον `int`.

Να τη γράψουμε λοιπόν και για άλλους τύπους! Και για να μην αλλάζουμε πολύ το όνομα από τον έναν τύπο στον άλλο, θα κάνουμε το εξής: θα αλλάξουμε αυτήν που γράψαμε σε *swapI*, θα γράψουμε μια *swapD*, για μεταβλητές τύπου `double`, μια *swapC*, για μεταβλητές τύπου `char`… Ναι αυτά θα κάνουμε μόνο που δεν χρειάζεται να αλλάζουμε ονόματα.

Το πρόβλημα «*swap* για διάφορους τύπους» μπορεί να λυθεί έτσι:

```
#include <iostream>
```

[8] Έλα, αστείο ήταν!

```
using namespace std;

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

void swap( int& x, int& y );
void swap( double& x, double& y );
void swap( char& x, char& y );
void swap( WeekDay& x, WeekDay& y );

int main()
{
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

void swap( int& x, int& y )
{
   int s( x );
   x = y;  y = s;
} // swap int

void swap( double& x, double& y )
{
   double s( x );
   x = y;  y = s;
} // swap double

void swap( char& x, char& y )
{
   char s( x );
   x = y;  y = s;
} // swap char

void swap( WeekDay& x, WeekDay& y )
{
   WeekDay s( x );
   x = y;  y = s;
} // swap WeekDay
```

Εδώ, γράψαμε *τέσσερις διαφορετικές συναρτήσεις* –για τέσσερις τύπους (`int`, `double`, `char`, `WeekDay`)– *με το ίδιο όνομα*. Ο μεταγλωττιστής διαλέγει τη συνάρτηση που ταιριάζει στην κάθε κλήση ανάλογα με τους τύπους των ορισμάτων που βρίσκει σε αυτήν. Αυτό που κάναμε ονομάζεται **επιφόρτωση συναρτήσεων** (function overloading).

Το παράδειγμά μας δείχνει μια καλή χρήση επιφόρτωσης συναρτήσεων· *όλες οι συναρτήσεις που γράψαμε κάνουν την ίδια δουλειά* σε μεταβλητές διαφόρων τύπων.

Μπορείς όμως να γράφεις συναρτήσεις με το ίδιο όνομα που να κάνουν τελείως διαφορετικά πράγματα. Για παράδειγμα, μαζί με τις άλλες *swap()* να επιφορτώσεις και αυτήν:

```
double swap( int a, double b ) {  return (a+b)/2;  }
```

Ο μεταγλωττιστής δεν έχει αντίρρηση, αλλά κάτι τέτοιο δεν είναι και η πιο σοφή ιδέα για ένα πρόγραμμα.

Ας δούμε τώρα μερικά προβλήματα που μπορεί να συναντήσεις με τις επιφορτώσεις συναρτήσεων. Ας πούμε ότι ορίζεις:

```
void f( int& x, int& y ) { /*...*/ }
int f( int x, int y ) { /*...*/ return ...; }
```

```
void f( int& x, int y=0 ) { /*...*/ }
```

και μέσα στο πρόγραμμά σου δίνεις:

```
   int j1, j2;
// . . .
   f( 17, j2 );
   f( j1 );
   f( j1, j2 );
```

Βάλε τώρα τον εαυτό σου στη θέση του μεταγλωττιστή: Ποιαν από τις τρεις συναρτήσεις θα διάλεγες;

- Για την πρώτη περίπτωση τη δεύτερη συνάρτηση· είναι η μόνη που μπορεί να δεχθεί το "`17`" ως πρώτο όρισμα
- Για τη δεύτερη περίπτωση την τρίτη συνάρτηση· είναι η μόνη που μπορεί να κληθεί με ένα όρισμα.
- Για την τρίτη περίπτωση; Οποιαδήποτε! Και οι τρεις ταιριάζουν!

Αυτό το πρόβλημα μπορεί να σου παρουσιάζεται συχνά. Για να το λύνεις θα διαβάζεις προσεκτικά το διαγνωστικό που έβγαλε ο μεταγλωττιστής και θα σκέφτεσαι τα εξής:

- ***Ο μεταγλωττιστής επιλέγει –μεταξύ συναρτήσεων με το όνομα που δίνεται στην κλήση– την «πιο κοντινή» με βάση την υπογραφή (signature) της συνάρτησης, που για τη C++ είναι: το όνομα της συνάρτησης και οι τύποι των παραμέτρων.***[9]

Πιο συγεκριμένα:

- Πρώτη προτεραιότητα δίνεται σε συνάρτηση που οι τύποι των παραμέτρων της είναι ακριβώς ίδιοι με αυτούς των ορισμάτων της κλήσης. Έτσι, αν έχουμε τις συναρτήσεις:

```
void f( char x )                    // (a)
void f( int x )                     // (b)
void f( double x )                  // (c)
```

και δώσουμε:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (a).

- Δεύτερη προτεραιότητα δίνεται σε συναρτήσεις που οι τύποι των παραμέτρων τους είναι δυνατόν να προκύψουν από αυτούς των ορισμάτων της κλήσης με *ακέραιη προαγωγή*[10] ή κάποια από τις `float` σε `double` και `double` σε `long double`. Έτσι αν έχεις τις

```
void f( int x )                     // (b)
void f( double x )                  // (c)
```

για την:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (b).

- Τρίτη προτεραιότητα έχουν συναρτήσεις που οι τύποι των ορισμάτων τους προκύπτουν από αυτόν του ορίσματος με πάγιες μετατροπές τύπου όπως `int` σε `double`, `double` σε `int` και άλλες που θα μάθουμε αργότερα. Έτσι, αν δεν υπάρχουν οι επιλογές (a) και (b) ο μεταγλωττιστής θα δεχθεί τη (c).[11]

---

[9] Προσοχή! Ο επιστρεφόμενος τύπος δεν είναι μέρος της υπογραφής για τη C++· έτσι, στο παράδειγμά μας (τρίτη περίπτωση) ταιριάζει και η δεύτερη συνάρτηση.

[10] Δες το Παράρ. Ε.

[11] Υπάρχουν άλλα δύο σκαλοπάτια στην κλίματα προτεραιότητας που έχουν σχέση με πράγματα που θα μάθουμε αργότερα.

14.6 Επιφόρτωση Τελεστών

Οι τελεστές είναι συναρτήσεις. Ας πάρουμε τον "`+`". Μπορείς να τον δεις ως:

+: `int`×`int` → `int`

Ναι, αλλά μπορώ να γράψω και `1.2 + 3`. Φυσικά! Έχουμε επιφόρτωση των:

+: `double`×`int` → `double`

+: `int`×`double` → `double`

και βέβαια δεν τελειώσαμε εδώ. Ως άσκηση, σκέψου μερικές ακόμη συναρτήσεις «επιφορτωμένες» στον "`+`".

Η C++ μας επιτρέπει να επιφορτώνουμε όποιον (σχεδόν) θέλουμε από τους τελεστές της, όπως επιφορτώνουμε τις (άλλες) συναρτήσεις. Έχει όμως κάποιους περιορισμούς:

- *Δεν μπορούμε να πειράξουμε τους τελεστές που είναι ορισμένοι στους πρωτογενείς τύπους*. Π.χ. δεν μπορείς να επιφορτώσεις τον τελεστή "`*`" για πράξη μεταξύ ακεραίων. Μπορούμε να επιφορτώνουμε τελεστές για δικούς μας τύπους.
- *Δεν μπορούμε να αλλάξουμε το συντακτικό χρήσης των τελεστών*: αν ένας τελεστής είναι ενικός, όπως ο "`++`", θα πρέπει να επιφορτωθεί ως ενικός, αν είναι δυαδικός, όπως ο "`/`", θα πρέπει να επιφορτωθεί ως δυαδικός.
- *Δεν επιτρέπεται να επιφορτώσουμε τους τελεστές*: "`::`", "`.`", "`.*`", "`?:`".[12]

Και γιατί να επιφορτώσουμε έναν τελεστή, θα ρωτήσεις. Για να κάνουμε τη ζωή μας (προγραμματιστικώς) πιο εύκολη. Αν, για παράδειγμα, η *wd* είναι τύπου *WeekDay*, είναι προτιμότερο να προχωρούμε στην επόμενη μέρα εβδομάδας γράφοντας `++wd` αντί για `next(wd)` ή `advance(wd)`. Βεβαίως, να τονίσουμε ότι, αν επιφορτώσουμε για τη δουλειά αυτή τον "`--`", τότε... καλύτερα να μην επιφορτώσουμε! Να θυμάσαι πάντοτε έναν βασικό κανόνα:[13]

- ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***
  Από τεχνική άποψη:
- ***Η επιφόρτωση ενός τελεστή, ας πούμε του "`@`", γίνεται με την επιφόρτωση της συνάρτησης "`operator@()`".***

Προς το παρόν, ο μόνος τύπος που έχουμε ορίσει είναι ο *WeekDay* (§4.8). Στη συνέχεια λοιπόν θα δώσουμε παραδείγματα επιφόρτωσης των τελεστών "`++`" (ενικός) και "`<<`" (δυαδικός) για τον τύπο αυτόν και θα δεις τις τεχνικές λεπτομέρειες της επιφόρτωσης.

14.6.1 Τελεστής για Έξοδο Στοιχείων Τύπου *WeekDay*

Θέλουμε να επιφορτώσουμε τον "`<<`" έτσι ώστε αν η *wd*, τύπου *WeekDay*, έχει τιμή `sunday` και δώσουμε:

```
tout << wd << endl;
```

να γραφεί στο αρχείο text που έχει συνδεθεί με το *tout*:

```
Κυριακή
```

Αν, αντί για *tout*, γράψουμε στο *cout* θα πρέπει να το δούμε στην οθόνη.

Ένας δυαδικός τελεστής, σαν τον "`<<`", υλοποιείται (επιφορτώνεται) με συνάρτηση που έχει δύο παραμέτρους και όνομα "`operator<<`":

- Η πρώτη αντιστοιχεί στο όρισμα που εμφανίζεται αριστερά του "`<<`", στο παράδειγμά μας το "`tout`".

---

[12] Αργότερα θα τους μάθουμε όλους!

[13] Αυτή είναι και η σύσταση DCL11 του (CERT 2009): "Preserve operator semantics when overloading operators."

- Η δεύτερη αντιστοιχεί στο όρισμα που εμφανίζεται δεξιά του "`<<`", στο παράδειγμά μας το "`wd`".

  Αν θυμηθούμε τώρα ότι, όπως λέγαμε στην §13.9.2:

- «*Η* `cout << " x = "` *είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.» (Για την περίπτωσή μας να σκέφτεσαι "`tout`" αντί για "`cout`")
- «*Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* ***in out*** *(αναφοράς).*»
- «*Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο cout βάζε παράμετρο τύπου* `ostream` *και όχι* `ofstream`.»

καταλήγουμε στο συμπέρασμα ότι η επιφόρτωση θα πρέπει να γίνει με συνάρτηση που έχει επικεφαλίδα:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
```

και όχι "`void operator<<(…`" όπως λένε οι κανόνες μας.

Τα υπόλοιπα είναι απλά:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   string fv;
   switch ( rhs )
   {
      case sunday:    fv = "Κυριακή";  break;
      case monday:    fv = "Δευτέρα";  break;
      case tuesday:   fv = "Τρίτη";  break;
      case wednesday: fv = "Τετάρτη";  break;
      case thursday:  fv = "Πέμπτη";  break;
      case friday:    fv = "Παρασκευή";  break;
      case saturday:  fv = "Σάββατο";  break;
   } // switch
   return ( tout << fv );
} // operator<< WeekDay
```

Βέβαια, μπορούμε να το γράψουμε και έτσι:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   switch ( rhs )
   {
      case sunday:    return ( tout << "Κυριακή" );
      case monday:    return ( tout << "Δευτέρα" );
      case tuesday:   return ( tout << "Τρίτη" );
      case wednesday: return ( tout << "Τετάρτη" );
      case thursday:  return ( tout << "Πέμπτη" );
      case friday:    return ( tout << "Παρασκευή" );
      case saturday:  return ( tout << "Σάββατο" );
   } // switch
} // operator<< WeekDay
```

Αυτή η μορφή είναι πιο γρήγορη από την πρώτη αλλά παραβιάζει (άλλον) έναν από τους κανόνες μας: έχει πολλές `return`!

Όπως θα δούμε και στη συνέχεια, έτσι θα επιφορτώνουμε τους τελεστές εξόδου (αλλά και εισόδου) για όποιους τύπους μας ενδιαφέρει.

14.6.2 Ο Τελεστής "++" για τον Τύπο *WeekDay*

Όπως, πιθανότατα, μαντεύεις, αυτός ο τελεστής θα επιφορτωθεί με συνάρτηση που έχει όνομα "`operator++()`" και μια παράμετρο. Θέλουμε να τον ορίσουμε έτσι που *να μας δίνει την επόμενη ημέρα εβδομάδας*, όπως για ακέραιους μας δίνει τον επόμενο ακέραιο. Αν δεν δώσουμε δικό μας ορισμό, οι

```
   WeekDay d;
   . . .
```

```
   ++d;
```

δεν έχουν πρόβλημα αν η *d* έχει τιμή από *sunday* μέχρι *friday*: η τιμή της *d* προχωρεί στην επόμενη μέρα (ας πούμε: `d = d + 1`). Αν όμως η *d* έχει τιμή *saturday* τότε παίρνει τιμή έξω από τα όρια του τύπου ενώ θα έπρεπε να «ξαναγυρίζει» στη *sunday*.

Να πώς διορθώνουμε αυτό το πρόβλημα:[14]

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday ) lhs += 1;
                    else lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Ας δούμε τώρα μια δοκιμή (με οποιαδήποτε από τις δύο μορφές του `operator<<`):

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

enum WeekDay { sunday, def, tuesday, wednesday, thursday, friday,
               saturday };

ostream& operator<<( ostream& tout, WeekDay rhs );
WeekDay operator++( WeekDay& lhs );

int main()
{
// . . .
   WeekDay d( sunday );
   for ( int k(0); k < 10; ++k )
   {
      cout << d << endl;
      ++d;
   }
// . . .
} // main
```

Αποτέλεσμα:

```
Κυριακή
Δευτέρα
Τρίτη
Τετάρτη
Πέμπτη
Παρασκευή
Σάββατο
Κυριακή
Δευτέρα
Τρίτη
```

[14] Η Borland C++ v.5.5 θα δεχτεί αυτήν τη μορφή. Η gcc θα σου πει ότι δεν έχει ορισθεί ο "`+=`" για τον τύπο *WeekDay*. Στην περίπτωση αυτή γράψε την επιφόρτωση ως εξής:

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday )
   {
      int iv( static_cast<int>(lhs) );
      ++iv;
      lhs = static_cast<WeekDay>(iv);
   }
   else
      lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Πού καλείται η `operator++()`; Στην εντολή "`++d`"! Δεν το πιστεύεις; Λοιπόν άλλαξε τη "`++d`" σε:

```
operator++( d );
```

Εδώ δεν έχεις πρόβλημα: Καλείται η συνάρτηση `operator++()` με όρισμα *d*. Ε, το πρόγραμμά σου θα περάσει από τον μεταγλωττιστή και εκτελούμενο θα δώσει ακριβώς τα ίδια αποτελέσματα που πήραμε παραπάνω! Μπορείς να θεωρήσεις ότι αυτή είναι η «κανονική» κλήση ενώ η "`++d`" είναι μια συντομογραφία με οικείο συντακτικό.

Και πού καλείται η `operator<<()`; Στην "`cout << d`". Για να το πιστέψεις, άλλαξε και εδώ τη "`cout << d << endl`" σε:

```
operator<<( cout, d ) << endl;
```

και θα δεις ότι –και πάλι– το πρόγραμμά θα δώσει τα ίδια αποτελέσματα! Και εδώ, μπορείς να θεωρήσεις ότι η η "`cout << d`" είναι συντομογραφία ενώ «κανονική» κλήση είναι η "`operator<<( cout, d )`".

Παρατήρηση: ▶

Η `operator++()` δεν είναι συμβατή με τους κανόνες μας: αλλάζει την τιμή της παραμέτρου και επιστρέφει τιμή. Εδώ όμως προτιμούμε να είμαστε συμβατοί με τη φιλοσοφία της C++ για τους τελεστές "`++`" και "`--`".

Στην επόμενη υποπαράγραφο θα μάθουμε ότι η «φιλοσοφία της C++» (για την ακρίβεια της C) απαιτεί τύπο αποτελέσματος της συνάρτησης `operator++()` όχι `WeekDay` αλλά `WeekDay&`.◀

Και αν θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" τι κάνουμε; Κάτι «παράξενο»:

```
WeekDay operator++( WeekDay& lhs, int )
{
   WeekDay sv( lhs );
   ++lhs;
   return sv;
} // operator++( WeekDay, int
```

Η (ανύπαρκτη) δεύτερη παράμετρος είναι *σήμα προς τον μεταγλωττιστή ότι εδώ μιλάμε για τον μεταθεματικό τελεστή* και όχι για τον προθεματικό.

Όσο για την υλοποίηση: χρησιμοποιούμε τον προθεματικό που έχουμε ήδη ορίσει.

14.6.3 Η Πράξη της Εκχώρησης (ξανά)

Έχουμε μάθει ότι με την `v =` *Π* γίνονται τα εξής:

- Υπολογίζεται η τιμή της παράστασης *Π* και μετατρέπεται στον τύπο της μεταβλητής `static_cast<`*T*`>(`*Π*`)`.
- Η τιμή `static_cast<`*T*`>(`*Π*`)` φυλάγεται ως τιμή της μεταβλητής.
- Αποτέλεσμα της πράξης είναι η `v`.

Σε αυτό βασίζεται και η πολλαπλή εκχώρηση:

```
w = v = u = 0;
```

που εκτελείται ως:

```
w = (v = (u = 0));
```

Τι θα έλεγες τώρα αν έβλεπες:

```
(w = v) = u;
```

Δεν σου κάνει νόημα, ε; Δες το παρακάτω πρόγραμμα:

```
#include <iostream>

using namespace std;

int main()
```

```
{
   int u(1), v(2), w(3);
   (w = v) = u;
   cout << u << "  " << v << "  " << w << endl;
}
```

Αποτέλεσμα:

```
1  2  1
```

Η πρώτη έκπληξη είναι ότι έγινε δεκτή, δηλαδή η "`(w = v)`" θεωρήθηκε ως τιμή-*l*. Η δεύτερη έκπληξη είναι ότι η *w* πήρε τιμή "`1`" (της *u*) και όχι "`2`" (της *v*).

Παρόμοια ισχύουν και για τις συντομογραφίες της εκχώρησης. Για παράδειγμα οι:

```
   cout << u << endl;
   (u += 7) = 11;
   cout << u << endl;
   (++u) = 17;
   cout << u << endl;
```

δίνουν:

```
1
11
17
```

Πώς εξηγούνται τα παραπάνω; Με την υποσημείωση της §11.3 «*Πρόσεξε: «η* `v`*» και όχι «η τιμή της* `v`*». Θα καταλάβεις αργότερα...*» Το «αργότερα» είναι τώρα.

- Ας πάρουμε την "`(w = v) = u`". Η τιμή της *v* αποθηκεύεται στην *w*, αλλά το αποτέλεσμα της πράξης "`w = v`" είναι *η μεταβλητή w*, δηλαδή μια τιμή-*l*. Έτσι, στη συνέχεια εκτελείται η πράξη "`w = u`" και η *w* παίρνει την τιμή της *u* που είναι "`1`".
- Ας πάρουμε την "`(++u) = 17`" που είναι ισοδύναμη με "`(u=u+1) = 17`". Η "`u=u+1`" θα επιστρέψει ως αποτέλεσμα τη μεταβλητή *u* και στη συνέχεια θα εκτελεσθεί η "`u = 17`".

Για να είμαστε σύμφωνοι με αυτά, όταν επιφορτώνουμε οποιονδήποτε τελεστή εκχώρησης θα πρέπει να βγάζουμε ως αποτέλεσμα της πράξης τη «μεταβλητή του αριστερού μέρους». Θα βάλουμε λοιπόν ως τύπο αποτελέσματος όχι τον τύπο *T* της μεταβλητής αλλα τον *T*`&`.

- ***Για να μην αποκλίνουμε από τη «φιλοσοφία της C++» (της C) θα πρέπει και εμείς να επιφορτώνουμε τους τελεστές εκχώρησης ως:***

  ```
  T& operator=( T& lhs, const T& rhs )
  T& operator+=( T& lhs, const T& rhs )
  T& operator++( T& lhs )
  ```

  (*όχι* `const T&`).[15] ***Τα ίδια ισχύουν και για τους "--", "-=", "*=" κλπ.***

  Επομένως, θα ξαναγράψουμε την επιφόρτωση του

```
WeekDay& operator++( WeekDay& v )
{
   if ( v < saturday ) v += 1;
                  else v = sunday;
   return v;
} // operator++( WeekDay
```

Για τους μεταθεματικούς "`++`", "`--`" ισχύουν αυτά που είπαμε στην προηγούμενη παράγραφο. Στην περίπτωση αυτή δεν μπορούμε να επιστρέψουμε τύπο αναφοράς, αφού επιστρέφεται μια μεταβλητή (*sv*) τοπική στη συνάρτηση.

14.6.4 Γενικώς ...

Γενικώς, μπορούμε να πούμε ότι

[15] Δεν θα παραλείψουμε όμως να τονίσουμε ότι η σύσταση 34 της (ELLEMTEL 1992) λέει: "An assignment operator ought to return a `const` reference to the assigning object."

- Επιφορτώνουμε έναν *δυαδικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( Tl lhs, Tr rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *Tl* ο τύπος της πρώτης παραμέτρου και *Tr* ο τύπος της δεύτερης παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, y )...`" είτε ως "`...x @ y...`". Στην πρώτη παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*) και στη δεύτερη (*rhs*) αυτό που εμφανίζεται μετά από αυτόν (*y*).
- Επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( T rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x )...`" είτε ως "`...@x...`". Στη μοναδική παράμετρο (*rhs*) θα πάει το όρισμα που εμφανίζεται μετά από τον τελεστή (*x*).
- Επιφορτώνουμε έναν *μεταθεματικό ενικό* τελεστή `@` (όταν υπάρχει και *προθεματικός ενικός* τελεστής `@`) με μια συνάρτηση
  `Trv operator@( T lhs, int )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, 1 )...`" είτε ως "`...x@...`". Στην πρώτη (και στην πραγματικότητα) μοναδική παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*).

Αργότερα θα δούμε ότι για κλάσεις μερικά από αυτά τα πράγματα θα αλλάξουν. Αλλά –να επαναλάβουμε με άλλα λόγια– σε κάθε περίπτωση, το ουσιώδες είναι το εξής:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

14.7 Γενικές Συναρτήσεις

Στην §14.5 γράψαμε και επιφορτώσαμε τέσσερις φορές τη *swap*. Αλλά, η μόνη διαφορά στις τέσσερις συναρτήσεις είναι ο τύπος των παραμέτρων. Ο μεταγλωττιστής της C++ μας δίνει την εξής δυνατότητα: να του δώσουμε το σχέδιο –το «πατρόν– και να γράψει εκείνος όποιες συναρτήσεις θα χρειαστούν. Αυτό το σχέδιο λέγεται **περίγραμμα συνάρτησης** (function template).

Για το περίγραμμα θα δεις και τον όρο **γενική συνάρτηση** (generic function). Πάντως, όπως θα καταλάβεις στη συνέχεια *μια γενική συνάρτηση δεν είναι συνάρτηση αφου δεν έχει συγκεκριμένο πεδίο ορισμού ή/και πεδίο τιμών*.

14.7.1 Περιγράμματα Συναρτήσεων

Ξαναγράφουμε το παράδειγμα της §14.5 ως εξής:

```
#include <iostream>

using std::cout;
using std::endl;

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

template < typename T >
void swap( T& x, T& y );

int main()
{
```

```
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Η βασική διαφορά[16] είναι: αντί για τις τέσσερις συναρτήσεις έχουμε το:

```
template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Αυτό είναι ένα **περίγραμμα** (template) της συνάρτησης *swap* με παράμετρο τον τύπο *T*. Αντί για "`typename T`" θα μπορούσαμε να γράψουμε και "`class T`". Τα "`class`", "`template`" και "`typename`" είναι λέξεις-κλειδιά της C++.

Το πρόγραμμα αυτό δίνει:

```
20  10
2.34  1.23
B  A
2  0
```

Όταν ο μεταγλωττιστής βρει τη "`swap(j1, j2)`" δημιουργεί ένα **στιγμιότυπο** (instance) του περιγράμματος σε μια συνάρτηση σαν την

```
void swap( int& x, int& y )
```

Το στιγμιότυπο δημιουργείται *αυτομάτως* αφού δεν υπάρχει αμφιβολία για το ότι θα πρέπει να βάλει όπου *T* τον `int`.

Παρομοίως, όταν βρεί τη "`swap(d1, d2)`" ο μεταγλωττιστής θα δημιουργήσει ένα στιγμιότυπο:

```
void swap( double& x, double& y )
```

κ.ο.κ.

Ένα περίγραμμα μπορεί να έχει ως παραμέτρους

- έναν ή περισσότερους *τύπους* ή/και
- *ακέραιους*,
- *βέλη* ή *αναφορές*.

και μπορεί να μας δώσει μια συνάρτηση –που τη λέμε **στιγμιότυπο του περιγράμματος** (template instance)– με δύο τρόπους:

- Όπως στα παραπάνω παραδείγματα, όπου οι τύποι που θα αντικαταστήσουν τους τύπους-παραμέτρους συνάγονται αυτομάτως από τους τύπους των ορισμάτων της κάθε κλήσης. Αυτή είναι η **συναγόμενη δημιουργία στιγμιοτύπου** (implicit instantiation).
- Με τη **ρητή** (explicit) δημιουργία στιγμιοτύπου κατά την οποία γράφουμε, στην κλήση, τις τιμές των παραμέτρων μετά το όνομα του περιγράμματος. Π.χ. στο παράδειγμά

[16] Υπάρχει και η «μικροδιαφορά» στο "`using`". Θα τη συζητήσουμε παρακάτω.

μας, αντί για `swap(d1, d2)`, θα μπορούσαμε να γράψουμε: `swap<double>(d1, d2)`. Ας δούμε ένα

Παράδειγμα 1

Έστω ότι θέλουμε να μετατρέψουμε σε περίγραμμα τη *vectorSum* (§12.1) Μια απλή προσπάθεια μας οδηγεί στο εξής:

```
template < typename CT >
CT vectorSum( const CT x[], int n, int from, int upto )
{
   CT sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

Εδώ θα πρέπει να σκεφτούμε την εξής περίπτωση: Έστω ότι βάζουμε ως *CT* τον `double`. Παρ' όλο όμως που τα στοιχεία του πίνακα είναι `double` μπορεί το άθροισμα τους να είναι μεγαλύτερο από το *DBL_MAX*. Σε μια τέτοια περίπτωση θα μπορούσαμε να πάρουμε το σωστό αποτέλεσμα σε `long double`. Την αλλάζουμε λοιπόν:

```
template < typename CT, typename RT >
RT vectorSum( const CT x[], int n, int from, int upto )
{
   RT sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

Τώρα όμως έχουμε άλλο πρόβλημα: Έστω ότι έχουμε δηλώσει

```
   double ar[7971];
   long double res;
```

Αν δοκιμάσουμε να δώσουμε

```
   res = vectorSum( ar, 7971, 0, 7970 );
```

ο μεταγλωττιστής δεν θα μπορέσει να βγάλει άκρη. Το σωστό είναι να γράψουμε τις οδηγίες για τη δημιουργία στιγμιοτύπου:

```
   res = vectorSum<double, long double>( ar, 7971, 0, 7970 );
```

Όταν η παράμετρος είναι *ακέραιος*

- Αυτή μπορεί να χρησιμοποιηθεί στη συνάρτηση ως σταθερά.
- Το αντίστοιχο όρισμα θα πρέπει να είναι σταθερά.

Παράδειγμα 2

Στο Παράδ. 6 της §13.9.3 συζητήσαμε για δυο συναρτήσεις

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
void output2DAr( ostream& tout,
                 const int* a, int nRow, int nCol )
```

για να μη γράφουμε ξανά και ξανά τις ίδιες εντολές στο πρόγραμμα πολλαπλασιασμού πινάκων (Παράδ. 4, §12.4). Εκεί γράψαμε την πρώτη από αυτές και εσύ, πού έλυσες την Άσκ. 13-1, έχεις γράψει τη δεύτερη.

Τώρα όμως μας κατεβαίνει μια ιδέα: Αφού μπορούμε να βάλουμε στο περίγραμμα ακέραιη παράμετρο που έρχεται στη συνάρτηση ως σταθερά, να βάλουμε εκεί τον αριθμό στηλών του πίνακα και να δηλώσουμε τον πίνακα ως δισδιάστατο:

```
template< typename T, unsigned int nCol >
void input2DAr( istream& tin, T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
```

```
} // input2DAr
```

Καλό; Εξαιρετικό! Να κάνουμε έτσι και την άλλη:

```
template < typename T, unsigned int nCol >
void output2DAr( ostream& tout,
                 const T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Τώρα, στο πρόγραμμά μας διαβάζουμε με τις:

```
   input2DAr< int, m >( atx, a, l );
   input2DAr< int, n >( atx, b, m );
```

γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr< int, m >( cout, a, l );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr< int, n >( cout, b, m );
   cout << " Στοιχεία του πίνακα c" << endl;
   output2DAr< int, n >( cout, d, l );
```

και όλα είναι μια χαρά![17] Πρόσεξε ότι εδώ *είναι απαραίτητη η ρητή δημιουργία στιγμιοτύπων* αφού ο αριθμός στηλών του πίνακα δεν υπάρχει στην κλήση ώστε να μπορεί να συναχθεί η τιμή της δεύτερης παραμέτρου.

Αλλά τίποτε δεν δίνεται δωρεάν στον κόσμο αυτόν! Αν χρησιμοποιήσουμε τις συναρτησεις με τον «γραμμοποιημένο» πίνακα το μέγεθος του εκτελέσιμου είναι 156160 ψηφιολέξεις ενώ με τα περιγράμματα έχουμε 156672.[18]

Γιατί; Διότι στην πρώτη λύση έχουμε δύο συναρτήσεις ενώ στη δεύτερη έχουμε πέντε: δύο *input2DAr* και τρεις *output2DAr*! Γράφηκαν αυτομάτως βέβαια, αλλά γράφηκαν.

Και μετά από αυτό, να απαντήσουμε και το ερώτημα που βάλαμε στην υποσημείωση: Αν θέλουμε παράμετρο για το εύρος πεδίου θα τη βάλουμε στη συνάρτηση. *Αν τη βάλεις ως παράμετρο του περιγράμματος για κάθε τιμή της θα έχουμε διαφορετικό στιγμιότυπο*· αυτό είναι υπερβολή.

Ας πούμε και κάτι ακόμη που –προς το παρόν– μπορεί να μην σου κάνει πολύ νόημα:

- Μπορείς να έχεις στιγμιότυπα της *swap* για οποιονδήποτε τύπο
  - έχει τον τελεστή εκχώρησης και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *vectorSum()* που έχουν για δεύτερο όρισμα (*RT*) οποιονδήποτε τύπο
  - έχει τον τελεστή "`+=`" ώς `operator+=( RT, CT )` και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *output2DAr* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`<<`" για έξοδο στοιχείων προς αρχείο text. Π.χ. μπορείς να έχεις στιγμιότυπο για τον *WeekDay*.

---

[17] Πάντως εκείνο το "`3`" στη "`tout.width(3)`" είναι μαγική σταθερά. Να το βάλουμε ως παράμετρο στη συνάρτηση ή μήπως στο περίγραμμα;

[18] Borland C++, v.5.5.

- Μπορείς να έχεις στιγμιότυπα της *input2Dar()* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`>>`" για είσοδο στοιχείων από αρχείο text. Π.χ. *δεν* μπορείς να δημιουργήσεις στιγμιότυπο για τον *WeekDay*.

Ας πούμε τώρα ότι θέλουμε να μετατρέψουμε σε περίγραμμα και τη *max*. Αυτό είναι εύκολο:

```
template< typename T >
T max( T x, T y )  {  return (x > y) ? x : y;  }
```

Από αυτό το περίγραμμα μπορούμε να πάρουμε στιγμιότυπα για οποιονδήποτε τύπο ορίζεται ο τελεστής "`>`", όπως είναι οι `int`, `double`, `char` κ.ο.κ. Και αν θέλουμε να συγκρίνουμε δύο ορμαθούς C (πίνακες χαρακτήρων); Τώρα *T* είναι ο `char*` και προφανώς δεν μπορούμε να πάρουμε στιγμιότυπο. Η C++ όμως μας δίνει τη δυνατότητα να κάνουμε μια **εξειδίκευση** (specialization) του περιγράμματος για τον `char*`:

```
template<>
char* max<char*>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

- Η δήλωση μιας εξειδίκευσης περιγράμματος αρχίζει με "`template<>`".
- Οι τιμές των παραμέτρων για την εξειδίκευση γράφονται μέσα σε "`<`", "`>`" μετά το όνομα του περιγράμματος.

Αν οι τιμές των παραμέτρων της εξειδίκευσης συνάγονται από τους τύπους των παραμετρων της συνάρτησης μπορείς να τις παραλείψεις. Για τη *max* μπορούμε να γράψουμε:

```
template<>
char* max<>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

Ακόμη, μπορείς να παραλείψεις τις παραμέτρους εξειδίκευσης αν α) υπάρχουν προκαθορισμένες τιμές για τις παραμέτρους του περίγράμματος και β) η εξειδίκευση γίνεται για τις προκαθορισμένες τιμές. Με τις προκαθορισμένες τιμές που βάλαμε στις παραμέτρους του *output2DAr()* γράφοντας:

```
template <>
void output2DAr<>( ostream& tout,
                   const int a[][nCol], int nRow )
// . . .
```

δηλώνουμε εξειδίκευση για τύπο `int` και *nCol* `==` 2.

14.7.1.1 Η «Μικροδιαφορά» στο "`using`"

Όταν αλλάξαμε το πρόγραμμα για να δοκιμάσουμε το περίγραμμα της *swap* βγάλαμε το γενικό

```
using namespace std;
```

και βάλαμε συγκεκριμένα:

```
using std::cout;
using std::endl;
```

Γιατί; Διότι στο *sdt* υπάρχει ήδη –μεταξύ άλλων– περίγραμμα *swap()* που μας έρχεται έτοιμο από τη βιβλιοθήκη της C++! Στα «άλλα» περιλαμβάνονται περιγράμματα για τις *min()*, *max()* και άλλες συνηθισμένες συναρτήσεις. Δοκίμασέ τα! Αν δεν φτάνει το "`using namespace std`" βάλε και "`#include <algorithm>`".

14.7.2 Επιφόρτωση στο Περίγραμμα

Ας γυρίσουμε στο περίγραμμα της *swap()* για να σκεφτούμε την εξής περίπτωση: μπορούμε να το χρησιμοποιήσουμε για να αντιμεταθέσουμε τις τιμές των:

```
   char s1[15], s2[20];
```

Όχι, φυσικά! Θα πρέπει να γράψουμε μια:

```
   void swap( char x[], char y[] )
```

και να την επιφορτώσουμε στο περίγραμμα που προϋπάρχει. Ας πούμε ότι τη γράφουμε:

```
void swap( char x[], char y[] )
{
   string s( x );
   strcpy( x, y );
   strcpy( y, s.c_str() );
} // swap( char*
```

Παρατηρήσεις: ▶

1. Γιατί «ας πούμε...»; Διότι για να δουλέψει θα πρέπει η τιμή του x να «χωράει» στον y και το αντίστροφο. Αυτό το αφήνουμε στον προγραμματιστή-χρήστη της συνάρτησης.
2. Για να τη δοκιμάσεις θα πρέπει –εκτός από τις "`using std::cout`" και "`using std::endl`"– να βάλεις και μια "`using std::string`".◀

Όταν ο μεταγλωττιστής βρει την εντολή:

```
   swap( s1, s2 );
```

θα κάνει τη σωστή επιλογή.

Αν θέλεις κάνε και το εξής πείραμα: όρισε

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }
```

δήλωσε:

```
   int k( 23 );
   double e( 7.33 );
```

και ζητήσε:

```
   qsc( k, 'a' );
   qsc( e, 'c' );
```

Η πρώτη κλήση θα εξυπηρετηθεί από εξειδίκευση του περιγράμματος ενώ η δεύτερη από την απλή συνάρτηση και θα πάρεις:

```
in template
in function
```

Μπορείς να κάνεις και επιφόρτωση περιγραμμάτων: Για παράδειγμα, μπορείς πέρα από τους παραπάνω ορισμούς της *qsc* να δώσεις και

```
template< typename T >
void qsc( int x, T c )
{  cout << "in template 2" << endl;  }
```

Δεν υπάρχει οποιοδήποτε πρόβλημα εκτός από την περίπτωση που θα βάλεις στο πρόγραμμά σου:

```
   qsc( k, 'a' );
```

Στην περίπτωση αυτήν ο μεταγλωττιστής δεν μπορεί να καταλάβει ποιο περίγραμμα εννοείς.

14.7.2.1 Επιφόρτωση, Εξειδίκευση και Άλλα Ψιλά Γράμματα...

Όταν ο μεταγλωττιστής βρει κλήση σε συνάρτηση που μπορεί να προκύψει από περίγραμμα που είναι επιφορτωμένο με άλλο (-α) περίγραμμα (-τα) ή/και απλή (-ές) συνάρτηση (-σεις) επιλέγει την κατάλληλη συνάρτηση ως εξής:

- Από όλα τα περιγράμματα με το όνομα συνάρτησης της κλήσης γίνεται προσπάθεια δημιουργίας στιγμιοτύπων που να ταιριάζουν με την κλήση.

- Από όλα τα στιγμιότυπα και τις απλές συναρτήσεις που τυχόν ταιριάζουν επιλέγεται η «πιο κοντινή» προς την κλήση με βάση αυτά που είπαμε στο τέλος της §14.5.
- Αν υπάρχει «ισοπαλία» στιγμιοτύπου περιγράμματος και απλής συνάρτησης προτεραιότητα έχει η απλή συνάρτηση.
- Αν η επιλογή είναι στιγμιότυπο κάποιου περιγράμματος και υπάρχει εξειδίκευση με την ίδια υπογραφή χρησιμοποιείται η εξειδίκευση.

Ας πούμε ότι έχουμε:

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }

template<> void qsc<double>( double x, char c )
{  cout << "in template spec" << endl;  }
```

Δηλαδή: γράφουμε μια εξειδίκευση του περιγράμματος ακριβώς για την περίπτωση που έχουμε την επιφόρτωση! Τα καταφέραμε να μπερδέψουμε τον μεταγλωττιστή με την κλήση:

```
   qsc( e, 'c' );
```

Όχι! Το πρόγραμμα θα μεταγλωττισθεί και θα δώσει:

```
in function
```

Ας δούμε γιατί γίνεται αυτό: Σύμφωνα με αυτά που είπαμε παραπάνω έχουμε να επιλέξουμε μεταξύ του στιγμιοτύπου:

```
void qsc<double>( double x, char c );
```

και της απλής συνάρτησης

```
void qsc( double x, char c );
```

Αφού στην κλήση "`qsc(e,'c')`" το πρώτο όρισμα είναι `double` και το δεύτερο `char` και τα δύο έχουν πρώτη προτεραιότητα. Αφού έχουμε «ισοπαλία» επιλέγεται η απλή συνάρτηση.

Όπως βλέπεις, στο παράδειγμά μας, ο μεταγλωττιστής δεν πρόκειται να φτάσει να εξετάσει την «υποψηφιότητα» της εξειδίκευσης του περιγράμματος για την κλήση "`qsc(e,'c')`".

Αν είχαμε επιλογή του στιγμιοτύπου του περιγράμματος τότε θα είχαμε χρήση της εξειδίκευσης.

14.8 Η Στοίβα

Στην επόμενη παράγραφο θα προσπαθήσουμε να δούμε πώς δουλεύει *ο μηχανισμός των εξαιρέσεων* της C++. Στη συνέχεια θα (ξανα)μιλήσουμε για *αναδρομή*. Και στις δύο περιπτώσεις θα αναφερθούμε στη στοίβα. Καλό είναι λοιπόν να πούμε από πριν δυο λόγια για αυτήν την πολύτιμη περιοχή μνήμης.

Μέχρι στιγμής έχουμε γνωρίσει δύο είδη[19] –από την άποψη της διαχείρισης– μνήμης:

- τη **στατική** (static), για τα στατικά αντικείμενα· οτιδήποτε βάλουμε εκεί δημιουργείται μια φορά και παραμένει στην ίδια διεύθυνση μέχρι το τέλος της εκτέλεσης του προγράμματος,
- την **αυτόματη** (automatic) ή μνήμη **στοίβας** (stack) για τα ορίσματα και τα τοπικά αντικείμενα συναρτήσεων και ομάδων· ό,τι βάλουμε εκεί ζει όσο ζει η αντίστοιχη

[19] Αργότερα θα μάθουμε και ένα τρίτο είδος: τη **δυναμική** μνήμη ή μνήμη **σωρού**.

συνάρτηση: δημιουργείται με την κλήση της και καταστρέφεται με το τέλος της εκτέλεσής της.

Το παρακάτω πρόγραμμα σου επιτρέπει να διακρίνεις αυτές τις δύο περιοχές:

```
#include <iostream>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long fl( double p )
{
   long s;

   cout << " &p = " << &p << "    &s = " << &s << endl;
   p = fd(p, 1.56);
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   y = fl(5);
   x[2] = fd( x[0], y );
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B178
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 &p = 0012FF44    &s = 0012FF38
 &p1 = 0012FF28    &p2 = 0012FF30
 &p1 = 0012FF3C    &p2 = 0012FF44
```

Εδώ, αφού πάρεις υπόψη σου ότι έχουμε δεκαεξαδικό σύστημα, παρατήρησε τα εξής:

- Η αποθήκευση της καθολικής μεταβλητής *g* έγινε σε άλλη περιοχή της μνήμης (0041B178) από αυτήν που αποθηκεύονται οι υπόλοιπες (0012FF...). Η *g* είναι στη στατική μνήμη ενώ οι υπόλοιπες είναι στη μνήμη στοίβας.
- Όπως αναμένεται, η διεύθυνση της παραμέτρου αναφοράς *a*3, στην *q*, είναι ίδια με αυτήν της *b* στη **main**.
- Η εκτέλεση του προγράμματος αρχίζει από τη **main**. Έτσι, οι πρώτες μεταβλητές που υλοποιούνται στη στοίβα είναι οι τοπικές της **main** στις διευθύνσεις από 0012FF54 (x) μέχρι 0012FF88 (*a*).
- Με την κλήση της *fl* υλοποιούνται και οι δικές της τοπικές μεταβλητές στη στοίβα από 0012FF38 (*s*) μέχρι 0012FF44 (*p*).
- Με την κλήση της *fd*, μέσα από την *fl*, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF28 (*p*1) μέχρι 0012FF30 (*p*2).
- Μετά την ολοκλήρωση της κλήσης της *fl*, όλη η μνήμη που χρησιμοποιήθηκε από την *fl* και την *fd* επιστρέφεται στη στοίβα. Αυτό φαίνεται από το επόμενο βήμα:

- Με την κλήση της *fd*, από τη **main**, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF3C (*p*1) μέχρι 0012FF44 (*p*2). Η μνήμη αυτή είχε χρησιμοποιηθεί προηγουμενως για την υλοποίηση των μεταβλητών της *fl*.

Αυτή η λειτουργία της αυτόματης μνήμης, ό,τι εισάγεται τελευταίο να εξάγεται πρώτο (Last In First Out), είναι χαρακτηριστικό της, συχνότατα χρησιμοποιούμενης, δομής στοιχείων που ονομάζεται **στοίβα** (stack, LIFO stack).

14.8.1 Η Συνάρτηση *stackavail*

Αν δουλεύεις με τον μεταγλωττιστή Borland C++, v.5.5 (ή v.5.02) μπορείς να δεις πιο καλά τη χρήση της στοίβας χρησιμοποιώντας τη συνάρτηση (εκτός προτύπου) *stackavail* που δίνει τη διαθέσιμη μνήμη στοίβας:

```
#include <iostream>
#include <malloc.h>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " μέσα στην fd, stackavail: " << stackavail() << endl;
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long int fl( double p )
{
   long s;

   cout << " μέσα στην fl, stackavail: " << stackavail() << endl;
   cout << " &p = " << &p << "    &s = " << &s << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   p = fd(p, 1.56);
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long int a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   cout << " πριν κληθεί η fl, stackavail: " << stackavail()
        << endl;
   y = fl(5);
   cout << " μετά την fl, stackavail: " << stackavail() << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   x[2] = fd( x[0], y );
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B17C
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 πριν κληθεί η fl, stackavail: 1048376
```

```
μέσα στην fl, stackavail: 1048356
&p = 0012FF40    &s = 0012FF34
πριν κληθεί η fd, stackavail: 1048356
μέσα στην fd, stackavail: 1048332
&p1 = 0012FF24    &p2 = 0012FF2C
μετά την fd, stackavail: 1048356
μετά την fl, stackavail: 1048376
πριν κληθεί η fd, stackavail: 1048376
μέσα στην fd, stackavail: 1048352
&p1 = 0012FF38    &p2 = 0012FF40
μετά την fd, stackavail: 1048376
```

Πρόσεξε τα εξής:

- Πριν κληθεί η *fl* έχουμε μνήμη στοίβας 1048376 ψηφιολέξεις. Μόλις άρχισε η εκτέλεσή της η διαθέσιμη μνήμη γίνεται 1048356 ψηφιολέξεις. Οι 20 ψηφιολέξεις διατίθενται για την υλοποίηση των μεταβλητών (8+4) και για τη διαχείριση της κλήσης. Η διαθέσιμη μνήμη στοίβας ξαναπαίρνει την αρχική της τιμή «`μετά την fl, stackavail: 1048376`».
- Η κάθε κλήση της *fd* κοστίζει 24 ψηφιολέξεις αλλά σε διαφορετική περιοχή της στοίβας κάθε φορά.

14.9 Διαχείριση Εξαιρέσεων με Δυο Λόγια

Ας ξεκινήσουμε με το παράδειγμα που είδαμε στην §7.8. Γράψαμε τη συνάρτηση *V*, που δεν ορίζονταν στα σημεία $\{ k: \mathbb{Z} \bullet 2k \}$:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      cerr << " η v κλήθηκε με όρισμα: " << x << endl;
      exit( EXIT_FAILURE );
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Εδώ εφαρμόσαμε το βήμα 7 της συνταγής του Πλ. 7.4 που λέει:
«*Αν η συνάρτηση είναι μερική, γράψε τις εντολές που εξαιρούν τις τιμές του ορίσματος για τις οποίες δεν ορίζεται η συνάρτηση:*

```
if (x δεν ανήκει στο πεδίο ορισμού)
{
   cerr << " η ... κλήθηκε με x = " << x << endl;
   exit( EXIT_FAILURE );
}
else
   Υπολόγισε την τιμή της συνάρτησης»
```

Φυσικά η συνταγή αυτή εφαρμόζεται και στις συναρτήσεις χωρίς τύπο. Αν ας πούμε, έχουμε το πρόβλημα:

> *Γράψε συνάρτηση, με το όνομα pqer, που θα τροφοδοτείται μέσω των ορισμάτων της με τρεις πραγματικές τιμές, ας πούμε x, y, z και θα υπολογίζει και θα επιστρέφει τις τιμές των παραστάσεων:*

$$t = \frac{xy}{x^2 - y^2} z^{x-y}, \quad u = \frac{xy - \dfrac{1}{x}}{z}, \quad \text{αν } |x| \neq |y| \text{ και } z > 0.$$

θα γράψουμε:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
   {
      cerr << " η pqer κλήθηκε με ορίσματα " << x << ", "
           << y << ", " << z << endl;
      exit( EXIT_FAILURE );
   }
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Αυτός ο τρόπος διαχείρισης τέτοιων καταστάσεων –ορίσματα εκτός πεδίου ορισμού– έχει δυσάρεστο αποτέλεσμα. Π.χ. οι παρακάτω εντολές προσπαθούν να δώσουν πίνακα τιμών της *v* από -5 ως 5 ανά 0.5:

```
   for ( int k(-10); k <= 10; ++k )
      cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v κλήθηκε με όρισμα: -4
```

Θα δούμε τώρα έναν άλλον τρόπο, σαφώς πιο ευέλικτο:

```
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

double v( double x );

int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " η v δεν ορίζεται στο " << x << endl;
      }
   } // for (int k...
} // main

double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      throw x;
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
```

```
η v δεν ορίζεται στο -4
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
η v δεν ορίζεται στο -2
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
η v δεν ορίζεται στο 4
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, έχουμε πολύ πιο «ήπια» αντιμετώπιση του σφάλματος. Παίρνουμε τουλάχιστον τις τιμές στις οποίες υπολογίζεται η συνάρτησή μας.

Ποιες διαφορές έχουμε τώρα; Οι «μαγικές» λέξεις είναι: `throw`, `try` και `catch`.

- Όταν καταλαβαίνουμε ότι έχουμε τιμή εκτός πεδίου ορισμού, με τη "`throw x`" **ρίχνουμε** (ή **εγείρουμε**) **μιαν εξαίρεση** (throw (raise) an exception). Η *x* είναι το **αντικείμενο της εξαίρεσης** (exception object). Ύστερα από αυτό διακόπτεται η εκτέλεση της συνάρτησης *v()* και ο έλεγχος επιστρέφει στη συνάρτηση που την κάλεσε, στην περίπτωσή μας στη **main**.
- Στη **main**, η κλήση της *v* γίνεται μέσα σε μια εντολή `try` (δοκίμασε). Στην ομάδα της εντολής `try` βάζουμε τις εντολές –κλήσεις συναρτήσεων– από τις οποίες περιμένουμε ότι μπορεί να ριχθεί κάποια εξαίρεση.
- Η εξαίρεση **συλλαμβάνεται** από κάποια `catch` (σύλλαβε) που ακολουθεί την `try`. Μια `catch` συλλαμβάνει εξαιρέσεις του τύπου που αναφέρεται στην παράμετρό της. Μέσα στην ομάδα της `catch` κάνουμε **διαχείριση της εξαίρεσης** (exception handling): λέμε στον υπολογιστή τι θέλουμε να κάνει όταν τη συλλάβει.

Στο παράδειγμα που δώσαμε δεν τονίζουμε και τόσο την υπεροχή αυτού του τρόπου. Ας δούμε μια άλλη παραλλαγή, κάπως πιο «εποικοδομητική»:

```
int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " v(" << (x+1e-5) << ") = " << v(x+1e-5) << endl;
      }
   } // for (int k...
} // main
```

που μας δίνει:

```
v(-5) = -2.33333
v(-4.5) = -3.06667
v(-3.99999) = -100001
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
v(-1.99999) = -100001
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
v(4.00001) = -100001
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, στη διαχείριση της εξαίρεσης, *ξανακαλούμε τη συνάρτηση* με τιμή που σίγουρα δεν έχει πρόβλημα, αλλά είναι κοντά στην τιμή που έριξε την εξαίρεση. Βέβαια, σε μια πραγματική εφαρμογή, κάτι τέτοιο είναι απίθανο να έχει νόημα.

Βάζοντας διαφορετικά τις **try-catch** παίρνουμε αποτελέσματα παρόμοια με αυτά που παίρναμε με την *exit()*:

```
int main()
{
   try
   {
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
} // main
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v δεν ορίζεται στο -4
```

Πάντως το πρόγραμμα είναι ουσιωδώς διαφορετικό:

- Στην αρχική μορφή *η συνάρτηση παίρνει την απόφαση* να διακόψει την εκτέλεση του προγράμματος καλώντας την *exit()*.
- Στη νέα μορφή η συνάρτηση στέλνει μήνυμα ότι συνάντησε ανυπέρβλητο πρόβλημα. Την απόφαση για το τι θα γίνει παίρνει ο «γενικός διεύθυντής», δηλαδή η **main**.

Θα πρέπει όμως, να διαλύσουμε δυο συνηθισμένες παρεξηγήσεις:

- Θα διαχειριζόμαστε πάντοτε τις εξαιρέσεις στη **main**; Όχι. Στη συνέχεια, θα δούμε πώς αποφασίζουμε πού και πώς μπορεί να γίνει η καλύτερη διαχείριση της κάθε εξαίρεσης.
- Οι εξαιρέσεις ρίχνονται μόνο από *συναρτήσεις* που καλούμε στην ομάδα **try**; Όχι! Αργότερα θα δεις παραδείγματα που θα έχουμε εντολές **throw** *μέσα στην ομάδα* της **try**.

Ας πούμε τώρα ότι στο ίδιο πρόγραμμα χρησιμοποιούμε και την *pqer*, που την αλλάζουμε ως εξής:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
      throw " η pqer κλήθηκε με ορίσματα εκτός πεδίου ορισμού";
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Τώρα, η *pqer* ρίχνει εξαίρεση τύπου **char*** (το βέλος προς το μήνυμα).

Θα πεις: «Ναι, αλλά τώρα, όταν πιάσουμε την εξαίρεση, το πολύ-πολύ να γράψουμε το μήνυμα. Δεν μπορούμε όμως να έχουμε τις τιμές των *x*, *y*, *z* που προκάλεσαν το πρόβλημα και να τις χειριστούμε "πιο δημιουργικά".» Αργότερα θα μάθουμε πώς μπορούμε να περνούμε τέτοιες πληροφορίες.

Για να γράψουμε τη **main** έχουμε το εξής πρόβλημα: η *v* μπορεί να ρίξει αντικείμενο εξαίρεσης τύπου **double** ενώ η *pqer* μπορεί να ρίξει αντικείμενο τύπου **char***. Πώς τα συλλαμβάνουμε; Δες πώς γίνεται η **main**:

```
int main()
{
   double t, u;

   try
   {
```

```
   // . . .
      pqer( 1, 2, 3, t, u );
   // . . .
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
} // main
```

Όπως βλέπεις, μια `try` μπορεί να έχει περισσότερες από μια `catch` για να χειριστεί διάφορες εξαιρέσεις που προέρχονται από μια ή περισσότερες συναρτήσεις που καλούνται μέσα στην ομάδα της.

Αν βάλεις μια `catch( ... )` συλλαμβάνεις όλες τις εξαιρέσεις. Π.χ. στο τελευταίο παράδειγμα μπορείς να βάλεις:

```
   try
   {
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
   catch ( ... )
   {
      cout << " μη αναμενόμενη εξαίρεση" << endl;
   }
```

Τέλος, να πούμε ότι μπορείς, αν θέλεις, να δηλώνεις στην επικεφαλίδα της συνάρτησης τις **προδιαγραφές εξαιρέσεων** (exception specification) –δηλαδή τους τύπους των εξαιρέσεων που μπορεί να ρίξει– ως εξής:

```
double v( double x ) throw( double );
void pqer( double x, double y, double z,
           double& t, double& u ) throw( char* );
double f( double x ) throw( double, int );
```

Αν μια συνάρτηση ρίξει εξαίρεση εκτός προδιαγραφών προκαλεί διακοπή της εκτέλεσης του προγράμματος. Προσοχή όμως: η

```
double g( double x ) throw()
```

σημαίνει ότι *από τη g() δεν θα ριχτεί ούτε θα περάσει οποιαδήποτε εξαίρεση* και είναι διαφορετική από την

```
double g( double x )
```

που σημαίνει ότι *από την g() περιμένεις οποιαδήποτε εξαίρεση.*

14.9.1 Μια Ιστορία με Εξαιρέσεις

Τώρα, θα δούμε ένα πιο πολύπλοκο παράδειγμα για να σου δώσουμε μια καλύτερη αίσθηση της λειτουργίας του μηχανισμού των εξαιρέσεων. Ξαναγράφουμε το πρόγραμμα του παραδ. 1 της §7.7 ως εξής:

```
#include <iostream>

using namespace std;

unsigned long int comb( int m, int n );

int main()
{
   int m, n;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   try
   {
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb(m,n) << endl;
   }
   catch( int )
   {
      cout << " Λάθος δεδομένα" << endl;
   }
} // main

// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
unsigned long int natProduct( int m, int n )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// 0 < m <= n
   unsigned long int fv( m );
   for ( int k(m+1); k <= n; ++k )  fv *= k;
   return fv;
} // natProduct

// factorial -- Υπολογίζει το a!
unsigned long int factorial( int a )
{
   return (a == 0) ? 1 : natProduct(1, a);
} // factorial

// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
unsigned long int comb( int m, int n )
{
   unsigned long int fv;

   if ( n < m-n )  fv = natProduct(m-n+1, m)/factorial(n);
             else  fv = natProduct(n+1, m)/factorial(m-n);
   return fv;
} // comb
```

Κατ' αρχάς, να εξηγήσουμε τι κάνουμε: Ας πούμε ότι έχουμε να υπολογίσουμε τους $\binom{5}{2}$. Στο αρχικό πρόγραμμα ζητούσαμε τον υπολογισμό:

$$\binom{5}{2} = \frac{5!}{2!(5-2)!} = \frac{5!}{2!3!} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5}{2!\cdot 1\cdot 2\cdot 3}$$

Φυσικά, μπορούμε να απλοποιήσουμε το 1·2·3 και να αποφύγουμε άσκοπες πράξεις. Η *comb()* κάνει ακριβώς αυτό, επιλέγοντας τη μέγιστη απλοποίηση. Για τον σκοπό αυτόν

όμως χρειαζόμαστε τη `natProduct( int m, int n )` που υπολογίζει το γινόμενο φυσικών αριθμών: $m\cdot(m+1)\cdot\ldots\cdot(n-1)\cdot n$, με την προϋπόθεση $0 < m \le n$. Αν η *natProduct* κληθεί χωρίς να ισχύει η προϋπόθεση ρίχνει εξαίρεση.

Έχοντας τη *natProduct()* μπορούμε να γράψουμε, όπως βλέπεις, πολύ πιο απλά τη *factorial()*.

Ο μόνος έλεγχος εγκυρότητας των δεδομένων γίνεται στη *natProduct()*. Αυτό έχει ως αποτέλεσμα να είναι το πρόγραμμα καθαρογραμμένο!

Αν η *natProduct()* κληθεί με ορίσματα που δεν ικανοποιούν την $0 < m \le n$ ρίχνει την εξαίρεση (τύπου `int`) "`-1`". Ας πούμε λοιπόν ότι, από λάθος, ζητείται στη **main** ο υπολογισμός των συνδυασμών $\binom{8}{10}$.

- Η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση "`comb(m, n)`" της *comb()*. Επειδή η $n < m-n$ δεν ισχύει, η εκτέλεση συνεχίζεται στην περιοχή του `else`. Η *comb()* για να κάνει τον υπολογισμό θα ζητήσει τους υπολογισμούς "`natProduct(11, 8)`" και "`factorial(-2)`". Ας πούμε τώρα ότι ζητείται πρώτα ο υπολογισμός "`factorial(-2)`".
- Και πάλι η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *factorial()*. Η *factorial()*, με τη σειρά της, κάνει την κλήση `natProduct(1, -2)`.
- Και αυτήν τη φορά η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *natProduct()*. Η *natProduct()* βρίσκει ότι έχουμε $m \le 0$, δεν κάνει υπολογισμούς ρίχνει την εξαίρεση "`-1`".
- Τα σχετικά με την κλήση της *natProduct()* φεύγουν από τη στοίβα και επιστρέφουμε στη *factorial()*.
- Η *factorial()* δεν συλλαμβάνει την εξαίρεση. Τα σχετικά με την κλήση της *factorial()* φεύγουν από τη στοίβα και επιστρέφουμε στην *comb()*.
- Αλλά ούτε η *comb()* έχει κάποια `catch(int ...)` για να συλλάβει την εξαίρεση· τα σχετικά με την κλήση της *comb()* φεύγουν από τη στοίβα και επιστρέφουμε στη **main** όπου η εξαίρεση συλλαμβάνεται.

Το πέρασμα του ελέγχου από κάθε συνάρτηση προς αυτήν που την κάλεσε χωρίς να γίνεται οποιαδήποτε άλλη δουλειά λέγεται **ξετύλιγμα της στοίβας** (stack unwinding). Αν η εξαίρεση δεν συλληφθεί ούτε στη **main** θα προκαλέσει τελικώς διακοπή της εκτέλεσης του προγράμματος.

Αργότερα θα δούμε τη διαχείριση των εξαιρέσεων πιο εκτεταμένα, αλλά προς το παρόν θα τονίσουμε ότι:

1. Ο μηχανισμός των εξαιρέσεων, που υλοποιείται με τις εντολές `throw`, `try` και `catch`, επιτρέπει στον προγραμματιστή να σχεδιάσει σωστά το πρόγραμμά του ώστε να διαχειρίζεται προβληματικές καταστάσεις.
2. Στις συναρτήσεις χωρίς τύπο, μπορείς, αν δεν θέλεις να χρησιμοποιήσεις εξαιρέσεις, να βάζεις μια παράμετρο από την οποίαν θα επιστρέφεις κάποια πληροφορία σχετικώς με το σφάλμα. Ξαναδές το παράδειγμα της §13.11. Πάντως το ξετύλιγμα της στοίβας με τις εξαιρέσεις είναι πιο απλό.
3. Σε συνέχεια του παραπάνω: Δες πόσο απλά και *καθαρά* γράφηκαν οι *factorial()* και *comb()*!

14.10 Αναδρομή (ξανά)

Όπως λέγαμε και στην §7.10, στη C++ υπάρχει η δυνατότητα αναδρομικής διατύπωσης μιας συνάρτησης, όπως και στα μαθηματικά υπάρχει η δυνατότητα αναδρομικής διατύπωσης μερικών ορισμών. Είδαμε ακόμη δύο παραδείγματα: μια συνάρτηση που υπολογίζει το $n!$ και μια που υπολογίζει τον μέγιστο κοινό διαιρέτη δύο ακεραίων και είχαμε παρατηρήσει

ότι ο αναδρομικός τρόπος, συγκρινόμενος με τον ισοδύναμο επαναληπτικό τρόπο, είναι πιό απλός, πιό σύντομος και πλησιέστερος προς τον μαθηματικό ορισμό.

Συχνά όμως, όπως θα δούμε παρακάτω, ο αναδρομικός τρόπος είναι λιγότερο αποδοτικός από τον επαναληπτικό και σε χρόνο και σε μνήμη.

Τί πληρώνουμε για αυτήν την καλύτερη γραφή; Για να γίνει ο υπολογισμός με την αναδρομική μορφή θα χρειαστούμε n+1 ενεργοποιήσεις της συνάρτησης, που κάθε μια παίρνει μνήμη από τη στοίβα. Χρησιμοποιούμε δηλαδή μνήμη ανάλογη του n και φυσικά αντίστοιχο χρόνο.

Ας δούμε δύο ακόμη παραδείγματα. Το πρώτο είναι κλασικό και πολυσυζητημένο. Ο (Dijkstra 1976), στη μονογραφία του "A Discipline of Programming", άφηνε έξω από το δομημένο προγραμματισμό την αναδρομή. Οι θιασώτες της φυσικά δεν το δέχτηκαν. Οι (Manna & Waldinger 1978), που δούλευαν ερευνητικώς στην αυτόματη σύνθεση προγράμματος, θεωρούσαν την αναδρομή απαραίτητη τουλάχιστον στην αρχική σχεδίαση· έγραψαν λοιπόν μια εργασία που είναι −οξύτερη από ό,τι συνηθίζεται στις επιστημονικές εργασίες− κριτική στις θέσεις του Dijkstra. Στην εργασία τους σχολιάζουν δυο (το 6ο και το 2ο) από τα «οκτώ μικρά παραδείγματα» του Dijkstra. Στη συνέχεια θα δούμε το πρώτο από αυτά τα παραδείγματα.

Παράδειγμα

Μας δίνονται δυο ακέραιοι, $x > 1$ και $y \geq 0$ (προϋπόθεση), και θέλουμε να βρούμε κάποιο z τέτοιο ώστε να ισχύει η (απαίτηση):

R: $z == x^y$

Ο Dijkstra προτείνει να χρησιμοποιήσουμε μια βοηθητική μεταβλητή h, τέτοια ώστε να ισχύει η

P: $h \cdot z == x^y$

και να γράψουμε μια

while ($h \neq 1$) «συμπίεσε» το h κρατώντας αναλλοίωτη την P

Για να ισχύει αρχικώς η αναλλοίωτη βάζουμε: "`h = x^y; z = 1`". Είναι φανερό ότι, όταν τελειώσει η εκτέλεση της **while**, θα έχουμε $h == 1$ που μαζί με την αναλλοίωτη μας δίνουν την R.

Η τιμή x^y δεν μας είναι γνωστή, ώστε να την εκχωρήσουμε αρχικά στην h. Το πρόβλημα αυτό μπορεί να λυθεί αν γράψουμε την h ως xx^{yy} οπότε η αναλλοίωτή μας γράφεται

P: $xx^{yy} \cdot z == x^y$

ενώ το πρόγραμμά μας γίνεται:

```
xx = x;  yy = y;  z = 1;
while (yy != 0)
   «συμπίεσε» το yy κρατώντας αναλλοίωτη την P
```

Η «συμπίεση» μπορεί να γίνεται με την εντολή "`yy = yy - 1`", οπότε η P παραμένει αναλλοίωτη αν συγχρόνως αλλάζουμε και την τιμή της `z = z*xx`. Είναι φανερό ότι η εκτέλεση της **while** κάποτε τερματίζεται. Να λοιπόν η συνάρτηση υπολογισμού της δύναμης:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      --yy;
      z *= xx;
   } // while
   return z;
```

```
} // power
```

Ύστερα από αυτό, ο Dijkstra ψάχνει για πιο γρήγορο αλγόριθμο. Παρατηρεί ότι αν ο *yy* είναι άρτιος και βάλουμε **xx = xx²** και **yy = yy/2** η τιμή της *h* δεν αλλάζει:

$$h == xx^{yy} == (xx^2)^{yy/2}$$

Έτσι, καταλήγει στην[20]

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      while ( even(yy) )
      {
         xx *= xx;
         yy /= 2;
      } // while (even(yy))
      --yy;
      z *= xx;
   } // while
   return z;
} // power
```

Ο τερματισμός της εσωτερικής **while** είναι σίγουρος, διότι ο μόνος άρτιος που μπορεί να διαιρείται επ' άπειρον δια 2, χωρίς να δώσει περιττό, είναι ο 0 (μηδέν), αλλά από αυτόν μας προστατεύει η εξωτερική **while**.

Ενώ η πρώτη λύση έχει χρόνο εκτέλεσης ανάλογο του *y*, η δεύτερη έχει χρόνο εκτέλεσης ανάλογο του $log_2 y$.

Οι Manna και Waldinger αντιτείνουν ότι η αναλλοίωτη, η συνθήκη συνέχισης και –τελικώς– το πρόγραμμα δεν βγήκαν, αλλά γράφτηκαν μια και ήταν γνωστά εκ των προτέρων. Και δίνουν, ως το πιο απλό πρόγραμμα που θα μπορούσε να γραφτεί για την περίπτωση, το:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
   if ( y == 0 )  fv = 1;
            else  fv = x * power( x, y-1 );
   return fv;
} // power
```

Τί πιο απλό! Δεν γράφουμε παρά τις βασικές ιδιότητες: $x^0 == 1$ και $x^y == x \cdot x^{y-1}$.

Η βελτίωση του αλγορίθμου, η αντίστοιχη αυτής που κάνει ο Dijkstra, είναι επίσης απλούστατη:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
```

[20] Η *even* παίρνει ένα όρισμα τύπου **int** και μας δίνει **true** αν το όρισμα είναι άρτιος (even), **false** αν το όρισμα είναι περιττός. Θα μπορούσες να τη γράψεις ως εξής:

```
bool even( int x )  {  return ( x % 2 == 0 );  }
```

```
   if ( y == 0 )
      fv = 1;
   else if ( even(y) )
   {  unsigned int z( power(x,y/2) );
      fv = z*z;  }
   else
      fv = x * power( x, y-1 );
   return fv;
} // power
```

και αν είχες πρόβλημα να καταλάβεις το βελτιωμένο πρόγραμμα του Dijkstra, δεν πρέπει να έχεις δυσκολία να καταλάβεις αυτό εδώ!

Δεν μπορούμε παρά να θαυμάσουμε την ομορφιά και την απλότητα του αναδρομικού προγράμματος. Αλλά, δεν είναι δωρεάν! Τι πληρώνουμε; Για σκέψου πώς θα εκτελεσθεί αυτή η συνάρτηση; Οι διαδοχικές αναδρομικές κλήσεις εισάγουν στη στοίβα τις τιμές y, y-1,... 1, 0. Στη συνέχεια αφαιρεί όλες αυτές τις τιμές, υπολογίζοντας το γινόμενο $1 \cdot x \cdot x \cdot \ldots \cdot x$. Στη βελτιωμένη περίπτωση οι κλήσεις είναι, στην καλύτερη περίπτωση, $log_2 y$. Μικρό το κακό (φυσικά για μικρά y);

Πάντως, υπάρχουν και περιπτώσεις όπου η απλότητα του αναδρομικού προγράμματος δρα σαν «σειρήνα» που μας τραβάει στον «κακό δρόμο». Τυπικό παράδειγμα οι αριθμοί Fibonacci, που ως γνωστόν ορίζονται ως εξής:[21]

$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$ για $n \geq 2$

Μεταφράζουμε λοιπόν αμέσως:

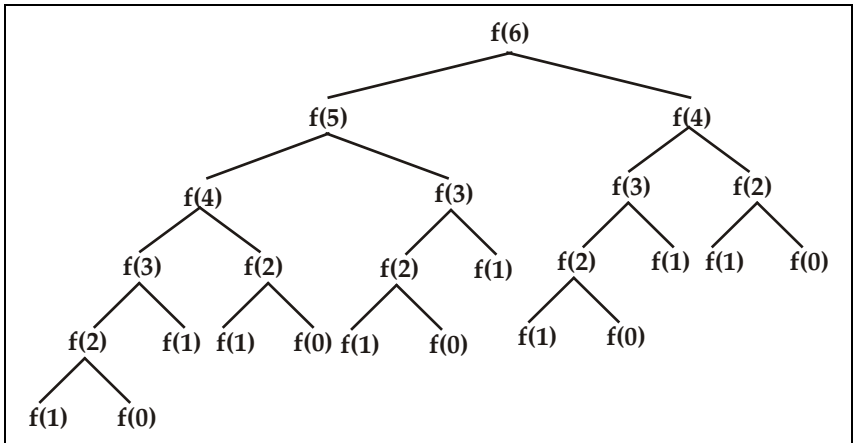
```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fv;
   if ( n < 2 )  fv = n;
           else  fv = f(n-1) + f(n-2);
   return fv;
} // f
```

και στο Σχ. 14-1 βλέπεις τι γίνεται για να υπολογιστεί ο $f(6)$. Ο $f(4)$ υπολογίζεται 2 φορές, το $f(3)$ 3, το $f(2)$ 5 και το $f(1)$ 8.

Στην περίπτωση αυτή συμφέρει να γράψουμε μια επαναληπτική συνάρτηση, φροντίζοντας να φυλάγουμε *δυο* προηγούμενους αριθμούς σε κάθε βήμα:

```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fn, fp, fpp;
   if ( n < 2 )
   {
      fn = n;
   }
   else
   {
      fp = 0;  fn = 1;
      for ( int j(1); j < n; ++j )
      {
         fpp = fp;  fp = fn;
         fn = fp + fpp;
      } // for
```

[21] Έλυσες την Άσκηση 7-14;



Σχ. 14-1 Αναδρομικές κλήσεις για τον υπολογισμό f(6).

```
   } // if (n < 2)
   return fn;
} // f
```

Για τον υπολογισμό της δύναμης είπαμε «μικρό το κακό»· εδώ δεν μπορούμε να πούμε το ίδιο. Θα γράφουμε αναδρονικές συναρτήσεις, αλλά καλό θα είναι να σκεφτόμαστε που οδηγούν (και πόσο καλές είναι άλλες εναλλακτικές λύσεις).

Ας ξαναγυρίσουμε στο παράδειγμα της *power()*. Μερικοί μπορεί να αναρωτιούνται «Είναι δυνατόν, στο επαναληπτικό πρόγραμμα να σπαζοκεφαλιάζω για αναλλοίωτες και για τερματισμούς, ενώ όταν γράφω το αναδρομικό να μην υπάρχουν τέτοια προβλήματα;» Όχι βέβαια! Δεν συμβαίνει κάτι τέτοιο. Εκείνο που συμβαίνει είναι ότι αυτά τα πράγματα είναι, συχνά, πολύ απλά για το αναδρομικό πρόγραμμα. Η αναλλοίωτη εμφανίζεται τώρα σαν συμμόρφωση προς τις προδιαγραφές της συνάρτησης· δηλαδή, η κλήση μιας συνάρτησης είτε απ' έξω είτε από μέσα από το σώμα του υποπρογράμματος, με αναδρομή, θα πρέπει να γίνεται με τους ίδιους όρους. Για να έχουμε τερματισμό, σε μια αναδρομική συνάρτηση *θα πρέπει να υπάρχει μια τουλάχιστον περίπτωση εκτέλεσης χωρίς αναδρομική κλήση*. Ακόμη, *οι αναδρομικές κλήσεις θα πρέπει να οδηγούν σε μια τέτοια εκτέλεση*.

Στην *power()* οι προδιαγραφές είναι:

το όρισμα που αντιστοιχεί στη x πρέπει να είναι θετικός ακέραιος,
το όρισμα που αντιστοιχεί στη y, πρέπει να είναι μη αρνητικός ακέραιος,
το αποτέλεσμα είναι ίσο με x^y.

Οι δυο προϋποθέσεις ελέγχονται στην αρχή της συνάρτησης. Φυσικά, δεν φτάνει αυτό: Κάθε αναδρομική κλήση μειώνει την τιμή της y κατά 1 εκτός από την περίπτωση που έχουμε y == 1. Η παραπάνω παρατήρηση μας βεβαιώνει και για τον τερματισμό. Για y == 1 δεν έχουμε αναδρομική κλήση και είναι σίγουρο ότι θα φτάσουμε σε μια κλήση της συνάρτησης με y == 1.

14.11 * Ακαθόριστο Πλήθος Παραμέτρων

Είδαμε πιο πριν ότι η κλήση μιας συνάρτησης μπορεί να έχει λιγότερες πραγματικές από τις τυπικές παραμέτρους. Η C++ σου επιτρέπει να γράφεις συναρτήσεις με μη προκαθορισμένο πλήθος παραμέτρων.

Ας πούμε ότι θέλουμε μια συνάρτηση που θα τροφοδοτείται με μια τιμή τύπου `char` και μη προκαθορισμένο πλήθος τιμών τύπου `double` και αν η τιμή τύπου `char` είναι `'>'` θα μας επιστρέφει ως τιμή τη μέγιστη από τις τιμές τύπου `double` ενώ αν είναι `'<'` θα μας επιστρέφει την ελάχιστη.

Σύμφωνα με όσα έχουμε μάθει, η συνάρτησή μας, ας την πούμε *maxormin()*, θα είναι συνάρτηση με τύπο: θα επιστρέφει τιμή τύπου **double**. Θα έχει μια παράμετρο τιμής, τύπου **char**, ας την πούμε *tel*, δηλαδή:

```
double maxormin(char tel, ...)
```

και μετά; Ε λοιπόν: η C++ δέχεται αυτήν την επικεφαλίδα·[22] τα **αποσιωπητικά** (ellipsis) δείχνουν ότι, όταν κληθεί, μπορεί να πάρει πολλά ορίσματα.

Δες πώς γράφεται η συνάρτηση και ας τη συζητήσουμε στη συνέχεια:

```
#include <iostream>
#include <cstdarg>

using namespace std;

double maxormin( char tel, ... )
{
   if ( tel != '<' && tel != '>' )
   {
      throw tel;
   }
   va_list ap;
   va_start( ap, tel );

   double x( va_arg(ap, double) );
   if ( x == 0 )
   {
      throw 0;
   }
   double fv( x );
   if ( tel == '>' )
   {
      while ( x != 0 )
      {
         if ( x > fv )  fv = x;
         x = va_arg( ap, double );
      }
   }
   else // tel == '<'
   {
      while ( x != 0 )
      {
         if ( x < fv )  fv = x;
         x = va_arg( ap, double );
      } // while (x != 0)
   } // if (tel == '>')
   va_end( ap );
   return fv;
} // maxormin

int main()
{
   double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
   double mn = maxormin( '<', 1.1, 2.2, 1.3, 0.4, 7.1, 0.0 );
   cout << mx << "  " << mn << endl;
} // main
```

1. Περιλάβαμε στο πρόγραμμά μας το cstdarg. Στο αρχείο αυτό ορίζεται ο τύπος *va_list* και δηλώνονται οι συναρτήσεις *va_arg()*, *va_start()* και *va_end()*.

2. Μέσα στη συνάρτησή μας δηλώσαμε τη μεταβλητή *ap* τύπου *va_list*. Πρόκειται για έναν πίνακα.

[22] Κληρονομιά από τη C!

3. Με την κλήση "`va_start(ap, tel)`" τροφοδοτούμε τη *va_start()* με τη μοναδική σταθερή παράμετρο (*tel*) και αυτή μας επιστρέφει πληροφορίες για τις υπόλοιπες πραγματικές παραμέτρους στην *ap*.

4. Καλούμε ξανά και ξανά τη "`va_arg(ap, double)`". Κάθε φορά μας επιστρέφει ως τιμή *την τιμή του επόμενου ορίσματος* και αλλάζει την τιμή της *ap* ώστε στη συνέχεια να προχωρήσουμε παρακάτω (στο μεθεπόμενο). Πρόσεξε ότι δεύτερο όρισμα στην κλήση της *va_arg* είναι ένας *τύπος* (!). Είναι *ο τύπος του ορίσματος που περιμένουμε να διαβάσουμε*.

5. Στο τέλος βάζουμε την κλήση της `va_end(ap)`. Είναι απαραίτητη για την ολοκλήρωση της κλήσης (αυτή θα κάνει ανάταξη της στοίβας).

Ας πούμε ότι η συνάρτησή μας καλείται να βρει τον μέγιστο. Ξεχνούμε για λίγο τις δύο **if** και παρακολουθούμε την εκτέλεση:

```
x = va_arg( ap, double );  // πάρε το πρώτο όρισμα στη x...
fv = x;                    // ... και θεώρησε ότι είναι μέγιστο
x = va_arg( ap, double );  // πάρε το επόμενο όρισμα στη x...
while ( x != 0 )           // όσο δεν είναι 0 (μηδέν)
{
   if ( x > fv )  fv = x;   //  αν x > fv διόρθωσε το μέγιστο
   x = va_arg( ap, double );//  πάρε το επόμενο όρισμα στη x...
} // while
```

Ο τρόπος που υπολογίζουμε το μέγιστο είναι ο γνωστός αλλά είναι φανερό ότι η **while** ελέγχεται με φρουρό την τιμή 0 (μηδέν) στη *x*! Ακριβώς! Όπως μπορείς να δεις στην κλήση της συνάρτησης έχουμε:

```
double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
```

Το "`0.0`" στο τέλος είναι φρουρός! Δεν θα μπορούσαμε να βάλουμε μετρούμενη επανάληψη; Βέβαια, αρκεί να βάλουμε άλλη μια σταθερή παράμετρο στην αρχή (1η ή 2η) που να περνάει στη συνάρτηση το πλήθος των ορισμάτων που ακολουθούν.

Αν θελήσεις να γράψεις τέτοια συνάρτηση (συνήθως δεν είναι η καλύτερη λύση) πρόσεξε τα εξής:

1. Ο φρουρός θα πρέπει να είναι του ίδιου τύπου με τα άλλα ορίσματα και να επιλέγεται όπως ξέρουμε (το "`0.0`" στο παράδειγμά μας δεν είναι πολύ καλή επιλογή). Αν προτιμήσεις να περάσεις το πλήθος πρόσεξε να μετρήσεις σωστά!

2. Η συνάρτησή σου θα πρέπει να έχει μια τουλάχιστον σταθερή παράμετρο. Είναι προφανές ότι οι σταθερές παράμετροι μπαίνουν στην αρχή.

3. Αν έχεις περισσότερες από μία σταθερές παραμέτρους, στην κλήση της *va_start() θα βάλεις ως δεύτερη παράμετρο την τελευταία σταθερή παράμετρο* που πρέπει να μην είναι παράμετρος αναφοράς.

4. Οι μη σταθερές παράμετροι θα πρέπει να είναι του ίδιου τύπου ή να εμφανίζουν κάποιο σταθερό σχήμα επανάληψης (π.χ. μια τύπου `char` μια τύπου `int`). Μόνον έτσι μπορείς να χρησιμοποιήσεις επαναληπτική εντολή (και έχει νόημα αυτό το είδος της συνάρτησης).

5. Οι πραγματικές παράμετροι θα πρέπει να συμφωνούν πλήρως με τους τύπους που βάζεις στην κλήση της *va_arg()*. Αν, στο παράδειγμά μας βάζαμε ως 3η παράμετρο όχι 2.0 αλλά 2 θα είχαμε πρόβλημα.

14.12 Συνοψίζοντας...

Κυρίως σε μαθηματικές εφαρμογές (αλλά όχι μόνον), παρουσιάζεται η ανάγκη γράψουμε συναρτήσεις με παράμετρο συνάρτηση. Η C++ λύνει αυτό το πρόβλημα με παράμετρο-βέλος-προς-συνάρτηση.

Πολύ συχνά παρουσιάζεται η ανάγκη να αλλάξουμε τύπους κάποιων μεταβλητών. Είναι πολύ σημαντικό να μην χρειάζεται να αλλάξουμε και τα ονόματα συναρτήσεων ή να

αλλάξουμε πράξεις με τελεστές σε κλήσεις συναρτήσεων. Η *επιφόρτωση* συναρτήσεων και τελεστών είναι η τεχνική με την οποία λύνουμε αυτά τα προβλήματα.

Αν οι συναρτήσεις που επιφορτώνονται διαφέρουν μόνο σε τύπους δεδομένων και δώσουμε στον μεταγλωττιστή ένα σχέδιο, το *περίγραμμα* (template), αυτός θα τις γράψει μόνος του.

Ας πούμε τώρα ότι για να λύσεις ένα συγκεκριμένο πρόβλημα γράφεις κάποια ή κάποιες συναρτήσεις που καταλαβαίνεις ότι θα σου είναι χρήσιμη/ες γενικότερα. Τα περιγράμματα συναρτήσεων και οι συναρτησιακές παράμετροι είναι δύο εργαλεία που σου επιτρέπουν να δώσεις την κατάλληλη παραμετρική μορφή ώστε να τα χρησιμοποιείς με ευκολία και στο μέλλον.

Πιθανότατα μεγαλύτερης σπουδαιότητας είναι ένα άλλο εργαλείο που είδαμε στο κεφάλαιο αυτό: οι *εξαιρέσεις*. Μαθαίνοντας να τις χρησιμοποιείς θα δεις ότι θα αλλάξει συνολικώς ο τρόπος που γράφεις τα προγράμματά σου: θα γίνονται πιο ευέλικτα, πιο λειτουργικά και –καθώς θα μαθαίνεις πώς να χειρίζεσαι τις εξαιρέσεις– πιο καλοσχεδιασμένα και με λιγότερα λάθη. Οι δομές (κλάσεις) εξαιρέσεων θα είναι ένα πολύ σοβαρό εργαλείο για την ανάπτυξη αλλά και τη συντήρηση του λογισμικού που γράφεις.

Τρία θέματα που ασχοληθήκαμε ακόμη ήταν

- Οι *συναρτήσεις* **inline**: με αυτές μπορείς να επιταχύνεις κάπως την εκτέλεση του προγράμματός σου.
- Η *αναδρομή*: ένα πολύ καλό εργαλείο που έχει όμως και μερικές παγίδες. Τώρα μπορείς να το χρησιμοποιείς «μετά λόγου γνώσεως».
- Το *ακαθόριστο πλήθος παραμέτρων*: Βάλαμε αυτήν την παράγραφο μόνο για να καταλαβαίνεις συναρτήσεις που έχουν γραφεί έτσι. *Μη χρησιμοποιείς αυτήν τη δυνατότητα.* Αντιθέτως, *η δυνατότητα να παραλείπεις παραμέτρους που έχουν ερήμην καθορισμένη τιμή είναι μια χαρά.*

Ασκήσεις

Α Ομάδα

14-1 (Γενίκευση της Ασκ.9-5) Γράψε μια

```
double vectorSumIf( const double x[], int n, bool (*predic)(double) )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία ισχύει η *predic*(x[*k*]).

Β Ομάδα

14-2 Θέλουμε συνάρτηση με όνομα *pluMin()* που

α) όταν καλείται με δύο ορίσματα *a*1, *a*2, τύπου `int`, θα αυξάνει την τιμή του πρώτου ορίσματος, *a*1, κατά 1 και θα μειώνει την τιμή του *a*2 κατά 1.

β) όταν καλείται με τέσσερα ορίσματα *a*1, *a*2, *b*1, *b*2 τύπου `int`, θα προσθέτει στην τιμή του *a*1 την τιμή του *b*1 και θα αφαιρεί από την τιμή του *a*2 την τιμή του *b*2.

Τι συνάρτηση θα γράψουμε, με τύπο ή χωρίς τύπο; Δικαιολόγησε την απάντησή σου.

Για κάθε μια παράμετρο δικαιολόγησε όλα τα χαρακτηριστικά της (τιμής ή αναφοράς, τύπο κλπ).

Γράψε τη συνάρτηση και δώσε παράδειγμα χρήσης με 2 και 4 ορίσματα. Στην κάθε περίπτωση θα βάζεις μέσα σε σχόλια τις τιμές των μεταβλητών (που χρησιμοποιούνται) πριν και μετά την κλήση.

Η συνάρτηση που έγραψες μπορεί να κληθεί με 3 ορίσματα; Δικαιολόγησε την απάντησή σου· αν είναι θετική δώσε παράδειγμα όπως παραπάνω.

14-3 (Να την απαντήσεις εσύ και *όχι ο μεταγλωττιστής C++ που χρησιμοποιείς*.) Έστω ότι έχουμε:[23]

```
template<typename T1, typename T2>
void f( T1, T2 );                       // 1
template<typename T> void f( T );       // 2
template<typename T> void f( T, T );    // 3
template<typename T> void f( T* );      // 4
template<typename T> void f( T*, T );   // 5
template<typename T> void f( T, T* );   // 6
template<typename T> void f( int, T* ); // 7
template<> void f<int>( int );          // 8
void f( int, double );                  // 9
void f( int );                          // 10
```

Αν

```
int             i;
double          d;
float           ff;
```

ποια από τις παραπάνω θα κληθεί για την κάθε μια από τις:

```
f( i );         // a
f<int>( i );    // b
f( i, i );      // c
f( i, ff );     // d
f( i, d );      // e
f( i, &d );     // f
f( &d, d );     // g
f( &d );        // h
f( d, &i );     // i
f( &i, &i );    // j
```

Γ Ομάδα

14-4 Όπως είναι γνωστό, το ορισμένο ολοκλήρωμα: $\int_a^b f(x)dx$ υπολογίζει το εμβαδό που περικλείεται ανάμεσα στον άξονα *x'x* και στην καμπύλη *y* = *f* (*x*), από *x* = *a* μέχρι *b*. Μια αριθμητική μέθοδος για να προσεγγίσουμε την τιμή του ολοκληρώματος είναι αυτή του **μέσου** (midpoint):

Διαιρούμε το διάστημα [*a*,*b*] σε *N* ίσα τμήματα που το καθένα έχει μήκος $h = (b - a)/N$. Κατασκευάζουμε *N* παραλληλόγραμμα, που το καθένα έχει βάση *h* και ύψος $f(x_k)$, όπου $x_k = a + (k-0.5)h$, είναι το μέσο του *k* διαστήματος. Το άθροισμα των εμβαδών των παραλληλογράμμων μας δίνει μια προσέγγιση του ολοκληρώματος.

$$\int_a^b f(x)dx \approx h\sum_{k=1}^{N} f(x_k)$$

Γράψε μια συνάρτηση *midPoint* που θα τροφοδοτείται με τη συνάρτηση που έχουμε να ολοκληρώσουμε, τα άκρα του διαστήματος της ολοκλήρωσης και το *N* και θα υπολογίζει και θα επιστρέφει την προσέγγιση της τιμής του ολοκληρώματος με τη μέθοδο του μέσου. Τέλος θα έχει μία ακόμη παράμετρο, τη *errCode*· αν η *midPoint* κληθεί με *N* < 1, δεν θα γίνονται υπολογισμοί και η *errCode* θα επιστρέφει τιμή 1, αλλιώς, αν όλα πάνε καλά, θα επιστρέφει τιμή 0.

14-5 Στην §14.3, λέγαμε: Αν m_0 είναι το μέσο του διαστήματος [a_0,b_0], ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
```

[23] Από το (Sutter 1998).

```
    αλλιώς
       ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

και γράψαμε σε C++:

```
    a = a0;  b = b0;
    m = (a + b) / 2;
    if (f(a)*f(m) <= 0.0)  b = m;
                   else  a = m;
```

Αλλά, το «ψάχνουμε για τη λύση στο διάστημα [a_0,m_0]» μπορεί να μεταφρασθεί και ως εξής: «κάλεσε τον εαυτό σου για να ψάξει λύση στο διάστημα [a_0, m_0]». Παρομοίως μπορεί να μεταφρασθεί και το «ψάχνουμε για τη λύση στο διάστημα [m_0,b_0]». Με βάση αυτές τις παρατηρήσεις γράψε μια *αναδρομική* μορφή της *bisection*.

14-6 Με βάση το πρόγραμμα για τη συγχώνευση ταξινομημένων πινάκων (§9.4) γράψε περίγραμμα συνάρτησης:

```
template<typename T> void merge( T v[], T w[], T z[],
                                 int pv, int tv, int pw, int tw,
                                 int pz, int tz,
                                 bool& ok )
```

που θα συγχωνεύει τα τμήματα v[*pv*]...v[*tv*] και w[*pw*]...w[*tw*] των πινάκων v, w στο z[*pz*]...z[*tz*]. Προφανώς θα πρέπει να έχουμε $0 \le pv \le tv$, $0 \le pw \le tw$ και $tz - pz + 1 \ge (tv - pv + 1) + (tw - pw + 1)$. Αν δεν ισχύουν οι παραπάνω, η *ok* επιστρέφει `false`. Τα τμήματα v[*pv*]...v[*tv*] και w[*pw*] ... w[*tw*] πρέπει να είναι ταξινομημένα κατ' αύξουσα τάξη. Την ίδια ταξινόμηση θα έχει και το z[*pz*]... z[*tz*].

14-7 Στο τέλος της §9.6 δίναμε μια άλλη ιδέα για ταξινόμηση:

- Κόβουμε τον πίνακά μας στη μέση και έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε ένα από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικά: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

  Με τις διαδικασίες που γράψαμε, αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα απ' αυτό, δεν μπορούμε να μη σκεφτούμε: «*γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;*» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log(N)$, που φυσικά είναι καλύτερος από N^2.

Γράψε και δοκίμασε τη *mergeSort()* για πίνακες με στοιχεία τύπου *string*. Μετάτρεψε τη συνάρτηση που έγραψες σε περίγραμμα.

κεφάλαιο

15

Δομές – Αρχεία ΙΙ

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις

- να ομαδοποιείς δεδομένα που αναφέρονται σε μια φυσική οντότητα σε ένα *αντικείμενο*, με τον «παλιό» τρόπο, της C, κατά βάση[1],
- να τα φυλάγεις σε *μη-μορφοποιημένα αρχεία*,
- να διαχειρίζεσαι ένα (μη-μορφοποιημένο) αρχείο με *τυχαία πρόσβαση* στις συνιστώσες του,
- να χρησιμοποιείς *δομές εξαιρέσεων*.

Προσδοκώμενα αποτελέσματα:

- Θα κάνεις την πρώτη γνωριμία με την αντικειμενοστρέφεια.
- Θα μάθεις να χρησιμοποιείς τύπους (δομές) εξαιρέσεων που έχουν νόημα.
- Θα μάθεις να χρησιμοποιείς μη-μορφοποιημένα αρχεία και να κρίνεις πότε να τα χρησιμοποιήσεις.
- Θα μάθεις να χρησιμοποιείς αρχεία τυχαίας πρόσβασης.

Έννοιες κλειδιά:

- *δομή - αντικείμενο*
- *μέλος δομής - μέλος αντικειμένου*
- *δημιουργός*
- *ερμηνευτική τυποθεώρηση* - `reinterpret_cast`
- *μή μορφοποιημένο αρχείο*
- *αρχείο τυχαίας πρόσβασης*
- *μέθοδοι seek - μέθοδοι tell*

Περιεχόμενα:

[1] Ο τρόπος της C++; Στο *Μέρος Γ*.

Εισαγωγικές Παρατηρήσεις:

E Pluribus Unum

Μια από τις εμβληματικές φράσεις των ΗΠΑ είναι η «*E Pluribus Unum*». Σημαίνει «από πολλά ένα» και εννοεί ότι πολλές πολιτείες αποτελούν ένα κράτος.

Και τι σχέση έχει αυτό με αυτά που μας ενδιαφέρουν; Στο κεφάλαιο αυτό θα ασχοληθούμε με το πώς μπορούμε να χειριζόμαστε ως ένα αντικείμενο πολλά στοιχεία που αναφέρονται σε μια οντότητα. Ένας πίνακας έχει –κατ' αρχήν– τέτοια χαρακτηριστικά αλλά με έναν βασικό περιορισμό: *όλα τα στοιχεία του πίνακα είναι του ίδιου τύπου*. Αν θέλεις να έχεις ένα αντικείμενο που να περιγράφει έναν άνθρωπο θα πρέπει να μπορεί να κρατήσει επώνυμο, όνομα, ημερομηνία γέννησης, τόπο γέννησης, διεύθυνση κατοικίας, αριθμό τηλεφώνου κλπ. Ο πίνακας δεν μπορεί να φιλοξενήσει αυτά τα στοιχεία.

Η ανάγκη για ομαδοποίηση στοιχείων που αναφέρονται στο ίδιο φυσικό αντικείμενο παρουσιάστηκε πολύ νωρίς στις εμπορικές εφαρμογές. Δεν είναι λοιπόν περίεργο το ότι πρωτοπόρος στον τομέα ήταν η COBOL (COmmon Business Oriented Language) –και αργότερα τη μιμήθηκε η PL/I– ενώ Algol-60 και FORTRAN δεν είχαν οτιδήποτε σχετικό.

Η Pascal έδωσε τη δυνατότητα στον προγραμματιστή να ορίσει **τύπους εγγραφών** (record types) σύμφωνα με τις ανάγκες του, με απλό και κομψό τρόπο. Η Ada σχεδόν αντίγραψε την Pascal.

Οι **δομές** (`struct`(ure)s) της C μοιάζουν πολύ με τις εγγραφές της Pascal. Η C++ είδε τις δομές ως «*κλάσεις με όλα τα μέλη ανοικτά*». Η Java τις αγνόησε: επιτρέπει μόνον κλάσεις. Η C# τις είδε ως *κλάσεις με όλα τα μέλη ανοικτά που δεν κληρονομούν ούτε κληρονομούνται.*

Τιμές τέτοιων τύπων μπορούμε να βάλουμε και σε αρχεία. Τώρα πρόσεξε: αν τις γράψουμε (σχεδόν) όπως είναι αποθηκευμένες στη μνήμη (εσωτερική παράσταση) όλες θα καταλαμβάνουν τον ίδιο χώρο. Αυτό όμως μας δίνει τη δυνατότητα να έχουμε και *τυχαία* (και όχι μόνον σειριακή) *πρόσβαση* στις συνιστώσες του αρχείου. Αξίζει λοιπόν τον κόπο να δούμε τα μη μορφοποιημένα αρχεία και να καταλάβουμε τα μειονεκτήματα και τα πλεονεκτήματά τους.

15.1 Δομές

Είδαμε ότι οι πίνακες μας δίνουν έναν τρόπο πρόσβασης σε πολλά στοιχεία με ένα όνομα. Αλλά, τα στοιχεία αυτά πρέπει να είναι όλα του ίδιου τύπου.

Πολύ συχνά όμως, θέλουμε να αποθηκεύουμε και να επεξεργαζόμαστε στοιχεία διαφορετικού τύπου που αναφέρονται στο ίδιο αντικείμενο.

Παράδειγμα 1 ✍

Ένα πρόγραμμα μισθοδοσίας θα πρέπει να χρησιμοποιεί στοιχεία όπως:

επώνυμο
όνομα
ηλικία
βαθμός
ημερομηνία πρόσληψης
οικογενειακή κατάσταση
αριθμός παιδιών κ.λ.π.

Αυτά, ενώ όλα αναφέρονται σε ένα μισθωτό, δεν είναι του ίδιου τύπου ώστε να μπορούν να μπούν σε έναν πίνακα και να τα επεξεργαστούμε με έναν ενιαίο τρόπο.

Η C++ μας διευκολύνει με μία κατηγορία τύπων που προσφέρει γι' αυτές τις περιπτώσεις: τις **κλάσεις** (class). Η απλούστερη περίπτωση κλάσης είναι η **δομή** (structure) και με αυτήν θα αρχίσουμε τη γνωριμία μας με τη σύγχρονη τεχνική προγραμματισμού που λέγεται **αντικειμενοστρέφεια** (object orientation). Έτσι, για το παραπάνω παράδειγμα, μπορούμε να ορίσουμε τον τύπο:

```
struct Employee
{
   char    surname[24];
   char    firstname[16];
   Date    birthDate;
   int     rank;
   Date    emplDate;
   int     numberOfChld;
   Address home;
}; // Employee
```

Αν στην συνέχεια δηλώσουμε:

```
   Employee clerk, typist, manager;
```

μπορούμε μέσα στο πρόγραμμά μας να δίνουμε:

```
   strcpy( clerk.surname, "ΝΙΚΟΛΟΠΟΥΛΟΣ" );
```

ή

```
   if ( manager.birthDate < clerk.birthDate )
   {
      megalhYpotesh();
      grafto( clerk );
   }
   else
      tiPerimenes();
```

✍✍✍

Μια μεταβλητή (ή σταθερά) που ο τύπος της είναι δομή (ή γενικότερα: κλάση) ονομάζεται και **αντικείμενο** (object) αυτής της δομής (κλάσης).

Κι εδώ λοιπόν, όπως στους πίνακες, με το ίδιο όνομα αναφερόμαστε σε μία ομάδα μεταβλητών. Αντί όμως να χρησιμοποιήσουμε τις αγκύλες με το δείκτη για να τις ξεχωρίσουμε, βάζουμε, μετά το όνομα του αντικειμένου, μια τελεία (`.`) και το όνομα της μεταβλητής.

Οι μεταβλητές που εμφανίζονται στη δήλωση μιας δομής, όπως οι *surname*, *birthDate*, *numberOfChld*, *rank*, λέγονται **μέλη** (members)[2] της δομής.

Η περιγραφή μιας δομής ξεκινάει με τα
"`struct`", *όνομα*, "`{`"
και τελειώνει με "`};`". Ενδιαμέσως γράφονται τα μέλη του τύπου, όπως ακριβώς γράφονται οι δηλώσεις μεταβλητών:

---

[2] Γενικώς, στην Επεξεργασία Δεδομένων ονομάζονται **πεδία** (fields). Η C++ έχει για τον όρο αυτόν άλλη χρήση.

τύπος, *όνομα*, ";"

Παράδειγμα 2

Στην §8.13 είδαμε ότι για τον χειρισμό ρευμάτων αρχείων η C χρησιμοποιεί βέλη προς μεταβλητές τύπου *FILE*. Το πρότυπο της γλώσσας (ISO/IEC 1999) λέει ότι μια τέτοια μεταβλητή «*θα πρέπει να έχει τη δυνατότητα να καταγράφει όλην την πληροφορία που χρειάζεται για τον έλεγχο ρεύματος περιλαμβάνοντας ενδείκτη θέσης αρχείου, βέλος προς τον αντίστοιχο ενταμιευτή (αν υπάρχει), ενδείκτη σφάλματος που καταγράφει τον άν έγινε σφάλμα ανάγνωσης/εγγραφής, ενδείκτη τέλους αρχείου που καταγράφει αν έφτασε το τέλος αρχείου.*»

Η Borland C++, v.5.5 υλοποιεί τον τύπο ως εξής:

```
struct FILE
{
   unsigned char* curp;        // ενδείκτης θέσης αρχείου
   unsigned char* buffer;      // βέλος προς ενταμιευτή
   int            level;       // επίπεδο πληρότητας ενταμιευτή
   int            bsize;       // μέγεθος ενταμιευτή
   unsigned short istemp;      // ενδείκτης προσωρινότητας αρχείου
   unsigned short flags;       // σημαίες κατάστασης αρχείου
   wchar_t        hold;        // για ungetc αν δεν υπάρχει
                               // ενταμιευτής
   char           fd;          // File descriptor
   unsigned char  token;       // Used for validity checking
}; // FILE
```

Παραδείγματα 3

```
struct complex
{
   double re;
   double im;
}; // complex
```

Εδώ έχουμε μια δομή που κάθε αντικείμενό της θα έχει δυο μέλη τύπου **double**. Το ένα με το όνομα *re* και το άλλο με το όνομα *im*. Όπως φαίνεται και από το όνομα (complex = μιγαδικός), τα αντικείμενα της δομής *complex* μπορούν να παραστήσουν μιγαδικούς αριθμούς. Θα μπορούσαμε να τη γράψουμε και ως:

```
struct complex
{
   double re, im;
}; // complex
```

αλλά ο πρώτος τρόπος είναι προτιμότερος.

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
}; // Date
```

Οι μεταβλητές αυτού του τύπου κρατούν ημερομηνίες ενώ του επόμενου κρατούν διευθύνσεις:

```
struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address
```

Μετά τη δήλωση της δομής μπορούμε να δηλώσουμε μεταβλητές (αντικείμενα) της δομής. Μετά τις δηλώσεις των τύπων που είδαμε παραπάνω μπορούμε να δηλώσουμε:

```
complex i, j, k;
Date    yesterday, today, tomorrow;
Address residence;
```

Σημείωση: ▶

Όπως είπαμε, η **struct** υπήρχε και στη C, από όπου την κληρονόμησε (και την εμπλούτισε) η C++. Θα τη βρεις λοιπόν πολύ συχνά μπροστά σου. Αλλά εκεί ο χειρισμός των ορισμών και δηλώσεων είναι κάπως διαφορετικός. Π.χ., για να μπορέσεις να κάνεις τη δήλωση:

```
complex i, j, k;
```

θα πρέπει να έχεις ορίσει: [3]

```
typedef struct {  double re;  double im;  }  complex;
```

◀

Μια δομή μπορεί να έχει μέλη τύπου πίνακα ή δομής· π.χ. στον τύπο *Employee* έχουμε δύο μέλη δομής *Date* και ένα μέλος δομής *Address*, ενώ ο τύπος των *surname*, *firstname* είναι πίνακες.

Τέλος, όπως στις απλές μεταβλητές και τους πίνακες έτσι και στα αντικείμενα μπορείς να δώσεις αρχική τιμή με τη δήλωση, π.χ.:

```
complex j = { 0, 1 }, isqrt2 = { 1/sqrt(2), 1/sqrt(2) };
Date d1 = { 2004, 11, 17 };
```

Μετά από αυτό, οι:

```
cout << "( " << j.re << ", " << j.im << " )" << endl;
cout << "( " << isqrt2.re << ", " << isqrt2.im << " )" << endl;
cout << int(d1.day) << '.' << int(d1.month)
     << '.' << d1.year << endl;
```

θα δώσουν:

```
( 0, 1 )
( 0.707107, 0.707107 )
17.11.2014
```

Δηλαδή, με τη δήλωση: **d1 = { 2004, 11, 17 }** η πρώτη τιμή (2004) πηγαίνει στο πρώτο μέλος (*d1.year*), η δεύτερη (11) στο δεύτερο (*d1.month*) κ.ο.κ.

Μπορείς να διαχειρίζεσαι τα αντικείμενα είτε ανά μέλος, όπως θα δούμε στην επόμενη παράγραφο, είτε ολόκληρα: την τιμή μιας τέτοιας μεταβλητής

- μπορείς να την εκχωρείς σε μιαν άλλη, του ίδιου τύπου,
- μπορείς να τη δίνεις ως παράμετρο στην κλήση κάποιας συνάρτησης,
- μπορείς να τη βάζεις σε εντολή **return** και να επιστρέφεται ως τιμή συνάρτησης (του ίδιου τύπου).

Παράδειγμα 4 🗲

Η συνάρτηση:

```
complex conj( complex c )
{
   complex cj = { c.re, -c.im };
   return cj;
} // conj
```

επιστρέφει τον συζυγή του ορίσματος. Οι:

```
k = conj( j );
cout << "( " << k.re << ", " << k.im << " )" << endl;
```

θα δώσουν:

---

[3] Αν κρατήσεις τον αρχικό ορισμό θα πρέπει να δηλώσεις τις μεταβλητές ως:

```
struct complex i, j, k;
```

```
( 0, -1 )
```

15.1.1 Παράμετρος – Δομή

Μπορείς να δώσεις ένα αντικείμενο «*ως παράμετρο στην κλήση κάποιας συνάρτησης*» αλλά πρέπει να λάβεις υπόψη σου και μερικά πράγματα σχετικώς με τη διαδικασία περάσματος των παραμέτρων (§13.3, 14.8).

Δες την *conj()* και την κλήση "`k = conj(j)`"· για την εκτέλεση της κλήσης δημιουργείται μια τοπική μεταβλητή *c* –στη στοίβα– με αρχική τιμή ίση με αυτήν της *j*. Στην περίπτωσή αυτήν η αντιγραφή της *j* στη *c* σημαίνει αντιγραφή 16 ψηφιολέξεων (ή κάτι παρόμοιο). Αν όμως το αντικείμενό μας είναι μεγάλο η αντιγραφή μπορεί

- να μεγαλώσει τον χρόνο εκτέλεσης του προγράμματός μας και
- να φορτώσει πολύ τη στοίβα.

Έτσι, αντί για παράμετρο τιμής επιλέγουμε γενικώς την εξής λύση:

```
complex conj( const complex& c )
{
   complex cj = { c.re, -c.im };
   return cj;
} // conj
```

Δηλαδή παράμετρο αναφοράς με "`const`" ωστε να μην επιτρέπεται η αλλαγή τιμής του ορίσματος. Με αυτόν τρόπο όταν καλείται η συνάρτηση αντιγράφεται μόνον ένα βέλος.

Εδώ θα πρέπει να κάνουμε και μια συμπλήρωση του κανόνα που δώσαμε στην §13.3:

♦ ***Η πραγματική παράμετρος που αντιστοιχεί σε μια παράμετρο αναφοράς χωρίς "`const`" δεν μπορεί να είναι σταθερά ή παράσταση, αλλά κάτι που να μπορεί να αλλάξει η τιμή του δηλαδή τροποποιήσιμη τιμή-l, π.χ. μια μεταβλητή ή ένα στοιχείο πίνακα.***

Αν η παράμετρος αναφοράς έχει "`const`" τότε το αντίστοιχο όρισμα μπορεί να είναι «*σταθερά ή παράσταση*».

Δηλαδή η παράμετρος αναφοράς με "`const`" μπορεί να υποκαταστήσει πλήρως την παράμετρο τιμής; Σχεδόν·

- Αν σου χρειάζεται, μπορείς να χρησιμοποιήσεις την παράμετρο τιμής ως τοπική μεταβλητή και να αλλάξεις την τιμή της χωρίς να αλλάξει η τιμή του ορίσματος.
- Η τιμή παραμέτρου αναφοράς με "`const`" δεν επιτρέπεται να αλλάξει.

15.2 Μέλη Δομής

Στην προηγούμενη παράγραφο είδαμε το μέλος ως **χαρακτηριστικό** (attribute) μιας δομής. Είπαμε όμως και ότι μπορούμε να χειριζόμαστε τα αντικείμενα μιας δομής ανά μέλος.

♦ ***Μπορούμε να μεταχειριζόμαστε κάθε μέλος ενός αντικειμένου όπως οποιαδήποτε μεταβλητή του τύπου του.***

Όπως είπαμε και στην εισαγωγή αυτού του κεφαλαίου, ένα μέλος αναφέρεται με το όνομα του αντικειμένου, μια τελεία και το όνομα του μέλους.

Παράδειγμα 1

Με βάση τα παραδείγματα της προηγούμενης παραγράφου έχουν νόημα τα:

```
i.re
j.im
today.month
residence.city
```

Στον τύπο *complex* υπάρχουν τα μέλη *re* και *im*. Τα `i.re` και `j.im` είναι μέλη των μεταβλητών *i*, *j* τύπου *complex*. Παρομοίως, τα *month* και *city* είναι μέλη των δομών *Date* και

Address αντιστοίχως. Αφού έχουμε δηλώσει: `Date today` και `Address residence` τα `today.month` και `residence.city` είναι μέλη των μεταβλητών *today* και *residence* αντιστοίχως.

Με βάση τον ορισμό της δομής *Employee* στην προηγούμενη παράγραφο, μπορούμε να έχουμε:

```
clerk.emplDate
```

Αυτό το μέλος μπορούμε να το χειριζόμαστε ως μεταβλητή τύπου *Date*. Φυσικά μπορούμε να έχουμε και μέλη αυτού του αντικειμένου:

```
clerk.emplDate.year
clerk.emplDate.day
```

Η **εμβέλεια** (scope) του ονόματος ενός μέλους είναι η δομή όπου ορίζεται. Έτσι, μπορείς να χρησιμοποιείς ως όνομα μέλους ένα όνομα που το χρησιμοποιείς και αλλού στο πρόγραμμά σου, χωρίς να υπάρχει κίνδυνος σύγχυσης του μεταγλωττιστή. Πάντως *υπάρχει κίνδυνος σύγχυσης των προγραμματιστών*: κάτι τέτοιο μπορεί να κάνει το πρόγραμμά σου λιγότερο ευανάγνωστο. Είναι λοιπόν προτιμότερο να το αποφεύγεις.

Ας δούμε τώρα την εντολή εκχώρησης για αντικείμενα. Αν έχουμε ορίσει:

```
struct Q
{
   T1 x1;  T2 x2; ... Tn xn;
}; // Q
```

και δηλώσουμε:

```
Q a, b;
```

τότε η εντολή

```
a = b;
```

είναι ισοδύναμη με τις:

```
a.x1 = b.x1;  a.x2 = b.x2; ... a.xn = b.xn;
```

Παράδειγμα 2

Η εντολή:

```
today = tomorrow;
```

(δες την προηγούμενη παράγραφο) ισοδυναμεί με τις:

```
today.day   = tomorrow.day;
today.month = tomorrow.month;
today.year  = tomorrow.year;
```

15.3 Δημιουργοί

Είδαμε πώς μπορούμε να δίνουμε αρχικές τιμές σε μια μεταβλητή-δομή:

```
complex j = { 0, 1 }, isqrt2 = { 1/sqrt(2), 1/sqrt(2) };
Date d1 = { 2004, 11, 17 };
```

Αυτό το στυλ είναι κληρονομιά από τη C. Μπορούμε να χρήσιμοποιούμε παρενθέσεις όπως κάνουμε με τις άλλες μεταβλητές της C++; Ναι, αρκεί να ορίσουμε έναν δημιουργό.

Ο **δημιουργός** ή **κατασκευαστής** (constructor) είναι μια συνάρτηση που περιγράφει τι πρέπει να γίνει για να δημιουργηθεί μια τιμή ενός τύπου. Το όνομά της είναι το όνομα του τύπου και δεν έχει (άλλον) τύπο αφού είναι μέρος του ορισμού του τύπου.

Ας ορίσουμε έναν δημιουργό που θα μας επιτρέπει να δηλώσουμε:

```
complex j( 0, 1 ), isqrt2( 1/sqrt(2), 1/sqrt(2) );
```

Αυτό γίνεται με συμπλήρωση του ορισμού της δομής ως εξής:

```
struct complex
```

```
{
   double re;
   double im;
   complex( double rp, double ip )  {  re = rp;  im = ip;  };
}; // complex
```

Εδώ έχουμε έναν πολύ απλό δημιουργό: το μόνο που κάνει είναι να βάζει το πρώτο όρισμά του στο μέλος *re* και το δεύτερο στο μέλος *im*.

Μπορείς να δοκιμάσεις τη δήλωση με τις παρενθέσεις και θα δεις ότι «περνάει». Μπορείς όμως να δεις ότι χάσαμε τη δυνατότητα να κάνουμε τη δήλωση: `complex q`. Αυτό διορθώνεται με έναν δεύτερο δημιουργό:

```
struct complex
{
   double re;
   double im;
   complex( double rp, double ip )  {  re = rp;  im = ip;  };
   complex() {  re = 0.0;  im = 0.0;  };
}; // complex
```

Αυτό που ορίζουμε είναι ότι αν δεν δοθούν από τον προγραμματιστή αρχικές τιμές βάζουμε *ερήμην* αρχικές τιμές μηδέν και στα δύο μέλη. Αυτός είναι ο λεγόμενος **ερήμην δημιουργός** (default constructor). Αυτός ο δημιουργός θα κληθεί όταν δηλώνουμε και έναν πίνακα με στοιχεία τύπου *complex* (μια φορά για κάθε στοιχείο).

Γενικώς, μπορούμε να ορίζουμε πολλούς δημιουργούς, αλλά αυτούς τους δύο μπορούμε να τους συγχωνεύσουμε σε έναν. *Ο δημιουργός είναι συνάρτηση*· ειδική περίπτωση συνάρτησης αλλά συνάρτηση. Μπορούμε λοιπόν να χρησιμοποιήσουμε προκαθορισμένες τιμές παραμέτρων (§14.2):

```
struct complex
{
   double re;
   double im;
   complex( double rp=0.0, double ip=0.0 )
   {  re = rp;  im = ip;  };
}; // complex
```

Τώρα έχουμε έναν δημιουργό «2 σε 1»: έναν ερήμην δημιουργό που όμως μπορεί να δεχθεί και αρχικές τιμές.

Πρόσεξε και μια άλλη χρήση του δημιουργού με αρχικές τιμές: Δηλώνουμε

```
   complex q;
```

και στη συνέχεια της δίνουμε τιμή ως εξής:

```
   q = complex( 1.7, 8.1 );
```

Δηλαδή, καλούμε τον δημιουργό για να δημιουργήσει μια τιμή τύπου *complex* και στη συνέχεια την εκχωρούμε στην *q*. Έτσι, θα μπορούσαμε να ξαναγράψουμε την *conj* πιο απλά:

```
complex conj( complex c )
{
   return complex( c.re, -c.im );
} // conj
```

Και για την κλάση *Date* θα μπορούσαμε να ορίσουμε:

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
   Date( int yp=1, int mp=1, int dp=1 )
   {  year = yp;  month = mp;  day = dp;  }
}; // Date
```

Αυτός ο δημιουργός μας δίνει το δικαίωμα να δηλώσουμε:

```
   Date d0( 2011, 3, 5 )  // d0 == 05.03.2011
   Date d1;               // d1 == 01.01.0001
```

αλλά και:

```
Date d2( 2011, 3 )     // d2 == 01.03.2011
Date d3( 2011 );       // d3 == 01.01.2011
```

Πάντως, εδώ να επισημάνουμε ένα πρόβλημα: Ο δημιουργός δεν μας απαγορεύει να δηλώσουμε:

```
Date d1( 2004, 14, 37 );
```

Θα πρέπει να τον εφοδιάσουμε με μερικούς ελέγχους ώστε να απαγορεύει τέτοια λάθη.

Περισσότερα για τους δημιουργούς αργότερα, όταν θα συζητούμε για κλάσεις.

15.3.1 Αποκάλυψη Τώρα!

Πώς κάναμε το παράδειγμα της §11.1; Ορίζοντας τον τύπο:

```
struct MyType
{
   int v;
   MyType( int a=0 )
   {
      cout << "creating a MyType variable. Initialize with "
           << a << endl;
      v = a;
   }
   ~MyType()
   {
      cout << "destroying a MyType variable having value "
           << v << endl;
   }
}; // MyType
```

Ναι, αλλά εδώ έχει καινούρια πράγματα· τι είναι αυτό το *~MyType()*; Αυτή η συνάρτηση είναι ο **καταστροφέας** (destructor) της *MyType*. Καλείται για να καταστρέψει μια μεταβλητή τύπου *MyType*. Αν ξεχάσουμε το παράδειγμά μας με τα μηνύματα καταστροφής, η *MyType* δεν χρειάζεται καταστροφέα διότι η καταστροφή γίνεται αυτομάτως. Σε άλλες περιπτώσεις όμως είναι απαραίτητος.

15.4 Βέλος προς Τιμή–Δομή

Πολύ συχνά θα χρειαστεί να χειριστούμε κάποια τιμή-δομή με βέλος προς αυτήν. Η C++ μας δίνει μια μικρή διευκόλυνση για τον χειρισμό των μελών.

Ας πούμε ότι δηλώνουμε:

```
complex  q;
complex* pC( &q );
```

Μπορούμε να δώσουμε τιμές στα μέλη της *q* μέσω του *pC* ως εξής:

```
(*pC).re = 2.5;  (*pC).im = 1.03;
```

και να γράψουμε τις τιμές τους ως εξής:

```
cout << "*pC = ( " << (*pC).re << ", " << (*pC).im << " )"
     << endl;
```

Μέχρι εδώ τίποτε περίεργο: με την αποπαραπομπή (`*pC`) παίρνουμε τη μεταβλητή (*q*) που δείχνει το βέλος *pC*. Από εκεί και πέρα χρησιμοποιούμε την τελεία για να ξεχωρίσουμε τα μέλη.

Η C++ μας επιτρέπει να κάνουμε τα παραπάνω ως εξής:

```
pC->re = 2.5;  pC->im = 1.03;
cout << "*pC = ( " << pC->re << ", " << pC->im << " )" << endl;
```

Τι κερδίσαμε; Αντί να γράφουμε για κάθε μέλος "`(*).`" γράφουμε: "`->`".

Θα βλέπεις συχνά αυτόν τον συμβολισμό και –παρ' όλο που μπορείς να ζήσεις χωρίς αυτόν– καλά θα κάνεις να τον χρησιμοποιείς.

15.5 Επιφόρτωση Τελεστών για Τύπους Δομών

Αν έχουμε ορίσει, όπως πριν:

```
struct Q
{
   T1 x1;  T2 x2; ... Tn xn;
}; // Q
```

και δηλώσουμε:

```
Q a, b;
```

τότε η σύγκριση

```
a == b
```

είναι, κατ' αρχήν, ισοδύναμη με τις:

```
(a.x1 == b.x1) && (a.x2 == b.x2) && ... && (a.xn == b.xn)
```

Αυτή σύγκριση δεν γίνεται αυτομάτως. Για κάθε τύπο δομής θα πρέπει να επιφορτώνεις τον τελεστή "`==`" αν τον χρειάζεσαι. Πώς θα γίνει αυτό, ας πούμε για τον τύπο `complex`; Όπως είπαμε στην §14.6.4, δηλαδή:[4]

```
bool operator==( const complex& lhs, const complex& rhs )
{
   return lhs.re == rhs.re && lhs.im == rhs.im;
} // bool operator==( const complex
```

και μπορείς να κάνεις συγκρίσεις όπως:

```
if ( isqrt2 == j ) . . .
```

Για τον "`==`" (και τους άλλους τελεστές σύγκρισης) θα τα ξαναπούμε στη συνέχεια.

Παρομοίως θα επιφορτώσεις και τις αριθμητικές πράξεις "`+`", "`-`", "`*`", "`/`".

Ας πούμε τώρα ότι θέλεις να επιφορτώσεις το πρόσημο "`-`" που είναι ενικός (προθεματικός) τελεστής. Σύμφωνα με αυτά που λέμε στην §14.6.4, θα έχουμε:

```
complex operator-( const complex& lhs )
{
   return complex( -lhs.re, -lhs.im );
} // operator-( const complex
```

Ας πούμε τώρα ότι θέλουμε να μπορούμε να γράφουμε:

```
cout << isqrt2 << endl;
```

και να έχουμε το ίδιο αποτέλεσμα που έχουμε με την:

```
cout << "( " << isqrt2.re << ", " << isqrt2.im << " )" << endl;
```

Θα επιφορτώσουμε τον τελεστή "`<<`" όπως μάθαμε στην §14.6.1:

```
ostream& operator<<( ostream& tout, const complex rhs )
{
   return tout << "( " << rhs.re << ", " << rhs.im << " )";
} // operator<<( ostream& tout, const complex
```

Μπορούμε να τον επιφορτώσουμε και για τη *Date* ώστε η:

```
cout << d1 << endl;
```

να κάνει το ίδιο με την

```
cout << int(d1.day) << '.' << int(d1.month)
     << '.' << d1.year << endl;
```

Και αυτή η επιφόρτωση είναι απλή:

---

[4] Κατά τη συνήθειά μας, θα υπενθυμίσουμε ότι σύγκριση για ισότητα στον τύπο `double` δεν έχει και πολύ νόημα.

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << static_cast<unsigned int>(rhs.day) << '.'
               << static_cast<unsigned int>(rhs.month) << '.'
               << rhs.year;
} // operator<<( ostream& tout, const Date
```

Να τον επιφορτώσουμε και για τον τύπο *Address*; Δοκίμασέ το, αλλά δεν θα σου αρέσει!...

Στην §15.1 έχουμε τη σύγκριση "`manager.birthDate < clerk.birthDate`". Για να μπορεί να γίνει θα πρέπει να επιφορτώσουμε τον τελεστή "`<`" για τον τύπο *Date*. Αν έχουμε

```
Date d1, d2;
```

η σύγκριση `d1 < d2` θα πρέπει να δίνει τιμή `true` μόνο στην περίπτωση που η ημερομηνία *d*1 προηγείται της *d*2.

Ακολουθώντας αυτά που λέγαμε στην§14.6.4, θα πρέπει να κάνουμε την επιφόρτωση με μια συνάρτηση της μορφής:

```
Trv operator<( Tl lhs, Tr rhs )
```

όπου, στην περίπτωσή μας:

- *Tl* και *Tr* είναι ο `const Date&` και
- *Trv* είναι ο `bool`.

δηλαδή:

```
bool operator<( const Date& lhs, const Date& rhs )
```

Η υλοποίηση είναι απλή: ξεκινάμε από τα έτη. Αν δεν βγαίνει συμπέρασμα –αν δηλαδή είναι ίσα– πάμε στους μηνες και αν δεν βγαίνει και εκεί συμπέρασμα πάμε στις ημέρες:

```
bool operator<( const Date& lhs, const Date& rhs )
{
   bool fv;

   if ( lhs.year < rhs.year )           fv = true;
   else if ( lhs.year > rhs.year )      fv = false;
   else // lhs.year == rhs.year
   {
      if ( lhs.month < rhs.month )      fv = true;
      else if ( lhs.month > rhs.month ) fv = false;
      else // lhs.year == rhs.year && lhs.month == rhs.month
         fv = ( lhs.day < rhs.day );
   }
   return fv;
} // operator<( const Date . . .
```

Όπως καταλαβαίνεις, τώρα μπορούμε να ορίσουμε πολλούς δικούς μας τύπους και να επιφορτώσουμε πολλούς τελεστές σε αυτούς.

15.5.1 Συγκρίσεις και Κλειδιά

Είπαμε παραπάνω ότι η ισότητα ορίζεται όπως είδαμε «κατ' αρχήν». Τι θα πει αυτό; Αυτός ο ορισμός της ισότητας είναι ο –ας πούμε– «προγραμματιστικός». Αν πάρουμε όμως υπόψη μας και το νόημα των αντικειμένων τα πράγματα μπορεί να απλουστευθούν. Ας πούμε ότι σε μια εφαρμογή, όπου χρησιμοποιούμε αντικείμενα τύπου *Employee*, έχουμε δεχθεί ότι *δεν υπάρχει περίπτωση να έχουμε δύο εργαζόμενους με ίδιο όνομα, επώνυμο και ημερομηνία γέννησης*. Με βάση αυτό μπορούμε να ορίσουμε την ισότητα μεταβλητών τύπου *Employee* ως εξής:

```
bool operator==( const Employee& lhs, const Employee& rhs )
{
   return ( strcmp(lhs.surname, rhs.surname)==0 &&
            strcmp(lhs.firstname, rhs.firstname)==0 &&
            lhs.birthDate == rhs.birthDate );
```

```
} // operator==( const Employee
```

Πρόσεξε ότι εδώ συγκρίνουμε αντικείμενα τύπου *Date*. Θα πρέπει λοιπόν να έχουμε ορίσει:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day == rhs.day );
}; // operator==( const Date
```

Το κέρδος μας είναι ότι δεν χρειάζεται να επιφορτώσουμε τον "==" και για τον τύπο *Address*.

Λέμε ότι τα μέλη (χαρακτηριστικά) *surname, firstname, birthDate* αποτελούν **κλειδί** (key) για αντικείμενα τύπου *Employee*.[5]

Φυσικά, με βάση το κλειδί γίνονται και οι επιφορτώσεις των άλλων τελεστών σύγκρισης. Ας πούμε ότι χρειαζόμαστε και τον "<" για τον *Employee*.

```
bool operator<( const Employee& lhs, const Employee& rhs )
{
   bool fv;
   if ( strcmp(lhs.surname, rhs.surname) < 0 ) fv = true;
   else if ( strcmp(lhs.surname, rhs.surname) > 0 ) fv = false;
   else // strcmp(lhs.surname, rhs.surname)==0
   {
      if ( strcmp(lhs.firstname, rhs.firstname) < 0 )  fv = true;
      if ( strcmp(lhs.firstname, rhs.firstname) > 0 )  fv = false;
      else // strcmp(lhs.surname, rhs.surname)==0 &&
           // strcmp(lhs.firstname, rhs.firstname)==0
         fv = lhs.birthDate < rhs.birthDate;
   }
   return fv;
}; // operator<( const Employee
```

Όπως βλέπεις είναι πιο πολύπλοκος από τον "==" και για να υλοποιηθεί θα πρέπει θα πάρουμε μια απόφαση: με ποια σειρά θα συγκρίνουμε τα μέρη του κλειδιού. Εδώ είπαμε: πρώτα τα επώνυμα –αλφαβητικώς– μετά τα ονόματα –και αυτά αλφαβητικώς– και τέλος οι ημερομηνίες γέννησης. Για την τελευταία σύγκριση θα πρέπει να επιφορτώσουμε τον "<" για τον *Date*· αυτό σου το αφήνουμε ως άσκηση.

15.6 Αποθήκευση Μελών Δομής

Το πρότυπο της C++ λέει ότι

- ***Σε μια τιμή-δομή η διεύθυνση του κάθε μέλους είναι ανώτερη (μεγαλύτερη) από τη διεύθυνση του προηγούμενου (κατά τη σειρα δήλωσης).***

Τίποτε περισσότερο!

Δουλεύοντας με ορισμένους μεταγλωττιστές μπορεί να σχηματίσεις μια λαθεμένη εικόνα. Στην επόμενη υποπαράγραφο σου δείχνουμε πώς μπορείς να το ψάξεις.

15.6.1 * Σκαλίζοντας τη Μνήμη

Για να το επιβεβαιώσουμε δοκιμάζουμε το παρακάτω πρόγραμμα:

```
#include <iostream>
```

[5] Το συγκεκριμένο κλειδί του παραδείγματος δεν είναι ικανοποιητικό. Η ταυτοποίηση προσώπων είναι χαρακτηριστική περίπτωση όπου χρειαζόμαστε κλειδί με πολλά χαρακτηριστικά. Συνήθως, σε τέτοιες περιπτώσεις, ορίζουμε ένα πιο εύχρηστο **υποκατάστατο** (surrogate) κλειδί όπως είναι ο αριθμός ταυτότητας, ο ΑΦΜ, ο ΑΜΚΑ και διάφοροι άλλοι αριθμοί μητρώου.

```
using namespace std;

struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address

int main()
{
   Address a;

   cout << " sizeof a: " << (sizeof a) << endl;
   cout << " members: "
        << sizeof(a.country) + sizeof(a.city) +
           sizeof(a.areaCode) + sizeof(a.street) +
           sizeof(a.number) << endl;
   cout << " address of a: "
        << reinterpret_cast<long int>(&a) << endl;
   cout << " address of a.country: "
        << reinterpret_cast<long int>(a.country) << "  "
        << reinterpret_cast<long int>(&a) << endl;
   cout << " address of a.city: "
        << reinterpret_cast<long int>(a.city) << "  "
        << reinterpret_cast<long int>(a.country)+sizeof(a.country)
        << endl;
   cout << " address of a.areaCode: "
        << reinterpret_cast<long int>(&a.areaCode) << "  "
        << reinterpret_cast<long int>(a.city)+sizeof(a.city)
        << endl;
// . . .
```

Όπως βλέπεις, προσπαθούμε να συγκρίνουμε

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (`sizeof a`) με το άθροισμα των μεγεθών των μελών της (`sizeof(a.country) + sizeof(a.city) + sizeof(a.areaCode) + sizeof(a.street) + sizeof(a.number)`),
- Τη διεύθυνση του πρώτου μέλους (`a.country`) με τη διεύθυνση της μεταβλητής-δομής (`&a`).
- Τη διεύθυνση του κάθε επόμενου μέλους (π.χ. `a.city`) με αυτήν που προκύπτει αν στη διεύθυνση του προηγούμενου (`a.country`) προσθέσουμε το μέγεθός του (`sizeof(a.country)`).

Το `reinterpret_cast<long int>(&a)` σημαίνει: ερμήνευσε την εσωτερική παράσταση του βέλους `&a` ως τιμή τύπου `long int` (δες την επόμενη παράγραφο)· τελικώς, μας δίνει τη διεύθυνση σε δεκαδική μορφή αντί για δεκαεξαδική.

Αν περάσουμε το πρόγραμμά μας από τον Borland C++ 5.02 (για Win32) θα πάρουμε:

```
sizeof a: 59
members: 59
address of a: 1245008
address of a.country: 1245008  1245008
address of a.city: 1245025  1245025
address of a.areaCode: 1245042  1245042
address of a.street: 1245046  1245046
address of a.number: 1245063  1245063
```

Δηλαδή:

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (59) ισούται με το άθροισμα των μεγεθών των μελών της.

- Η αποθήκευση του πρώτου μέλους (1245008) αρχίζει εκεί που αρχίζει η αποθήκευση της *a*.
- Η αποθήκευση του κάθε επόμενου μέλους αρχίζει ακριβώς μετά το τέλος του προηγούμενου.

Αν όμως χρησιμοποιήσουμε τον Borland C++ 5.5 (για Win32) –που συμμορφώνεται πολύ περισσότερο με το πρότυπο της C++– θα πάρουμε:

```
sizeof a: 64
members: 59
address of a: 1245004
address of a.country: 1245004  1245004
address of a.city: 1245021  1245021
address of a.areaCode: 1245040  1245038
address of a.street: 1245044  1245044
address of a.number: 1245064  1245061
```

που είναι μια διαφορετική εικόνα. Παρόμοια θα πάρεις και από τον gcc (Dev-C++). Δηλαδή:

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (64) *δεν* ισούται με το άθροισμα των μεγεθών των μελών της (59).
- Η αποθήκευση του μέλους `a.areaCode` *δεν* αρχίζει ακριβώς μετά το τέλος του προηγούμενου (1245038) αλλά μετά από δύο ψηφιολέξεις (1245040). Παρομοίως, η αποθήκευση του μέλους `a.number` *δεν* αρχίζει ακριβώς μετά το τέλος του προηγούμενου (1245061) αλλά μετά από τρεις ψηφιολέξεις.

Θα περιγράψουμε με απλά λόγια έναν λόγο που δεν ισχύουν γενικώς τα παραπάνω: τη λεγόμενη **πλήρωση** (padding). Ορισμένοι υπολογιστές, για να αυξήσουν την απόδοσή τους δεν ξεκινούν την αποθήκευση μιας μεταβλητής από οποιαδήποτε ψηφιολέξη αλλά *από την αρχή μιας λέξης* (word) που μπορεί να περιέχει, π.χ., 2 ή 4 ψηφιολέξεις. Αν λοιπόν έχεις μια:

```
struct cl1
{
   char c1;
   int klm;
// ...
};
```

σε έναν υπολογιστή που θέλει τα πάντα να ξεκινούν από την αρχή λέξης τεσσάρων ψηφιολέξεων, το *c*1 θα καταλάβει μεν μια ψηφιολέξη αλλά οι επόμενες τρεις θα παραμείνουν κενές.

Κάτι τέτοιο κάνουν στο παράδειγμά μας οι μεταγλωττιστές Borland C++ 5.5 και gcc: η αποθήκευση των δύο μελών τύπου `int` αρχίζει απο᾽διεύθυνση που είναι πολλαπλάσιο του 4.

15.7 Ερμηνευτική Τυποθεώρηση

Ένας νεόκοπος προγραμματιστής (που είχε ξεχάσει αυτά που λέγαμε στο Κεφ. 1 και θα παρουσιάσουμε εν εκτάσει στο επόμενο κεφάλαιο) είχε την εξής απορία: Στη C++ που χρησιμοποιούσε οι τιμές `long int` έπιαναν 4 ψηφιολέξεις, το ίδιο και οι τιμές `float`. Πόσο να μοιάζουν άραγε οι παραστάσεις των `7.345F` και `7L`, Αν δούμε τα δυαδικά ψηφία του `7.345F` ως τιμή τύπου `long int`, θα δούμε άραγε κάποιο εφτάρι (“`111`”); (!!!) Η πρώτη σκέψη ήταν να δοκιμάσει τα:

```
   float    ff( 7.345 );
   long int nn( static_cast<long>(ff) );
```

αλλά μέχρι εδώ ήξερε: Καμιά σχέση με αυτό που ήθελε, διότι με το `static_cast<long>(ff)`

- *κάνει μετατροπή σε άλλη εσωτερική παράσταση* (από `float` σε `long`) και φυσικά

- *αλλάζει την τιμή* (από 7.345 σε 7).

Μετά σκέφτηκε το εξής: να βάλει

```
long* pl( &ff );
cout << (*pl) << endl;
```

και να δει την τιμή του ***pl***. Αυτό είναι ακριβώς που θέλει να κάνει αλλά ο μεταγλωττιστής του είχε αντιρρήσεις:

```
"Cannot convert 'float*' to 'long*'"
```

Πώς μπορεί να γίνει; Έτσι:

```
long* pl( reinterpret_cast<long*>(&ff) );
cout << (*pl) << endl;
```

Αποτέλεσμα:

```
7  1089145405
```

Στην πρώτη περίπτωση βλέπουμε τα γνωστά και δεν έχουμε καμιά απορία για το πώς βγαίνει το «7» στη γραμμή αποτελεσμάτων. Να τονίσουμε ότι, όπως ξέρουμε και φαίνεται από το παράδειγμα,

- αλλάζει η *εσωτερική παράσταση* (από `long` σε `float`)
- αλλάζει η τιμή (από 7.345 σε 7).

Τι γίνεται στη δεύτερη περίπτωση; Στο βέλος *pl*, τύπου `long*`, δίνουμε ως τιμή την τιμή του βέλους `&ff`, που είναι τύπου `float*`. Και γιατί το γράφουμε έτσι; Διότι, όπως είδες, η C++, ελέγχοντας τις συμβατότητες τύπων, δεν θα μας επιτρέψει να γράψουμε `long* ll( &ff )`[6]. Εδώ

- αλλάζει μεν ο τύπος αλλά δεν αλλάζει η τιμή: το βέλος δείχνει την ίδια θέση της μνήμης,
- δεν αλλάζει η εσωτερική παράσταση (τα ίδια δυαδικά ψηφία),
  - ούτε για το βέλος,
  - ούτε για τον στόχο.

Έτσι, στο *pl* αποθηκεύεται η διεύθυνση της *ff* και έχουμε ένα βέλος τύπου `long*` να δείχνει μια μεταβλητή τύπου `float`. Αυτό που παίρνουμε από αποπαραπομπή του *pl* και βλέπεις ως αποτέλεσμα (1089145405) είναι αυτό που βγαίνει αν «διαβάσουμε» τα δυαδικά ψηφία της *ff* σαν να παριστάνουν μια τιμή τύπου `long`.

Γενικώς αν έχουμε:

```
reinterpret_cast< T >( Π )
```

Ο τύπος *T* πρέπει να είναι ακέραιος, βέλους, αναφοράς. Τα ίδια ισχύουν και για τον τύπο του αποτελέσματος της παράστασης *Π*. Όπως θα κατάλαβες, η **ερμηνευτική τυποθεώρηση** (reinterpretation casting), μας επιτρέπει να ερμηνεύσουμε τα δυαδικά ψηφία κάποιας θέσης της μνήμης με τους κανόνες κάποιου άλλου τύπου. Για τον ίδιο σκοπό μπορούμε να χρησιμοποιήσουμε και τη `union`, όπως θα δούμε στη συνέχεια.

Ας δούμε άλλο ένα παράδειγμα: Κάποιος, που μαθαίνει προγραμματισμό με C++, μαθαίνει ένα «μυστικό»: Αν, στη C++ που δουλεύει, μια τιμή *k*, τύπου `unsigned short int`, αποθηκεύεται σε δύο ψηφιολέξεις, τότε στη μια από αυτές υπάρχει το *k*`/`256 και στην άλλη το *k*`%`256. Αν, ας πούμε, έχουμε:

```
unsigned short int k( 8548 );
```

Η εντολή:

```
cout << (k%256) << "  " << (k/256) << endl;
```

δίνει:

```
100  33
```

[6] Και γιατί θα θέλαμε να κάνουμε κάτι τέτοιο; Θα δεις στη συνέχεια ότι αυτό είναι απαραίτητο σε πολλές περιπτώσεις.

Πώς μπορούμε να διαπιστώσουμε ότι αυτές οι τιμές υπάρχουν στις δύο ψηφιολέξεις; Θυμήσου αυτά που διάβασες την προηγούμενη παράγραφο· δηλώνουμε:

```
char* p;
```

Η *p* είναι μια μεταβλητή-βέλος που δείχνει προς μια μεταβλητή τύπου `char`. Και τώρα δίνουμε:

```
p = reinterpret_cast<char*>(&k);
```

Ποιο είναι το νόημά της; Πάρε το βέλος προς την *k* (τη διεύθυνση της *k*) και αφού το δεις ως βέλος προς μια θέση τύπου `char`, να το κάνεις τιμή της *p*.

Έτσι, έχουμε μια μεταβλητή-βέλος προς μια θέση τύπου `char`, που όμως δείχνει την *k*, μια μεταβλητή τύπου `int`. Όπως έχουμε πει στην §12.1, για τη C++ πίνακας και βέλος είναι το ίδιο πράγμα. Ε, τότε ας δοκιμάσουμε την:

```
cout << p[0] << "  " << p[1] << endl;
```

Λοιπόν: γίνεται δεκτή χωρίς πρόβλημα και δίνει:

```
d  !
```

ενώ αν δώσουμε:

```
cout << static_cast<int>(p[0]) << "  "
     << static_cast<int>(p[1]) << endl;
```

θα πάρουμε αποτέλεσμα:

```
100  33
```

Είδαμε (§15.7) πώς ερμηνεύουμε –με ερμηνευτική τυποθεώρηση– την τιμή βέλους (μια διεύθυνση) ως ακέραιη τιμή. Είπαμε όμως παραπάνω ότι, με τον ίδιο τρόπο μπορούμε να δούμε και έναν ακέραιο ως βέλος. Για παράδειγμα, ας πούμε ότι δηλώνουμε:

```
float a[10];
int   aToInt( reinterpret_cast<int>(a) );
```

Στην *aToInt* δίνουμε ως τιμή τη διεύθυνση που αρχίζει η αποθήκευση του πίνακα a. Η παράσταση:

```
reinterpret_cast<float*>( aToInt + k*sizeof(float) )
```

είναι ένας άλλος τρόπος να γράψεις το "`a+k`" ή "`&a[k]`". Φυσικά, *είναι σαφώς χειρότερος από τους άλλους δύο*.

Κατά το πρότυπο της C++, για την ερμηνευτική τυποθεώρηση το μόνο σίγουρο είναι το εξής: Αν μετατρέψεις ένα βέλος σε ακέραιο –με αρκετά μεγάλο μέγεθος[7]– και μετά τον ακέραιο στον ίδιο τύπο βέλους θα πάρεις την αρχική τιμή. Δηλαδή αν *IT* ακέραιος τύπος με ικανοποιητικό μέγεθος και

```
T* p;
```

τότε:

```
reinterpret_cast<T*>(reinterpret_cast<IT>(p)) ≡ p
```

Κατά τα άλλα:

- Η αντιστοίχιση βελών και ακεραίων εξαρτάται από την υλοποίηση.
- Για να την καταλάβεις θα πρέπει να ξέρεις τη δομή *διευθυνσιοδότησης* (addressing) του υπολογιστή σου.

Χρησιμοποιήσαμε και θα ξαναχρησιμοποιήσουμε την ερμηνευτική τυποθεώρηση για να «σκαλίσουμε» τη μνήμη του υπολογιστή και να καταλάβουμε πώς δουλεύουν ορισμένα πράγματα. Θα τη χρησιμοποιούμε σε «πραγματικό» πρόγραμμα; Μόνο σε μια περίπτωση: τη διαχείριση μη μορφοποιημένων αρχείων που θα δούμε παρακάτω. Γενικώς, θεωρείται ως επικίνδυνο εργαλείο.

---

[7] Για παράδειγμα: Στο περιβάλλον Windows (Win32) οι διευθύνσεις περιγράφονται με 32 δυαδικά ψηφία. Στη BC++ 5.5 και στη Dev-C++ σε 32 δυαδικά ψηφία αποθηκεύονται οι τιμές των τύπων `int` και `long int`.

15.8 * Ψηφιοπεδία

Ας ξεκινήσουμε με ένα παράδειγμα –ελαφρώς τροποποιημένο– από το (Kernighan & Ritchie 1988):

```
struct flags1
{
   bool isKeyword;
   bool isExtern;
   bool isStatic;
}; // flags1
```

Με την

```
   cout << "sizeof flags1: " << (sizeof(flags1)) << endl;
```

παίρνουμε:

```
sizeof flags1: 3
```

(ψηφιολέξεις) ενώ ξέρουμε ότι χρησιμοποιούμε 3 δυαδικά ψηφία. Μπορούμε να κάνουμε οικονομία; Ναι, έτσι:

```
struct flags2
{
   bool isKeyword: 1;
   bool isExtern:  1;
   bool isStatic:  1;
}; // flags2
```

Μια μεταβλητή τύπου *flags*2 αποθηκεύεται σε 1 ψηφιολέξη.

Τα μέλη *isKeyword*, *isExtern*, *isStatic* ονομάζονται **πεδία** (fields) ή –καλύτερα– **ψηφιοπεδία** (bit-fields). Γενικώς, στη δήλωση μιας δομής μπορείς να δηλώσεις ένα ψηφιοπεδίο ως εξής:

τύπος, *όνομα*, "`:`", *φυσικός*

Ο *τύπος* μπορεί να είναι: `int`, `unsigned int`, `char`, `unsigned char`, `wchar_t`, `bool`. Ο *φυσικός* δίνει το πλήθος των δυαδικών ψηφίων που θα χρησιμοποιηθούν για την παράσταση του ψηφιοπεδίου. Όπως καταλαβαίνεις, τα `int` και `char` δεν παίζουν και τόσο σημαντικό ρόλο. Το `unsigned` όμως είναι ουσιαστικό διότι, αν δεν υπάρχει, ένα δυαδικό ψηφίο θα χρησιμοποιηθεί για πρόσημο. Το όνομα μπορεί και να λείπει, οπότε το ψηφιοπεδίο δεν είναι προσβάσιμο (ούτε απ' ευθείας διαχειρίσιμο).

Η διαχείριση των ψηφιοπεδίων γίνεται με τους γνωστούς τελεστές επιλογής. Αν έχουμε δηλώσει:

```
   flags2  aSymbol;
   flags2* pToSymbol;
```

μπορούμε να χρησιμοποιούμε, με το νόημα που ξέρουμε, τα:

```
   aSymbol.isKeyword    aSymbol.isExtern
   pToSymbol->isExtern    pToSymbol->isStatic
```

Να δούμε άλλο ένα

Παράδειγμα

Έστω ότι σχεδιάζουμε έναν τύπο εγγραφής για να κρατούμε στοιχεία μιας χρονικής στιγμής:

```
struct time1
{
   unsigned short year;
   unsigned char  month;
   unsigned char  day;
   unsigned char  hour;
   unsigned char  min;
   unsigned char  sec;
}; // time1
```

Για όλα τα μέλη εκτός από το πρώτο, βάλαμε τύπο `unsigned char` –αν και θα τα χειριστούμε ως ακεραίους– με στόχο να παίρνουμε μία ψηφιολέξη μόνον για αποθήκευση. Μια μεταβλητή αυτού του τύπου καταλαμβάνει 8 ψηφιολέξεις (gcc - Dev-C++, BC++ v.5.5). Κάτι τέτοιο φαίνεται σπάταλο διότι:

- Για έτος μέχρι 4095 χρειαζόμαστε 12 δυαδικά ψηφία,
- για μήνα (1-12) όχι περισσότερα από 4 δυαδικά ψηφία (0 - 15),
- για ημέρα (1-31) όχι περισσότερα από 5 δυαδικά ψηφία (0 - 31),
- για ώρα (0 - 23) όχι περισσότερα από 5 δυαδικά ψηφία (0 - 31),
- για πρώτα λεπτά (0 - 59) όχι περισσότερα από 6 δυαδικά ψηφία (0 - 63)
- για δεύτερα λεπτά (0 - 59) όχι περισσότερα από 6 δυαδικά ψηφία (0 - 63).

Σύνολο: 38 δυαδικά ψηφία που μας τα δίνουν 5 ψηφιολέξεις και όχι 8 (που έχει 60% σπατάλη).

Η C++ σου επιτρέπει να περιορίσεις τη σπατάλη αν ορίσεις:

```
struct time2
{
   unsigned int year:  12;
   unsigned char month: 4;
   unsigned char day:   5;
   unsigned char hour:  5;
   unsigned char min:   6;
   unsigned char sec:   6;
}; // time2
```

που καταλαμβάνει 6 ψηφιολέξεις (gcc - Dev-C++, BC++ v.5.5).

Χρησιμοποιώντας δομές με ψηφιοπεδία μπορείς να κάνεις οικονομία στον χώρο που κατάλαμβάνουν τα στοιχεία σου, στη κύρια (και στη βοηθητική) μνήμη. Αλλά ποιο είναι το τίμημα; Το (εκτελέσιμο) πρόγραμμά σου γίνεται μεγαλύτερο και πιο αργό.

Μπορείς να χρησιμοποιείς ψηφιοπεδία μόνον μέσα σε **struct** ή σε **union** (τη βλέπεις στην επόμενη παράγραφο) ή σε **class** (θα τη δούμε αργότερα).

15.9 * union

Είδαμε ότι η ερμηνευτική τυποθεώρηση μας επιτρέπει να ερμηνεύσουμε το περιεχόμενο της μνήμης με διάφορους τρόπους. Έτσι είδαμε δύο διαδοχικές ψηφιολέξεις και ως τιμή τύπου `unsigned short int` και ως πίνακα τύπου `char` με δύο στοιχεία. Θα δούμε τώρα έναν άλλον τρόπο για να κάνουμε το ίδιο πράγμα.

Αυτό μπορεί να γίνει με ένα άλλο είδος δομής, αυτό της `union`. Ας το δούμε με ένα παράδειγμα. Ας πούμε ότι δηλώνουμε:

```
union U
{
   char c;
   int  i;
}; // U

   U  m;
```

και δίνουμε:

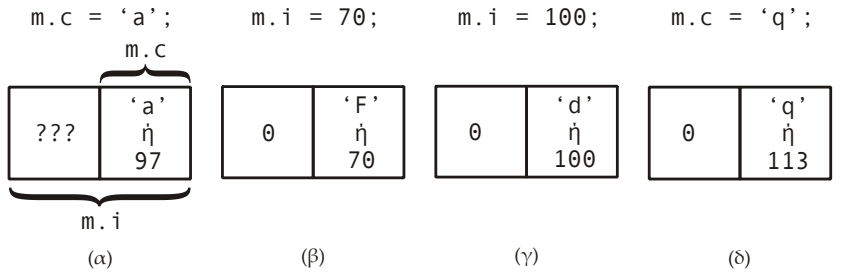
```
   m.c = 'a';  m.i = 70;
   cout << m.c << endl;
```

Αποτέλεσμα:

```
F
```

Δίνουμε:

```
   m.i = 180;  m.c = 'q';
```



Σχ. 15-1 Σχηματική παράσταση της `union { char c; int i; } m`. Το μέλος *m.i*, που είναι τύπου `int`, καταλαμβάνει δύο ψηφιολέξεις (α). Η μια από αυτές είναι η ψηφιολέξη που αποθηκεύεται το μέλος *m.c*. Αυτή η ψηφιολέξη είναι κοινή για τα δύο μέλη. Στο (α) βλέπεις την κατάσταση μετά την εκτέλεση της εντολής `m.c = 'a'`. Στην κοινή ψηφιολέξη αποθηκεύεται το `'a'`, που αν το δούμε ως ακέραιο έχει την τιμή 97. Στο (β) βλέπεις την κατάσταση μετά την εκτέλεση της εντολής `m.i = 70`. Η τιμή 70 αποθηκεύεται ως εξής: 70 στη μία ψηφιολέξη (την κοινή) και 0 στην άλλη. Αν ζητήσουμε να δούμε την τιμή του *m.c*, αυτή θα προκύψει αν δούμε ως χαρακτήρα το περιεχόμενο της κοινής ψηφιολέξης, δηλαδή `'F'`. Στα (γ) και (δ) βλέπεις την κατάσταση μετά τις εντολές `m.i = 180` και `m.c = 'q'` αντιστοίχως.

```
   cout << m.i << endl;
```

Αποτέλεσμα:

```
113
```

Δηλαδή δίνουμε στο *m.c* τιμή `'a'` και όταν την τυπώνουμε βγαίνει "`F`". Δίνουμε στο *m.i* τιμή 180 και μας βγαίνει "`113`". Τι γίνεται;

Σε μια **union** όλα τα μέλη αποθηκεύονται στην ίδια θέση της μνήμης. Ακριβέστερα, η αποθήκευση αρχίζει από την ίδια ψηφιολέξη, διότι τα μέλη μπορεί να έχουν διαφορετικά μεγέθη. Εδώ λοιπόν τι γίνεται; Έστω ότι μια θέση τύπου `int` πιάνει 2 ψηφιολέξεις και μια θέση τύπου `char` 1 ψηφιολέξη. Η αποθήκευση της μεταβλητής *m* και των μελών της φαίνεται στο Σχ. 15-1. Όπως καταλαβαίνεις, αλλάζοντας την τιμή του ενός μέλους αλλάζει ταυτοχρόνως και η τιμή του άλλου.

Θα μπορούσαμε λοιπόν να κάνουμε αυτό που είδαμε στην §15.7 ως εξής: Ορίζουμε μια

```
union U1
{
   char               c[2];
   unsigned short int i;
}; // U1
```

δηλώνουμε:

```
   U1 u1;
```

και οι

```
   u1.i = 8548;
   cout << u1.c[0] << "  " << u1.c[1] << endl;
   cout << static_cast<int>(u1.c[0]) << "  "
        << static_cast<int>(u1.c[1]) << endl;
```

θα δώσουν:

```
d  !
100  33
```

Πάντως η **union** μας δίνει και άλλες δυνατότητες. Ας πούμε, για παράδειγμα, ότι έχουμε την παράσταση:

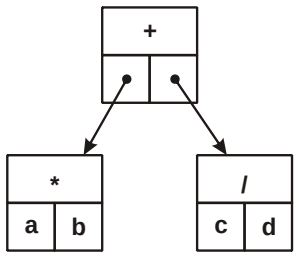
```
a * b + c / d
```

Συνήθως την παριστάνουμε όπως βλέπεις στο Σχ. 15-2. Αν θέλουμε να παραστήσουμε στο πρόγραμμά μας τον κόμβο του δένδρου έχουμε το εξής πρόβλημα: Έστω ότι ορίζουμε την:

```
   struct Node
```

```
   {
      char oper;
      Node* arg1;
      Node* arg2;
   }; // Node
```

με μέλη έναν χαρακτήρα, που σημειώνει την πράξη και δύο βέλη προς τους κόμβους των ορισμάτων. Αυτός ο τύπος είναι μια χαρά για τον κόμβο με την πρόσθεση. Αλλά δεν μας λύνει το πρόβλημα για τους άλλους δύο τύπους που έχουν αριθμητικά ορίσματα. Εκεί χρειαζόμαστε εγγραφές τύπου:



Σχ. 15-2 Δενδρική παράσταση απλής αριθμητικής παράστασης.

```
   struct Node
   {
      char oper;
      double arg1;
      double arg2;
   }; // Node
```

Το πρόβλημά μας μπορεί να λυθεί ως εξής· ορίζουμε:

```
struct Node;

union Arg
{
   double d;
   Node*  p;
}; // Arg

struct Node
{
   char oper;
   char leftArgKind;  Arg leftArg;
   char rightArgKind;  Arg rightArg;
}; // Node
```

Η αρχική `struct` *Node* είναι μια προαναγγελία δήλωσης, απαραίτητη για να γίνει δεκτός ο ορισμός της *Arg*, που έχει μέσα τη δήλωση `Node* p`.

Στην *Node*, εκτός από τα *oper*, *leftArg*, *rightArg*, υπάρχουν και τα μέλη *leftArgKind*, *rightArgKind*. Στην πρώτη αποθηκεύουμε μια τιμή που μας δείχνει ποιο από τα μέλη της *leftArg* είναι σε χρήση. Παρόμοιο ρόλο παίζει η *rightArgKind* σε σχέση με τη rightArg.

Ας πούμε ότι έχουμε δηλώσει:

```
   Node root, mulT, divT;
```

Στα μέλη της *pr* δίνουμε τις εξής τιμές:

```
   root.oper = '+';
   root.leftArgKind = 'p';  root.leftArg.p = &mulT;
   root.rightArgKind = 'p';  root.rightArg.p = &divT;
```

Όπως βλέπεις, στην *root*.*leftArgKind* δίνουμε τιμή στο μέλος (βέλος) *p* (τη διεύθυνση του κόμβου *mulT*). Στην *root*.*leftArgKind* δίνουμε τιμή `'p'` που σημαίνει ότι στη *root*.*leftArgKind* έχουμε σε χρήση το μέλος *p*.

Στα μέλη των *mulT* και *divT* δίνουμε:

```
   mulT.oper = '*';
   mulT.leftArgKind = 'd';  mulT.leftArg.d = a;
   mulT.rightArgKind = 'd';  mulT.rightArg.d = b;
   divT.oper = '/';
   divT.leftArgKind = 'd';  divT.leftArg.d = c;
   divT.rightArgKind = 'd';  divT.rightArg.d = d;
```

Τώρα στη *mulT*.*leftArg* δίνουμε τιμή στο μέλος *d*, την τιμή της μεταβλητής *a*. Στη *mulT*.*leftArgKind* δίνουμε τιμή `'d'` που σημαίνει ότι στη *mulT*.*leftArg* έχουμε σε χρήση το μέλος *d*.

Ο υπολογισμός της παράστασης γίνεται με την κλήση:

```
   . . . eval( &root ) . . .
```

προς την απλή αναδρομική συνάρτηση:

```
double eval( Node* r )
{
   double argL, argR, fv;

   argL = (r->leftArgKind == 'd') ?
             r->leftArg.d : eval(r->leftArg.p);
   argR = (r->rightArgKind == 'd') ?
             r->rightArg.d : eval(r->rightArg.p);
   switch ( r->oper )
   {
      case '+':  fv = argL + argR;  break;
      case '-':  fv = argL - argR;  break;
      case '*':  fv = argL * argR;  break;
      case '/':  fv = argL / argR;  break;
//    default: throw . . .
   }
   return fv;
} // eval
```

15.10 Δομές για Εξαιρέσεις

Όταν γράφεις κάποιο πρόγραμμα –πάνω από δυο γραμμές– έχεις μια σιγουριά: θα έχει λάθη! Τα βρίσκεις βέβαια και τα διορθώνεις αλλά, όπως λέει και ο πιο γνωστός νόμος του Murphy για τον προγραμματισμό: *Υπάρχει πάντοτε ακόμη ένα λάθος*.[8] Το πρόγραμμα θα πρέπει να είναι προετοιμασμένο ώστε, «όταν χτυπήσει ο κεραυνός», να μας αναφέρει για κάθε λάθος:

- Πού έγινε το λάθος.
- Τι είδους λάθος ήταν.
- Πώς προκλήθηκε.

Αυτές τις πληροφορίες θα πρέπει να μεταφέρουν οι εξαιρέσεις που ρίχνουμε. Θα ξαναγράψουμε τα παραδείγματα που δώσαμε στην §14.9 ορίζοντας όμως έναν τύπο εξαιρέσεων που μας επιτρέπει να μεταβιβάσουμε αυτές τις πληροφορίες:

```
struct ApplicXptn
{
   enum { domainErr, paramErr };
   char   funcName[100];
   int    errCode;
   double errDblVal1, errDblVal2, errDblVal3;

   ApplicXptn( const char* fn, int ec,
               double dv1, double dv2 = 0, double dv3 = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errDblVal1 = dv1;  errDblVal2 = dv2;  errDblVal3 = dv3;  }
}; // ApplicXptn
```

Αυτή είναι μια **δομή** (ή **κλάση**) **εξαιρέσεων**. Πρόσεξε τα μέλη της:

- `char funcName[100]`: εδώ θα βάζουμε το *όνομα της συνάρτησης* που ρίχνει την εξαίρεση.
- `int errCode`: εδώ θα βάζουμε έναν κωδικό που θα δείχνει το *είδος του λάθους* που παρουσιάστηκε. Οι κωδικοί λάθους απαριθμούνται στην αρχή. Στην περίπτωσή μας έχουμε δύο (*domainErr* = 0, *paramError* = 1). Χρησιμοποιούνται και στην έγερση και στη σύλληψη.
- `double errDblVal1, errDblVal2, errDblVal3`: εδώ θα βάζουμε *τις τιμές ή την τιμή* που προκάλεσαν το πρόβλημα.

[8] «There is always one more bug.»

Τέλος βλέπουμε:

- Έναν δημιουργό. Αυτός χρησιμοποιείται για την έγερση των εξαιρέσεων: δημιουργεί το αντικείμενο της εξαίρεσης. Μπορεί να χρειαστεί να γράψουμε και άλλους δημιουργούς.

Δες λοιπόν πώς μπορούμε να γράψουμε τις δύο συναρτήσεις:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
      throw ApplicXptn( "v", ApplicXptn::domainErr, x );
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Όπως βλέπεις, στη `throw` καλούμε τον δημιουργό της δομής για να δημιουργήσει το αντικείμενο της εξαίρεσης. Τι του δίνουμε:

- `"v"`: το όνομα της συνάρτησης.
- `ApplicXptn::domainErr`: τον κωδικό σφάλματος. Περίεργος συμβολισμός; Τον έχουμε ξαναδεί· στην §1.2 είχαμε δει τα `std::cout`, `std::endl`. Εδώ εννοούμε: το `domainErr` που ορίσαμε στον ονοματοχώρο `ApplicXptn`.[9]
- `x`: την τιμή που προκάλεσε το πρόβλημα.

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
      throw ApplicXptn( "pqer", ApplicXptn::paramErr, x, y, z );
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Πρόσεξε ότι εδώ περνούμε τρεις τιμές (x, y, z) στη συνάρτηση.

Τώρα έχουμε μια `catch` εκτός από την "`catch(...)`":

```
int main()
{
   double t, u;

   try
   {
   // . . .
      pqer( 1, 2, 3, t, u );
   // . . .
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   // . . .
   }
   catch ( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::domainErr:
            cout << " η " << x.funcName << " δεν ορίζεται στο "
                 << x.errDblVal1 << endl;
            break;
         case ApplicXptn::paramErr:
            cout << " η " << x.funcName
                 << " κλήθηκε με παραμέτρους " << x.errDblVal1
```

[9] Όπως θα μάθουμε αργότερα, κάθε δομή (κλάση) ορίζει τον δικό της ονοματοχώρο.

```
                << ", " << x.errDblVal2 << ", " << x.errDblVal2
                << endl;
            break;
         default:
            cout << " μη αναμενόμενη εξαίρεση ApplicXptn από "
                << x.funcName << endl;
      } // switch
   }
   catch( ... )
   {
      cout << " μη αναμενόμενη άγνωστη εξαίρεση" << endl;
   }
} // main
```

Ως δεύτερο παράδειγμα ας ξαναγράψουμε το πρόγραμμα με τους συνδυασμούς που είδαμε παραπάνω:

```
#include <iostream>

using namespace std;

struct ApplicXptn
{
   char funcName[100];
   int  errCode;
   int  iErrVal1;
   int  iErrVal2;

   enum { paramError, factOfNeg };
   ApplicXptn ( const char* fn, int ec, int iv1=0, int iv2=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      iErrVal1 = iv1;  iErrVal2 = iv2;  }
}; // ApplicXptn

   unsigned long int factorial( int n );
   unsigned long int comb( int m, int n );

int main()
{
   int m, n;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   try
   {
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb( m, n ) << endl;
   }
   catch ( ApplicXptn x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::paramError:
            cout << "Η " << x.funcName
                 << " κλήθηκε με λάθος παραμέτρους ( "
                 << x.iErrVal1 << " < " << x.iErrVal2 << " )"
                 << endl;
            break;
         case ApplicXptn::factOfNeg:
            cout << "Η " << x.funcName
                 << " κλήθηκε με αρνητικό όρισμα ( "
                 << x.iErrVal1 << " )" << endl;
            break;
         default:
            cout << "Μη αναμενόμενη εξαίρεση από την "
                 << x.funcName << " τιμές: " << x.iErrVal1 << ", "
                 << x.iErrVal2 << endl;
```

```
      } // switch
   } // catch
} // main

   // natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
   unsigned long int natProduct( int m, int n )
   {
      if ( m <= 0 || n < m )
         throw ApplicXptn( "natProduct",
                           ApplicXptn::paramError, m, n );
   // 0 < m <= n
      unsigned long int fv(m);
      for ( int k(m+1); k <= n; ++k )  fv *= k;
      return fv;
   } // natProduct

   // factorial -- Υπολογίζει το n!
   unsigned long int factorial( int n )
   {
      try
      {
         return ( n == 0 ) ? 1 : natProduct( 1, n );
      }
      catch( ApplicXptn x )
      {
         throw ApplicXptn( "factorial",
                           ApplicXptn::factOfNeg, n );
      }
   } // factorial

   // comb -- Υπολογίζει τους συνδυασμούς των m ανά n
   unsigned long int comb( int m, int n )
   {
      unsigned long int fv;

      if ( n < m-n )  fv = natProduct(m-n+1,m)/factorial(n);
                else  fv = natProduct(n+1,m)/factorial(m-n);
      return fv;
   } // comb
```

Εδώ πρόσεξε τα εξής:

1. Η δομή εξαιρέσεων κρατάει δύο «προβληματικές τιμές» διότι για ένα πρόβλημα που παρουσιάζεται στην *natProduct()* θα πρέπει να δώσουμε τις τιμές των *m* και *n*.
2. Αν προσπαθήσουμε να υπολογίσουμε το παραγοντικό ενός αρνητικού αριθμού, το να πιάσουμε μια εξαίρεση που θα έλθει από την *natProduct()* είναι μάλλον παραπλανητικό, Γι' αυτό στη *factorial()* πιάνουμε την εξαίρεση που έρχεται από την *natProduct()* και ρίχνουμε άλλη εξαίρεση με ακριβέστερη πληροφορία.

Τέτοιους τύπους εξαιρέσεων ορίζουμε για το πρόγραμμα, για μια βιβλιοθήκη συναρτήσεων κλπ. Έτσι, σε ένα πρόγραμμα μπορεί να ρίχνονται εξαιρέσεις διαφόρων τύπων και φυσικά θα πρέπει να έχουμε και τις κατάλληλες `catch`.

Αργότερα θα δούμε τις κλάσεις εξαιρέσεων πιο εκτεταμένα.

15.11 Μη Μορφοποιημένα Αρχεία

Μέχρι τώρα διαβάζουμε και γράφουμε μορφοποιημένα αρχεία, δηλαδή αρχεία με γραμμές, με (αναγνώσιμους) χαρακτήρες. Έχουμε πάντως τη δυνατότητα να γράφουμε σε αρχεία τα στοιχεία όπως είναι μέσα στη μνήμη του υπολογιστή, στην εσωτερική παράσταση.

Ξαναγράφουμε το πρώτο πρόγραμμα της §8.6 με κάποιες αλλαγές:

- Στο γράψιμο των τιμών στο αρχείο:

```
   double sum( 0 );
```

```
int    n( 0 );
ofstream a( "fltnum.dta", ios_base::binary );
cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
while ( x != sentinel )
{
// γράψε στο αρχείο την τιμή που πήρες
   a.write( reinterpret_cast<const char*>(&x), sizeof(x) );
   ++n;
   sum += x;
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
} // while
a.close();
cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
```

- Στο διάβασμα των τιμών από το αρχείο:

```
if ( n > 0 )
{
   double avrg( sum/n );
   cout << " ΑΘΡΟΙΣΜΑ = " << sum
        << "  <x> = " << avrg << endl;
   ifstream b( "fltnum.dta", ios_base::binary );
   int m( 0 );
   b.read( reinterpret_cast<char*>(&x), sizeof(x) );
   while ( !b.eof() )
   {
      ++m;
      y = exp( (avrg-x)/avrg );
      cout << " x[";
      cout.width(3);  cout << m << "] = ";
      cout.width(6);  cout << x << "   y[";
      cout.width(3);  cout << m << "] = " ;
      cout.width(6);  cout << y << endl;
      b.read( reinterpret_cast<char*>(&x), sizeof(x) );
   } // while
   b.close();
}
```

Τώρα, το βοηθητικό αρχείο **fltnum.dta** είναι **μη μορφοποιημένο** ή **δυαδικό** (binary, unformatted). Όταν δημιουργείς ή ετοιμάζεσαι να διαβάσεις ένα μη μορφοποιημένο αρχείο, πρέπει να βάλεις μια ακόμη παράμετρο. Ανοίγουμε δηλώνοντας το ρεύμα *a* με την:

```
ofstream a( "fltnum.dta", ios_base::binary );
```

και δείχνουμε ότι το αρχείο **fltnum.dta**, που θα δημιουργηθεί, θα είναι μη μορφοποιημένο. Θα μπορούσαμε, αν θέλαμε, να δηλώσουμε:

```
ofstream a;
```

και να ανοίξουμε με την:

```
a.open( "fltnum.dta", ios_base::binary );
```

Παρομοίως, ανοίγουμε δηλώνοντας το *b* με την:

```
   ifstream b( "fltnum.dta", ios_base::binary );
```

και δείχνουμε ότι το αρχείο **fltnum.dta** που θα διαβάσουμε είναι μη μορφοποιημένο. Και εδώ, θα μπορούσαμε, εναλλακτικώς, να δηλώσουμε:

```
ifstream b;
```

και να ανοίξουμε με τη:

```
   b.open( "fltnum.dta", ios_base::binary );
```

Αν μετατρέπαμε το δεύτερο πρόγραμμα, όπου χρησιμοποιούμε ρεύμα

```
fstream a;
```

θα ανοίγαμε το ρεύμα *a* με την:

```
a.open( "fltnum.txt",
        ios_base::in|ios_base::out|ios_base::binary );
```

Πώς γράφουμε σε ένα μη μορφοποιημένο αρχείο;

Η κλάση *ofstream* έχει τη μέθοδο *write()*, που περιμένει δύο ορίσματα:

- Το πρώτο είναι ένα βέλος, *p*, προς μεταβλητή τύπου `char` (ακριβέστερα: `const char*`).
- Το δεύτερο είναι ένας φυσικός *n*.

Αν δώσουμε `a.write( p, n )` εννοούμε:

- Στείλε στο ρεύμα *a*, *n* ψηφιολέξεις,
- ξεκινώντας από τη θέση μνήμης που δείχνει το *p*.

Στο παράδειγμά μας, έχουμε να γράψουμε στο αρχείο την τιμή της *x*. Θα μπορούσαμε να γράψουμε:

```
a.write( &x, sizeof(x) );
```

Όχι ακριβώς έτσι. Πράγματι, το βέλος `&x` δείχνει τη θέση μνήμης που αρχίζει η αποθήκευση των στοιχείων που θέλουμε να γράψουμε και η `sizeof(x)` μας δίνει τον αριθμό των ψηφιολέξεων. Το πρόβλημα είναι με τον τύπο του `&x`: είναι τύπου `double*` (βέλος προς μεταβλητή τύπου `double`) ενώ η *write()* περιμένει τιμή τύπου `const char*`. Εδώ χρειαζόμαστε την *ερμηνευτική τυποθεώρηση* που είδαμε πιο πριν. Γράφουμε λοιπόν:

```
a.write( reinterpret_cast<const char*>(&x), sizeof(x) );
```

Για διάβασμα, η κλάση *ifstream* έχει τη μέθοδο *read()*, που περιμένει δύο ορίσματα, σαν αυτά της *write()*. Αν δώσουμε `b.read(p, n)` εννοούμε: Πάρε από το ρεύμα *b*, *n* ψηφιολέξεις και αποθήκευσέ τις στη μνήμη, ξεκινώντας από τη θέση μνήμης που δείχνει το *p*. Εδώ, η *p* πρέπει να είναι τύπου `char*`. Έτσι, βλέπεις να διαβάζουμε με την:

```
b.read( reinterpret_cast<char*>(&x), sizeof(x) );
```

Η κλάση *fstream* έχει και τις δύο μεθόδους *write()* και *read()*.

Τα μη μορφοποιημένα αρχεία έχουν το πλεονέκτημα, σε σχέση με τα μορφοποιημένα, ότι

- επιτρέπουν μεγαλύτερη ταχύτητα στην επεξεργασία τους, διότι δεν καταναλίσκεται υπολογιστικός χρόνος για μετατροπές από την εσωτερική παράσταση σε χαρακτήρες (**μορφοποίηση**, formatting) –όταν γράφουμε– και από χαρακτήρες σε εσωτερική παράσταση –όταν διαβάζουμε.

Έχουν όμως και μειονεκτήματα:

- επειδή είναι γραμμένα σε εσωτερική παράσταση, πολλές φορές δεν είναι εύκολο να τα διαχειριστείς με προγράμματα που έχουν αναπτυχθεί σε διαφορετικά προγραμματιστικά περιβάλλοντα,
- δεν μπορείς να τα διαχειριστείς με κειμενογράφους.

Παρατήρηση: ▶
Να πούμε δυο λόγια παραπάνω για το τελευταίο «μειονέκτημα»: Το να διαχειρίζεσαι ένα αρχείο με κειμενογράφο σημαίνει και ότι μπορεί να εισαγάγεις σφάλματα. Το να διαχειρίζεσαι ένα αρχείο με ειδικό πρόγραμμα –όπως πρέπει να γίνει για ένα μη μορφοποιημένο αρχείο– σημαίνει ότι οι εισαγόμενες τιμές θα περνούν και κάποιους ελέγχους. Γι' αυτό όταν διαβάζουμε δεδομένα από μορφοποιημένο αρχείο κάνουμε όλους τους δυνατούς ελέγχους (σαν να διαβάζουμε από το πληκτρολόγιο). Αυτό δεν σημαίνει ότι δεν κάνουμε ελέγχους σε δεδομένα που διαβάζουμε από μη μορφοποιημένο αρχείο. ◀

Αν το σκεφτείς λιγάκι, το παράδειγμα που δώσαμε είναι από τα λίγα που σίγουρα συμφέρει να χρησιμοποιήσουμε μη μορφοποιημένο αρχείο. Σε άλλες περιπτώσεις υπερισχύουν τα μειονεκτήματα και αυτός είναι ο λόγος που αργήσαμε να τα παρουσιάσουμε. Στην επόμενη παράγραφο όμως θα δούμε και κάτι άλλο που τα κάνει ενδιαφέροντα.

15.12 Τυχαία Πρόσβαση σε Αρχεία – Μέθοδοι *seek* και *tell*

Σε ένα μη μορφοποιημένο αρχείο μπορούμε να γράφουμε τιμές διαφόρων τύπων. Αν όμως γράψουμε στοιχεία του ίδιου τύπου τότε όλες οι συνιστώσες ή **εγγραφές** (records) του αρχείου έχουν το ίδιο μήκος. Έτσι, στο παράδειγμα της προηγούμενης παραγράφου, αν, ας πούμε, το `sizeof(double)` είναι 8 ψηφιολέξεις, ξέρουμε ότι η εγγραφή 0 του αρχείου καταλαμβάνει τις ψηφιολέξεις από 0 μέχρι και 7, η εγγραφή 1 τις ψηφιολέξεις από 8 μέχρι και 15, η εγγραφή *k* τις ψηφιολέξεις από *k*·8 μέχρι και (*k*+1)·8–1. Αν λοιπόν μας δοθεί η δυνατότητα να διαβάζουμε ή να γράφουμε ξεκινώντας από οποιαδήποτε ψηφιολέξη του αρχείου, έχουμε τη δυνατότητα πρόσβασης σε οποιαδήποτε εγγραφή με την ίδια ευκολία· έχουμε, όπως λέμε, ένα **αρχείο τυχαίας πρόσβασης** (random access file). Στη C++ η τυχαία πρόσβαση επιτυγχάνεται με τις μεθόδους *seekg()* και *seekp()*.

Στο Κεφ. 8 είδαμε τη μέθοδο *seekg()*, και ειδικά τη χρήση `a.seekg(0)` που μας επιτρέπει την πρόσβαση στην αρχή ενός αρχείου συνδεμένου με το ρεύμα *a*. Τώρα θα δούμε τη χρήση της μεθόδου *seekg()* –των κλάσεων *ifstream*, και *fstream*– γενικότερα.

Αν έχουμε ένα εισερχόμενο ρεύμα *a*, συνδεμένο με κάποιο αρχείο, εκτέλεση της `a.seekg(n)`, όπου *n* φυσικός αριθμός, έχει ως αποτέλεσμα, η επόμενη εντολή εισόδου που θα εκτελεσθεί, να ξεκινήσει από την *n* ψηφιολέξη του αρχείου.

Όπως είδαμε παραπάνω

- ***Οι ψηφιολέξεις στο αρχείο αριθμούνται ξεκινώντας από 0 (μηδέν).***

Κατ' αρχάς να τελειώνουμε με το εξής: *Μπορούμε να χρησιμοποιούμε τη seekg() σε αρχεία text αλλά δεν έχει και τόσο νόημα*. Δες το παρακάτω

Παράδειγμα

Έστω τώρα ότι έχουμε ένα αρχείο –με όνομα `ranfl.txt`– και περιεχόμενο:

```
45␣␣␣␣89␣␣␣54
␣␣␣␣-3␣␣␣4.67␣␣␣dagdash␣␣␣jkfs␣jfejk
␣␣␣1
```

και το πρόγραμμα:

```
ifstream f( "ranfl.txt" );
double x;

for ( int k(0); k <= 12; k += 2 )
{
   f.seekg( k );  f >> x;
   cout << x << endl;
}
```

Τι θα πάρουμε από την εκτέλεσή του; Αυτά:

```
45
89
89
89
54
54
4
```

- Την πρώτη φορά έχουμε: `f.seekg(0); f >> x`, δηλαδή πήγαινε στην ψηφιολέξη 0 του αρχείου και διάβασε έναν πραγματικό· ο πρώτος αριθμός που διαβάζεται είναι ο 45.
- Την επόμενη φορά εκτελούνται οι `f.seekg(2); f >> x`, δηλαδή πήγαινε στην ψηφιολέξη 2 (το πρώτο διάστημα μετά το "45") του αρχείου και διάβασε έναν πραγματικό· τώρα, ο αριθμός που διαβάζεται είναι ο 89.
- Στη συνέχεια εκτελούνται οι `f.seekg(4); f >> x`, δηλαδή *επίστρεψε στο τρίτο διάστημα μετά το* "45" και διάβασε έναν πραγματικό· και πάλι διαβάζεται ο 89 που θα διαβαστεί και τρίτη φορά με την ανάγνωση που ακολουθεί την `f.seekg(6)`.

- Την πέμπτη και την έκτη φορά η *k* έχει τιμή 8 και 10 αντιστοίχως –έχει ξεπερασθεί το 89– και οι `f.seekg(k); f >> x` θα διαβάσουν το 54.
- Την τελευταία φορά η *k* έχει τιμή 12 και μετά την εκτέλεση της `f.seekg(k)`, το διάβασμα θα ξεκινήσει από το "4" του "54".

☜☜☜

Όπως βλέπεις, το πρόβλημα ξεκινάει από τον τρόπο που είναι γραμμένο ένα μορφοποιημένο αρχείο (text) σε αντιδιαστολή με ένα μη μορφοποιημένο που έχει (συνήθως) εγγραφές σταθερού μήκους.

Πάντως υπάρχουν δύο περιπτώσεις που χρησιμοποιούμε τη *seekg()* και σε μορφοποιημένα αρχεία:

- Με την `f.seekg(0)` πηγαίνουμε στην αρχή του αρχείου.
- Με την `f.seekg(0, ios_base::end)` πηγαίνουμε στο τέλος του αρχείου, όπως θα δούμε στη συνέχεια.

Οι κλάσεις *ofstream* και *fstream* έχουν τη μέθοδο *seekp*, για εξερχόμενα ρεύματα, με την οποίαν *καθορίζουμε την ψηφιολέξη του αρχείου από όπου θα αρχίσει το γράψιμο*.

Οι μέθοδοι *seekg()* και *seekp* είναι χρήσιμες σε μη μορφοποιημένα αρχεία με εγγραφές ίσου μήκους.

Έστω *x* μια μεταβλητή τύπου *T* και ένα ρεύμα `fstream a`, συνδεμένο με ένα αρχείο με τιμές τύπου *T*. Ανοίγουμε το ρεύμα, για διάβασμα και γράψιμο με την:

```
a.open( "...", ios_base::in|ios_base::out|ios_base::binary );
```

Οι εντολές

```
a.seekg( n*sizeof(T) );
a.read( reinterpret_cast<char*>(&x), sizeof(T) );
```

όπου *n* μη αρνητικός ακέραιος, ζητούν τα εξής:

- «να τοποθετηθεί η κεφαλή ανάγνωσης/εγγραφής» πριν από την υπ' αριθμό *n·sizeof*(*T*) ψηφιολέξη του αρχείου και
- η επόμενη *read* να ξεκινήσει από το σημείο αυτό, δηλαδή στην *x* θα αποθηκευτεί η υπ' αριθμόν *n* τιμή του αρχείου.

Παρομοίως, αν θέλεις να γράψεις το περιεχόμενο της *x* στην υπ' αριθμό *n* θέση του αρχείου θα πρέπει να δώσεις τις εντολές:

```
a.seekp( n*sizeof(T) );
a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

Χρησιμοποιώντας λοιπόν τις *seekg()* και *seekp()* μπορείς να διαβάζεις ή να γράφεις το ίδιο εύκολα οποιαδήποτε τιμή του αρχείου.

Αν λοιπόν έχεις να ενημερώσεις το περιεχόμενο κάποιας εγγραφής αυτό γίνεται ως εξής:

```
a.seekg( n*sizeof(T) );
a.read( reinterpret_cast<char*>(&x), sizeof(T) );
// Εντολές ενημέρωσης της x
a.seekp( n*sizeof(T) );
a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

«Δίδυμες» των *seekg()* και *seekp()* είναι οι *tellg()* και *tellp()*. Αν δώσεις

```
k = a.tellg();
```

όπου *a* ρεύμα ανοιγμένο για διάβασμα, στην *k* (`long`) θα έλθει ως τιμή η θέση από την οποία θα γίνει η επόμενη ανάγνωση. Παρομοίως, η:

```
k = a.tellp();
```

όπου *a* ρεύμα ανοιγμένο για γράψιμο, στην *k* (`long`) θα έλθει ως τιμή η θέση από την οποία θα ξεκινήσει η επόμενη εγγραφή.

15.12.1 Πώς Βρίσκουμε το Μέγεθος Αρχείου

Με χρήση των *seekg()* και *tellg()* μπορείς να βρεις το μέγεθος ενός αρχείου σε ψηφιολέξεις. Αν έχεις ανοίξει ένα ρεύμα με `ios_base::binary` για διάβασμα από το αρχείο που σε ενδιαφέρει τότε οι:

```
a.seekg( 0, ios_base::end );
k = a.tellg();
```

θα κάνουν τα εξής: Η πρώτη θα «πάει την κεφαλή ανάγνωσης στο τέλος του αρχείου» και η δεύτερη θα μας δώσει τη θέση της, που θα είναι ακριβώς το μέγεθος του αρχείου.

Γενικώς, η `a.seekg(n, ios_base::end)` ζητάει: η επόμενη ανάγνωση να ξεκινήσει *n* ψηφιολέξεις μετρώντας από το *τέλος του αρχείου*.

Παρατηρήσεις: ▶

Αν αυτή η δουλειά είναι η πρώτη που κάνεις μετά το άνοιγμα του αρχείου μπορείς να την κάνεις και ως εξής:

```
a.open( "myFileName.dta",
        ios_base::in|ios_base::binary|ios_base::ate );
k = a.tellg();
```

Θυμίσου (§8.12) ότι βάζοντας στην *open()* το `ios_base::ate` «*με το άνοιγμα του αρχείου τοποθετείται στο τέλος του.*» ◀

Παράδειγμα 🡆

Πολύ συχνά, όταν θέλουμε να επεξεργαστούμε κάποιο αρχείο text που δεν είναι πολύ μεγάλο, συμφέρει να το αντιγράψουμε ολόκληρο στην κύρια μνήμη του υπολογιστή, σε έναν πίνακα και να κάνουμε το πρόγραμμά μας πιο γρήγορο και πιο απλό. Να πώς μπορεί να γίνει αυτό με όσα μάθαμε στην παράγραφο αυτή:

```
ifstream a( "abc.txt",
            ios_base::in|ios_base::binary|ios_base::ate );
char t[16000];
long flSize;

// a.seekg( 0, ios_base::end ); αν δεν έχεις ios_base::ate
flSize = a.tellg();
if ( flSize <= 16000 )
{
   a.seekg( 0 );
   a.read( t, flSize );
}
a.close();
```

Παρατηρήσεις: ▶

1. Πρόσεξε το "`ios_base::binary`": είναι απαραίτητο για να φορτωθεί σωστά το αρχείο. Γενικώς: δίνεις *read()* ή *write()* για αρχείο που το έχεις ανοίξει με αυτό το χαρακτηριστικό.
2. Προσοχή στις αλλαγές γραμμών που, πιθανότατα, υπάρχουν στο αρχείο. Μπορεί να είναι **<cr><lf>** (στη C++:`'\r''\n'`), μπορεί να είναι κάτι άλλο![10]
3. Μη φανταστείς ότι, όταν φορτώσεις το αρχείο, θα μπει αυτομάτως κάποιο `'\0'` στο τέλος. Αν το χρειάζεσαι θα το βάλεις εσύ (`t[flSize] = '\0'`).
4. Αργότερα θα μάθουμε πώς να παίρνουμε ακριβώς όση μνήμη χρειαζόμαστε για το περιεχόμενο του αρχείου. ◀

🡆🡆🡆

Αν έχουμε ένα μη μορφοποιημένο αρχείο με τιμές τύπου *T* μπορούμε να υπολογίσουμε πόσες τιμές περιέχει[11]:

---

[10] Αν δεν βάλεις το "`ios_base::binary`" είναι πιθανό ότι ο μεταγλωττιστής θα βάλει τις εντολές που θα αλλάζουν (κατά την ανάγνωση) όλα τα `'\r''\n'` σε `'\n'`. Δεν το θέλεις όμως αυτό…

[11] Με την προϋπόθεση ότι η τιμές έχουν το ίδιο μέγεθος. Θα δεις στο επόμενο κεφάλαιο ότι αυτό δεν ισχύει πάντοτε.

- Βρίσκουμε το μέγεθος του αρχείου (όπως το *flSize* στο προηγούμενο πρόγραμμα) και
- Το διαιρούμε με το *μέγεθος φύλαξης ενός στοιχείου* τύπου *T* (`sizeof(T)` ή κάτι σχετικό).

15.13 Τιμή-Δομή σε Μη Μορφοποιημένο Αρχείο

Ενώ, όπως είδαμε, ένα μη μορφοποιημένο αρχείο με τιμές τύπου `int` ή `double` είναι περιορισμένης χρησιμότητας, ένα μη μορφοποιημένο αρχείο με τιμές τύπου `struct` (σταθερού μεγέθους) έχει ενδιαφέρον.

Ένα τέτοιο αρχείο έχει ανάγκη για πολλές ενημερώσεις. Όπως είδαμε παραπάνω, σε ένα τέτοιο αρχείο έχουμε τη δυνατότητα να κάνουμε ενημέρωση της κάθε τιμής **επι τόπου** (in situ, in place):

```
   a.seekg( n*sizeof(T) );
   a.read( reinterpret_cast<char*>(&x), sizeof(T) );
// Εντολές ενημέρωσης της x
   a.seekp( n*sizeof(T) );
   a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

και όχι με αντιγραφή σε άλλο αρχείο όπως γίνεται στα σειριακά αρχεία.

Με ένα τέτοιο μη μορφοποιημένο αρχείο το πρόβλημα που υπάρχει πάντοτε είναι αυτό της διαχείρισης με προγράμματα που έχουν αναπτυχθεί σε διαφορετικά περιβάλλοντα ανάπτυξης. Έτσι, το πρώτο πρόβλημα που μπορεί να προκύψει ξεκινάει από το ότι το `sizeof(T)` –πιθανότατα– δεν θα είναι παντού ίδιο. Αυτό θα προσπαθήσουμε να το ξεπεράσουμε ως εξής: αντί να φυλάγουμε μια τιμή-δομή με μια εντολή όπως η

```
   a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

θα γράψουμε μια συνάρτηση που θα τη φυλάγει μέλος προς μέλος. Π.χ. για την

```
struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address
```

θα γράψουμε μια:

```
void address_save( const Address& a, ostream& bout )
{
   bout.write( a.country, sizeof(a.country) );
   bout.write( a.city, sizeof(a.city) );
   bout.write( reinterpret_cast<const char*>(&a.areaCode),
               sizeof(a.areaCode) );
   bout.write( a.street, sizeof(a.street) );
   bout.write( reinterpret_cast<const char*>(&a.number),
               sizeof(a.number) );
}; // Address_save
```

Τώρα όμως στη *seekg()* και τη *seekp()* δεν θα βάζουμε το `sizeof(T)` αλλά το άθροισμα των μεγεθών φύλαξης των μελών. Θα μπορούσαμε να ορίσουμε την *Address* ως εξής:

```
struct Address
{
   enum { countrySz = 17, citySz = 17, streetSz = 17,
          saveSize = countrySz + citySz + sizeof(int) +
                     streetSz + sizeof(int) };
   char country[countrySz];
   char city[citySz];
   int  areaCode;
   char street[streetSz];
   int  number;
}; // Address
```

και να χρησιμοποιούμε τη *saveSize*:

```
   a.seekg( n*Address::saveSize );
   address_load( x, a );
// Εντολές ενημέρωσης της x
   a.seekp( n*Address::saveSize );
   address_save( x, a );
```

Παρατηρήσεις: ▶

1. Να τονίσουμε ότι το ρεύμα *a* έχει ανοιχτεί ως:

```
   fstream a( "...",
              ios_base::in|ios_base::out|ios_base::binary );
```

2. Πώς θα γράψουμε την *address_load()*; Με βάση την αρχή «διαβάζουμε όπως γράφουμε»:

```
void address_load( Address& a, istream& bin )
{
   bin.read( a.country, sizeof(a.country) );
   bin.read( a.city, sizeof(a.city) );
   bin.read( reinterpret_cast<char*>(&a.areaCode),
             sizeof(a.areaCode) );
   bin.read( a.street, sizeof(a.street) );
   bin.read( reinterpret_cast<char*>(&a.number),
             sizeof(a.number) );
}; // Address_load
```

Όπως βλέπεις υπάρχει αντιστοιχία 1-1 των *read()* της *address_load()* με τις *write()* της *address_save()*.

3. Για να βρούμε το πλήθος των εγγραφών στο αρχείο θα πρέπει να διαιρούμε δια *Address*`::`*saveSize* και όχι δια `sizeof(`*Address*`)`.

4. Λύσαμε όλα τα προβλήματα μεταφοράς του αρχείου από το ένα περιβάλλον σε οποιοδήποτε άλλο; Όχι, αλλά σε πολλές περιπτώσεις θα δουλεύει.◀

15.13.1 * Να Προτιμήσουμε τον Τύπο *string*;

Το ότι ορίσαμε τις σταθερές *countrySz*, *citySz*, *streetSz* μας απαλλάσσει και από τις τρεις «μαγικές σταθερές», τα μήκη των τριών πινάκων `char`. Μπορούμε όμως να τις «αξιοποιήσουμε» περισσότερο. Μπορούμε να χρησιμοποιήσουμε τον –πολύ πιο εύχρηστο– τύπο *string* για τα τρία μέλη και να τα μετατρέπουμε σε πίνακες μόνον για τη φύλαξη. Δηλαδή, ορίζουμε:

```
struct Address
{
   enum { countrySz = 17, citySz = 17, streetSz = 17,
          maxSize = 17,
          saveSize = countrySz + citySz + sizeof(int) +
                     streetSz + sizeof(int) };
   string country;
   string city;
   int    areaCode;
   string street;
   int    number;
}; // Address
```

Τώρα, η *address_save()* γίνεται:

```
void address_save( const Address& a, ostream& bout )
{
   char tmp[Address::maxSize];
   strncpy( tmp, a.country.c_str(), Address::countrySz-1 );
                                 tmp[Address::countrySz-1] = '\0';
   bout.write( tmp, Address::countrySz );
   strncpy( tmp, a.city.c_str(), Address::citySz-1 );
                                    tmp[Address::citySz-1] = '\0';
   bout.write( tmp, Address::citySz );
   bout.write( reinterpret_cast<const char*>(&a.areaCode),
```

```
                sizeof(a.areaCode) );
   strncpy( tmp, a.street.c_str(), Address::streetSz-1 );
                                   tmp[Address::streetSz-1] = '\0';
   bout.write( tmp, Address::streetSz );
   bout.write( reinterpret_cast<const char*>(&a.number),
                sizeof(a.number) );
}; // address_save
```

Πώς γίνεται η φύλαξη του *a.country*;

- Κατ' αρχάς θα φυλάξουμε *countrySz* ψηφιολέξεις το πολύ.
- Αντιγράφουμε *countrySz*–1 (το πολύ) χαρακτήρες στον tmp και σιγουρεύουμε ότι στο τέλος (`tmp[countrySz-1]`) θα υπάρχει ο φρουρός· στη συνέχεια θα καταλάβεις πού χρειάζεται.
- Φυλάγουμε τον tmp στο αρχείο (*countrySz* χαρακτήρες).

Δεν θα μπορούσαμε να φυλάξουμε *countrySz* χαρακτήρες κατ' ευθείαν το `a.country.c_str()`; Αν έχει μήκος μεγαλύτερο από *countrySz* δεν θα φυλαχθεί ο φρουρός.[12]

Με τον ίδιο τρόπο φυλάγουμε και τα μέλη *city*, *street*.

Η φόρτωση θα γίνει με την:

```
void address_load( Address& a, istream& bin )
{
   char tmp[Address::maxSize];
   bin.read( tmp, Address::countrySz );   a.country = tmp;
   bin.read( tmp, Address::citySz );   a.city = tmp;
   bin.read( reinterpret_cast<char*>(&a.areaCode),
             sizeof(a.areaCode) );
   bin.read( tmp, Address::streetSz );   a.street = tmp;
   bin.read( reinterpret_cast<char*>(&a.number),
             sizeof(a.number) );
}; // address_load
```

Για να εκτελεσθεί σωστά η `a.country = tmp` (και οι παρόμοιες) ο tmp θα πρέπει να έχει τον φρουρό (`'\0'`).

Δεν θα μπορούσαμε να φυλάγουμε τις τιμές των τριών μελών τύπου *string* με το ακριβές περιεχόμενο που έχουν κάθε φορά οποιοδήποτε μήκος και αν έχει; Ναι, αρκεί να φυλάγουμε και την τιμή του (κάθε) μήκους. Α, και κάτι άλλο: χάνουμε όλα εκείνα τα πλεονεκτήματα που λέγαμε ότι βασίζονται στο σταθερό μήκος εγγραφής· η διαχείριση γίνεται πολύπλοκη...

15.14 Ένα Παράδειγμα

Το πρόβλημα:

> *Μας δίνεται ένα αρχείο, με όνομα στο δίσκο* `elements.dta`*, που περιέχει πληροφορίες για τα χημικά στοιχεία. Το αρχείο έχει εγγραφές τύπου:*

```
struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn
```

> *που έχουν φυλαχθεί με τη συνάρτηση:*

```
void Elmn_save( const Elmn& a, ostream& bout )
{
   bout.write( reinterpret_cast<const char*>(&a.eANumber),
               sizeof(a.eANumber) );
   bout.write( reinterpret_cast<const char*>(&a.eAWeight),
```

[12] Μπορείς να σκεφτείς άλλη λύση του προβλήματος;

```
                sizeof(a.eAWeight) );
   bout.write( a.eSymbol, sizeof(a.eSymbol) );
   bout.write( a.eName, sizeof(a.eName) );
} // Elmn_save
```

Στην 1η εγγραφή υπάρχουν πληροφορίες για το Υδρογόνο (ατομικός αριθμός: 1), στη δεύτερη πληροφορίες για το Ήλιο (α.α.: 2) και γενικώς στην k-οστή εγγραφή πληροφορίες για το στοιχείο με α.α.= k.
Στο μέλος eName υπάρχει το όνομα του στοιχείου στα αγγλικά.
Θέλουμε να αντιγράψουμε το αρχείο σε ένα άλλο, με όνομα: **elementsGr.dta**, *που θα έχει εγγραφές του εξής τύπου:*

```
struct GrElmn
{
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[4];
   char               geName[14];
   char               geGrName[14];
}; // GrElmn
```

Στο μέλος geGrName θα βάλουμε το ελληνικό όνομα του στοιχείου. Η συμπλήρωση του νέου μέλους θα πρέπει να μπορεί να γίνεται τμηματικά, όποτε θέλει ο χρήστης.

Είναι φανερό ότι εδώ έχουμε δύο εντελώς ξεχωριστές δουλειές:

- Αντιγραφή του παλιού αρχείου στο νέο· αυτή η δουλειά θα γίνει μια φορά μόνο.
- Συμπλήρωση του ελληνικού ονόματος. Αυτή η δουλειά, όπως μας λέει και η διατύπωση του προβλήματος «*θα πρέπει να μπορεί να γίνεται τμηματικά, όποτε θέλει ο χρήστης.*»

Έχουμε λοιπόν να γράψουμε δύο προγράμματα.

15.14.1 Το Πρώτο Πρόγραμμα

Το πρώτο πρόγραμμα είναι απλό: Δουλεύουμε το αρχείο σειριακά, αφού θα πρέπει να το αντιγράψουμε ολόκληρο:

```
Άνοιξε ρεύμα bin από το παλιό αρχείο
Άνοιξε ρεύμα bout προς το νέο αρχείο
Διάβασε μια εγγραφή
while ( !bin.eof() )
{
   Αντίγραψε την εγγραφή
   Γράψε τη νέα εγγραφή
   Διάβασε την επόμενη εγγραφή
} // while
Κλείσε τα ρεύματα
```

Ανοίγουμε τα ρεύματα με τη δήλωσή τους και ελέγχουμε αν άνοιξαν:

```
      ifstream bin( "elements.dta", ios_base::binary );
      if ( bin.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotOpen,
                           "elements.dta" );
      ofstream bout( "elementsGr.dta", ios_base::binary );
      if ( bout.fail() )
      {
         bin.close();
         throw ApplicXptn( "main", ApplicXptn::cannotCreate,
                           "elementsGr.dta" );
      }
// ...
```

Πρόσεξε τι κάνουμε με το δεύτερο ρεύμα: Αν αποτύχει το άνοιγμα, πριν ρίξουμε τη σχετική εξαίρεση, *κλείνουμε το πρώτο ρεύμα που είναι ήδη ανοιχτό*!

Και τώρα το διάβασμα: Θα γράψουμε μια συνάρτηση *Elmn_load()* για να διαβάζουμε τα δεδομένα ενός στοιχείου με αποκλειστικό οδηγό την *Elmn_save()* («διαβάζουμε όπως γράψαμε»).

```
void Elmn_load( Elmn& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.eANumber),
             sizeof(a.eANumber) );
   bin.read( reinterpret_cast<char*>(&a.eAWeight),
             sizeof(a.eAWeight) );
   bin.read( a.eSymbol, sizeof(a.eSymbol) );
   bin.read( a.eName, sizeof(a.eName) );
   if ( bin.fail() && !bin.eof() )
      throw ApplicXptn( "Elmn_load", ApplicXptn::cannotRead );
} // Elmn_load
```

Πρόσεξε ότι εδώ ρίχνουμε και εξαίρεση αν δεν μπορέσουμε να διαβάσουμε από το ρεύμα εισόδου *bin* για οποιονδήποτε λόγο εκτός από "`bin.eof()`".

Αν λοιπόν έχουμε δηλώσει:

```
   Elmn oneElmn;
```

η «*Διάβασε μια εγγραφή*» γίνεται:

```
   Elmn_load( oneElmn, bin );
```

Πριν αρχίσουμε να ασχολούμαστε με τη *GrElmn* να την προσαρμόσουμε σε αυτά που είπαμε στην προηγούμενη παράγραφο:

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

Γυρνώντας στο σχέδιό μας, έχουμε να υλοποιήσουμε την «*Αντίγραψε την εγγραφή*». Μπορούμε να λύσουμε αυτό το πρόβλημα με μια συνάρτηση που θα τροφοδοτείται με ένα αντικείμενο τύπου *Elmn* και θα υπολογίζει και θα επιστρέφει ένα αντικείμενο *GrElmn*. Οι κανόνες μας λένε ότι πρέπει να γράψουμε συνάρτηση με τύπο:

```
GrElmn GrElmn_copyFromElmn( const Elmn& a )
{
   GrElmn fv;
   fv.geANumber = a.eANumber;
   fv.geAWeight = a.eAWeight;
   strcpy( fv.geSymbol, a.eSymbol );
   strcpy( fv.geName, a.eName );
   fv.geGrName[0] = '\0';
   return fv;
} // GrElmn_copyFromElmn
```

Αυτή συνάρτηση καλείται ως εξής:

```
         GrElmn oneGrElmn;
         oneGrElmn = GrElmn_copyFromElmn( oneElmn );
```

Αργότερα θα μάθουμε πώς μπορούμε να δώσουμε καλύτερη λύση.

Η «*Γράψε τη νέα εγγραφή*» υλοποιείται με την:

```
void GrElmn_save( const GrElmn& a, ostream& bout )
{
   bout.write( reinterpret_cast<const char*>(&a.geANumber),
               sizeof(a.geANumber) );
   bout.write( reinterpret_cast<const char*>(&a.geAWeight),
               sizeof(a.geAWeight) );
```

```
   bout.write( a.geSymbol, sizeof(a.geSymbol) );
   bout.write( a.geName, sizeof(a.geName) );
   bout.write( a.geGrName, sizeof(a.geGrName) );
   if ( bout.fail() )
      throw ApplicXptn( "GrElmn_save", ApplicXptn::cannotWrite );
} // GrElmn_save
```

Όπως βλέπεις, ρίχνουμε και εδώ εξαίρεση αν για κάποιον λόγο αποτύχει το γράψιμο. Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct ApplicXptn
{
   enum { cannotOpen, cannotCreate, cannotClose,
          cannotWrite, cannotRead };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ApplicXptn

struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn

void Elmn_load( Elmn& a, istream& bin );

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

void GrElmn_save( const GrElmn& a, ostream& bout );
GrElmn GrElmn_copyFromElmn( const Elmn& a );

int main()
{
   try
   {
   // Άνοιξε ρεύμα bin από το παλιό αρχείο
      ifstream bin( "elements.dta", ios_base::binary );
      if ( bin.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotOpen,
                           "elements.dta" );
   // Άνοιξε ρεύμα bout προς το νέο αρχείο
      ofstream bout( "elementsGr.dta", ios_base::binary );
      if ( bout.fail() )
      {
         bin.close();
```

```
         throw ApplicXptn( "main", ApplicXptn::cannotCreate,
                           "elementsGr.dta" );
      }
   // Διάβασε μια εγγραφή
      Elmn oneElmn;
      int k( 0 );
      Elmn_load( oneElmn, bin );
      while ( !bin.eof() )
      {
      // Αντίγραψε την εγγραφή
         GrElmn oneGrElmn;
         oneGrElmn = GrElmn_copyFromElmn( oneElmn );
      // Γράψε τη νέα εγγραφή
         GrElmn_save( oneGrElmn, bout );
         ++k;
      // Διάβασε την επόμενη εγγραφή
         Elmn_load( oneElmn, bin );
      }
   // Κλείσε τα ρεύματα
      bout.close();
      if ( bout.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           "elementsGr.dta" );
      bin.close();
      cout << "Αντίγραψα " << k << " εγγραφές" << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotCreate:
            cout << "cannot create file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "cannot close file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn ...
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Πρόσεξε πώς ορίζουμε την κλάση εξαιρέσεων, πώς ρίχνουμε εξαιρέσεις και πώς τις συλλαμβάνουμε.

Ειδικώς για τις περιπτώσεις ανοίγματος αρχείων, μπορείς να πεις ότι τα παρατάμε πολύ εύκολα. Θα μπορούσαμε, σε περίπτωση που κάτι δεν πάει καλά, να δώσουμε την ευκαι-

ρία στον χρήστη να κάνει κάτι. Δεν το κάνουμε διότι αυτό είναι ένα πρόγραμμα που θα δουλέψει μόνον μια φορά!

Θα πεις: «Πολύ κακό για το τίποτε! Θα μπορούσαμε να γράψουμε το πρόγραμμα σε δέκα γραμμές!» Σωστό! Το πρόγραμμα γράφτηκε έτσι για διδακτικούς λόγους (και είναι ένα καλό πρόγραμμα).

15.14.2 Το Δεύτερο Πρόγραμμα

Ας έρθουμε τώρα στο δεύτερο πρόγραμμα. Τώρα έχουμε να δουλέψουμε μόνο με ένα αρχείο, το **elementsGr.dta**, του οποίου τις εγγραφές ενημερώνουμε (διαβάζουμε – αλλάζουμε – ξαναγράφουμε). Δηλώνουμε λοιπόν:

```
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
```

και το ανοίγουμε ως εξής:

```
   bool ok;
   do {
      bInOut.open( flNm.c_str(),
                   ios_base::in|ios_base::out|ios_base::binary );
      if ( bInOut.fail() )
      {
         ok = false;
         cout << "cannot open " << flNm << endl;
         cout << "file name ('x' to exit): ";
         getline( cin, flNm, '\n' );
      }
      else
         ok = true;
   } while ( !ok && flNm != "x" && flNm != "X" );
```

Καλό είναι να βάλουμε τα παραπάνω σε μια συνάρτηση:

```
void openFile( string& flNm, fstream& bInOut )
{
   string prevFlNm;
   bool ok;
   do {
      bInOut.open( flNm.c_str(),
                   ios_base::in|ios_base::out|ios_base::binary );
      if ( bInOut.fail() )
      {
         ok = false;
         prevFlNm = flNm;
         cout << "cannot open " << flNm << endl;
         cout << "file name ('x' to exit): ";
         getline( cin, flNm, '\n' );
      }
      else
         ok = true;
   } while ( !ok && flNm != "x" && flNm != "X" );
   if ( !ok )
      throw ApplicXptn( "openFile", ApplicXptn::cannotOpen,
                        prevFlNm.c_str() );
} // openFile
```

Στην περίπτωση που ο χρήστης τα παρατήσει η *flNm* θα έχει τιμή **"x"** ή **"X"**. Φυσικά, ένα τέτοιο όνομα στην εξαίρεση δεν έχει νόημα. Για τον λόγο αυτόν φυλάγουμε στην *prevFlNm* το τελευταίο όνομα που δοκιμάσαμε στην *open()*.

Ας κάνουμε τώρα ένα σχέδιο για το πρόγραμμά μας. Μια απλή ιδέα είναι η εξής: ο χρήστης δίνει τον ατομικό αριθμό (α.α.) του στοιχείου. Ποιες είναι οι δεκτές τιμές για τον α.α.; Από 1 μέχρι το πλήθος των εγγραφών του αρχείου. Διαβάζουμε από το αρχείο την εγγραφή που αντιστοιχεί στο στοιχείο και την εμφανίζουμε στο χρήστη. Ο χρήστης μας δίνει το

ελληνικό όνομα που εισάγουμε στο μέλος *grName*. Τέλος, ξαναγράφουμε την εγγραφή στο αρχείο, στην παλιά της θέση. Αυτή η δουλειά θα γίνεται ξανά και ξανά, μέχρι... Μέχρι να τελειώσει τη δουλειά του ο χρήστης (ή να βαρεθεί). Αφού δεν υπάρχει στοιχείο με ατομικό αριθμό 0, ας κάνουμε τη συμφωνία: όταν δώσει 0 ως α.α. θα τελειώνουμε. Θα έχουμε δηλαδή επανάληψη με φρουρό.

Ωραίο σενάριο, ας το γράψουμε σε διακριτά βήματα:

```
Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
Διάβασε τον α.α.
while ( α.α. != 0 )
{
   Διάβασε από το αρχείο την αντίστοιχη εγγραφή
   Δείξε το περιεχόμενο της εγγραφής στο χρήστη
   Διάβασε το ελληνικό όνομα
   Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
   Διάβασε τον α.α.
}
```

Για την «*Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)*» γράφουμε τη συνάρτηση

```
void countRecords( fstream& bInOut, unsigned int& noOfRecords )
{
   unsigned long int initialPos;
   unsigned long int flSize;

   initialPos = bInOut.tellg();
   bInOut.seekg( 0, ios_base::end );  flSize = bInOut.tellg();
   noOfRecords = flSize / GrElmn::saveSize;
   bInOut.seekg( initialPos );
} // countRecords
```

δηλώνουμε μια μεταβλητή:

```
   unsigned int maxAtNo;
```

και αμέσως μετά το άνοιγμα του ρεύματος γράφουμε:

```
         countRecords( bInOut, maxAtNo );
```

Πρόσεξε τον ρόλο της μεταβλητής *initialPos*: φυλάγουμε την αρχική θέση του ρεύματος (`initialPos = bInOut.tellg()`) και στο τέλος, αφού υπολογίσουμε το πλήθος εγγραφών, το ξαναφέρνουμε στην αρχική του κατάσταση (`bInOut.seekg( initialPos )`).

Η «Διάβασε τον ατομικό αριθμό» μεταφράζεται εύκολα σε C++:

```
void readAtNo( int maxAtNo, int& aa )
{
   string aastr;

   do {
      cout << " Atomic Number (1.." << maxAtNo
           << ", 0 to exit): ";
      getline( cin, aastr, '\n' );
      aa = atol( aastr.c_str() );
   } while( aa < 0 || maxAtNo < aa );
} // readAtNo
```

Για τη «Διάβασε από το αρχείο την αντίστοιχη εγγραφή» θα γράψουμε μια συνάρτηση σαν την *Elmn_load()*, ας την πούμε *GrElmn_load*:

```
void GrElmn_load( GrElmn& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.geANumber),
             sizeof(a.geANumber) );
   bin.read( reinterpret_cast<char*>(&a.geAWeight),
             sizeof(a.geAWeight) );
   bin.read( a.geSymbol, sizeof(a.geSymbol) );
   bin.read( a.geName, sizeof(a.geName) );
   bin.read( a.geGrName, sizeof(a.geGrName) );
   if ( bin.fail() )
```

```
      throw ApplicXptn( "GrElmn_load", ApplicXptn::cannotRead );
} // GrElmn_load
```

Αλλά αυτή δεν είναι αρκετή: τώρα θα πρέπει να έχουμε τυχαία πρόσβαση, που θα γίνεται *με βάση τον ατομικό αριθμό που έδωσε ο χρήστης*. Και από ποια θέση θα πάμε να διαβάσουμε; Στην υπ' αριθμό 0 (μηδέν) εγγραφή του αρχείου βρίσκονται οι πληροφορίες για το Υδρογόνο, που έχει α.α. 1, στην υπ' αριθμό 1 εγγραφή βρίσκονται οι πληροφορίες για το Ήλιο, που έχει α.α. 2 και γενικώς: στην υπ' αριθμό *k*–1 εγγραφή υπάρχουν οι πληροφορίες για το στοιχείο που έχει α.α. *k*. Στην περίπτωσή μας, το μήκος της εγγραφής, σε ψηφιολέξεις, είναι *GrElmn*::*saveSize*· άρα η *aa*–1 εγγραφή, στην οποία υπάρχουν τα στοιχεία για το στοιχείο με α.α. *aa*, αρχίζει στην ψηφιολέξη `(`*aa*`-1)*`*GrElmn*`::`saveSize`.

Ας ξεχωρίσουμε τα προβλήματά μας: Ας γράψουμε πρώτα μια μέθοδο που διαβάζει μια εγγραφή παίρνοντας τον αριθμό της:

```
void readRandom( GrElmn& a, istream& bin, int atNo )
{
   if ( bin.fail() )
      throw ApplicXptn( "readRandom", ApplicXptn::streamNotOpen );
   if ( atNo <= 0 )
      throw ApplicXptn( "readRandom",
                        ApplicXptn::rangeErr, atNo );
   bin.seekg( (atNo-1)*GrElmn::saveSize );
   GrElmn_load( a, bin );
} // readRandom
```

Μέσα στη **main** βάζουμε τη δήλωση:

```
            GrElmn a;
```

και την κλήση:

```
            readRandom( a, bInOut, aa );
```

Για την υλοποίηση της «Δείξε το περιεχόμενο της εγγραφής στο χρήστη» γράφουμε μια συνάρτηση:

```
void GrElmn_display( const GrElmn& a, ostream& tout )
{
   tout << "atomic number: " << a.geANumber << endl
        << "atomic weight: " << a.geAWeight << endl
        << "symbol: " << a.geSymbol << endl
        << "name: " << a.geName << endl
        << "greek name: " << a.geGrName << endl;
} // GrElmn_display
```

Γιατί δείχνουμε το *geGrName*; Δεν θα είναι «κενό»; Μπορεί να είναι· αλλά αυτό το πρόγραμμα θα μπορεί να χρησιμοποιηθεί και για διόρθωση λαθεμένων τιμών, οπότε το παλιό όνομα χρειάζεται.

Η *GrElmn_display* καλείται από τη **main** ως εξής:

```
            GrElmn_display( a, cout );
```

Θα έλεγε κανείς ότι η «Διάβασε το ελληνικό όνομα» είναι απλή:

```
         cin >> newGrName;
         strcpy( a.geGrName, newGrName.c_str() );
```

όπου *newGrName* μεταβλητή τύπου *string*. Αλλά, αφού, όπως είπαμε, το πρόγραμμα θα μπορεί να χρησιμοποιηθεί και για διόρθωση τιμών, θα πρέπει να δίνουμε τη δυνατότητα στον χρήστη να αφήσει το ελληνικό όνομα αναλλοίωτο.

- Ας πούμε λοιπόν ότι αν ο χρήστης πιέσει απλώς το <enter> δεν θα αλλάξει το ελληνικό όνομα. Αυτό δεν μπορεί να γίνει με τη "`cin >> newName`", που, όπως ξέρεις, επιμένει να διαβάσει κάτι. Μπορεί να γίνει με τη "`getline( cin, newGrName, '\n' )`".
- Αν δεν δοθεί νέο ελληνικό όνομα δεν υπάρχει λόγος να ξαναφυλάξουμε την εγγραφή στο αρχείο.

Αλλάζουμε λοιπόν το σχέδιό μας· αντί για τις:

```
Διάβασε από το αρχείο την αντίστοιχη εγγραφή
Δείξε το περιεχόμενο της εγγραφής στο χρήστη
Διάβασε το ελληνικό όνομα
Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
```

θα έχουμε:

```
Διάβασε από το αρχείο την αντίστοιχη εγγραφή
Δείξε το περιεχόμενο της εγγραφής στο χρήστη
Διάβασε το ελληνικό όνομα
if ( άλλαξε το ελληνικό όνομα )
   Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
```

Ενσωματώνουμε τα παραπάνω στην:

```
void editGrName( fstream& bInOut, int atNo )
{
   GrElmn a;
   readRandom( a, bInOut, atNo );
   GrElmn_display( a, cout );
   string newGrName;
   cout << "new greek name: "; getline( cin, newGrName, '\n' );
   if ( !newGrName.empty() )
   {
      GrElmn_setGrName( a, newGrName );
      writeRandom( a, bInOut );
   }
} // editGrName
```

και τη χρησιμοποιούμε στη **main** ως εξής:

```
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
            editGrName( bInOut, aa );
```

Η *GrElmn_setGrName()* είναι απλή:

```
void GrElmn_setGrName( GrElmn& a, string newGrName )
{
   strncpy( a.geGrName, newGrName.c_str(), GrElmn::grNameSz-1 );
   a.geGrName[GrElmn::grNameSz-1] = '\0';
} // GrElmn_setGrName
```

Η *writeRandom()* είναι η «δίδυμη» της *readRandom()* αλλά εδώ έχουμε μια ασυμμετρία: δεν υπάρχει παράμετρος για τον ατομικό αριθμό, διότι αυτός υπάρχει ως μέλος της δομής *a*. Κατά τα άλλα:

```
void writeRandom( const GrElmn& a, ostream& bout )
{
   if ( bout.fail() )
      throw ApplicXptn( "writeRandom",
                        ApplicXptn::streamNotOpen );
   bout.seekp( (a.geANumber-1)*GrElmn::saveSize );
   GrElmn_save( a, bout );
} // writeRandom
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct ApplicXptn
{
   enum { cannotOpen, streamNotOpen, rangeErr, cannotClose,
          cannotWrite, cannotRead };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   int  errIntVal;
```

```
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
}; // ApplicXptn

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

void GrElmn_save( const GrElmn& a, ostream& bout );
void GrElmn_load( GrElmn& a, istream& bin );
void GrElmn_display( const GrElmn& a, ostream& tout );
void GrElmn_setGrName( GrElmn& a, string newGrName );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrName( fstream& bInOut, int atNo );

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrName( bInOut, aa );
         }
      } while ( aa != 0 );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "GrElmn_load", ApplicXptn::cannotClose,
                           flNm.c_str() );
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::streamNotOpen:
```

```
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::rangeErr:
            cout << "no element with such atomic number ("
                 << x.errIntVal << ") in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "cannot close file " << x.errIntVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Να και ένα παράδειγμα εκτέλεσης:

```
 Atomic Number (1..103, 0 to exit): 22
atomic number: 22
atomic weight: 47.9
symbol: Ti
name: Titanium
greek name:
new greek name: Τιτάνιο
 Atomic Number (1..103, 0 to exit): 55
atomic number: 55
atomic weight: 132.906
symbol: Cs
name: Cesium
greek name:
new greek name: Κέσιο
 Atomic Number (1..103, 0 to exit): 35
atomic number: 35
atomic weight: 79.904
symbol: Br
name: Bromine
greek name:
new greek name: Βρώμιο
 Atomic Number (1..103, 0 to exit): 0
```

15.14.3 Για το Παράδειγμά μας

Στο παράδειγμα αυτό είδαμε:

- Αρχεία εγγραφών (τιμές τύπου δομής),
- Αρχεία τυχαίας πρόσβασης.

Κατ' αρχάς να παρατηρήσουμε ότι:

- Το πρώτο πρόγραμμα δημιούργησε το αρχείο ως *σειριακό*.

α.α	Όνομα	σύμβ.	ατομικό βάρος
1	Υδρογόνο (Hydrogen)	H	1.0008
2	Ήλιο (Helium)	He	4.0026
3	Λίθιο (Lithium)	Li	6.941
4	Βηρύλλιο (Beryllium)	Be	9.01218
	. . .		

Πίν. 15-1 Τα δεδομένα για το κάθε στοιχείο βρίσκονται σε θέση (στο αρχείο) που σχετίζεται με τον ατομικό αριθμό του.

- Το δεύτερο πρόγραμμα επεξεργάζεται το αρχείο με *τυχαία πρόσβαση*. Μπορεί όμως να το χειριστεί *και ως σειριακό*. Για παράδειγμα, ας πούμε ότι θέλουμε να βγάλουμε έναν πίνακα σαν αυτόν που βλέπεις στον Πίν. 15-1. Γράφοντας την

```
void GrElmn_writeToTable( const GrElmn& a, ostream& tout )
{
   tout << a.geANumber << '\t' << a.geGrName
        << " (" << a.geName << ")\t" << a.geSymbol << '\t'
        << a.geAWeight << endl;
} // GrElmn_writeToTable
```

μπορούμε με τις εντολές:

```
      ofstream tout( "elementsTbl.txt" );
      GrElmn a;
      bInOut.seekg( 0 );
      GrElmn_load( a, bInOut );
      while ( !bInOut.eof() )
      {
         GrElmn_writeToTable( a, tout );
         GrElmn_load( a, bInOut );
      } // while
      tout.close();
```

να πάρουμε το αρχείο **elementsTbl.txt** με περιεχόμενο της μορφής:[13]

```
1\tΥδρογόνο (Hydrogen)\tH\t1.008
2\tΗλιο (Helium)\tHe\t4.0026
3\tΛίθιο (Lithium)\tLi\t6.941
4\tΒηρύλλιο (Beryllium)\tBe\t9.01218
 . . .
```

Όπως βλέπεις, έχουμε βάλει σε μια ομάδα τις συναρτήσεις που αρχίζουν με "**GrElmn_**". Αυτές έχουν το εξής κοινό χαρακτηριστικό: κάθε μια κάνει μια συγκεκριμένη δουλειά με μια μεταβλητή (ένα αντικείμενο) τύπου *GrElmn*. Αργότερα θα δεις ότι θα τις ενσωματώσουμε στο αντικείμενο και θα τις ονομάσουμε **μεθόδους** (methods) για τον χειρισμό του.

Ένα άλλο πράγμα που δείχνουμε είναι η διαχείριση εξαιρέσεων. Μεταξύ μας: μερικές από τις εξαιρέσεις που ετοιμαζόμαστε να συλλάβουμε δεν υπάρχει περίπτωση να ριχτούν. Η διαχείριση μπήκε για διδακτικούς λόγους.

Το παράδειγμά μας επιλέχτηκε για τον εξής λόγο: Ο *Ατομικός Αριθμός* (α.α.) είναι το σημαντικότερο χαρακτηριστικό ενός στοιχείου, έχει δηλαδή πολύ νόημα να λέμε «το χημικό στοιχείο με α.α. 6». Από την άλλη μεριά, ο α.α. παίρνει τιμές από 1 μέχρι 112 (ή 113 ή 114). Έτσι, ήταν πολύ απλό και φυσικό να πούμε ότι: θα βάλουμε τα δεδομένα για το στοιχείο με α.α. *k* στην (*k*-1)-οστή εγγραφή του αρχείου.

Ας πούμε όμως ότι θέλει να χρησιμοποιήσει το αρχείο –με το πρόγραμμά μας– κάποιος όχι και τόσο ειδικός. Για έναν τέτοιον χρήστη έχει περισσότερο νόημα το όνομα «άνθρακας» –ή ο αγγλικός όρος «carbon»– από το «α.α. 6». Αυτός πώς θα βρίσκει τα στοιχεία που θέλει; Αν έχει ένα **ευρετήριο** (index) σαν αυτό του Πίν. 15-2 μπορεί να κάνει τη δουλειά του.

Φυσικά, το σωστό είναι να έχουμε ένα ευρετήριο που θα το χειρίζεται ο υπολογιστής. Στην περίπτωση αυτή βέβαια δεν έχει νόημα να δίνουμε τον ατομικό αριθμό· μπορούμε να δίνουμε κατ' ευθείαν τον αριθμό της ψηφιολέξης από την οποίαν αρχίζει η αποθήκευση των δεδομένων για το συγκεκριμένο στοιχείο, Και πραγματικά, τέτοια ευρετήρια χρησιμοποιούνται στα **Συστήματα Διαχείρισης Βάσεων Στοιχείων** (Data Base Management Systems, DBMS) όπως και σε απλά συστήματα διαχείρισης αρχείων. Φυσικά, η υλοποίηση των ευρετηρίων γίνεται με μεθόδους πιο πολύπλοκες αλλά και πιο αποδοτικές από άποψη ταχύτητας και χρήσης μνήμης. Αργότερα θα δούμε μια πολύ απλή μορφή ευρετηρίου.

---

[13] Φυσικά, οι στηλοθέτες (tabs) δεν θα φαίνονται. Αν εισαγάγεις το αρχείο σε ένα εργαλείο σαν το MS Excel ή το MS Word παίρνεις τον πίνακα.

Εδώ να παρατηρήσουμε και το εξής: αν στη δεύτερη στηλη του πίνακα δεν έχουμε τον α.α. αλλά την ψηφιολέξη του αρχείου στην οποίαν αρχίζει η αποθήκευση των δεδομένων για το συγκεκριμένο στοιχείο τότε αλλάζουν πολλά πράγματα: μπορείς να ψάχνεις και εγγραφές μεταβλητού μήκους (μπορείς να ψάχνεις και μορφοποιημένα αρχεία). Αλλά να ψάχνεις μόνο· η ενημέρωση είναι πολύπλοκη.

Όνομα	Ατομικός Αριθμός
Actinium	89
Aluminum	13
Americium	95
Antimony	51
Argon	18
Arsenic	33
Astatine	85
Barium	56
. . .	. . .

Πίν. 15-2 Τα δεδομένα για το κάθε στοιχείο βρίσκονται σε θέση (στο αρχείο) που σχετίζεται με τον ατομικό αριθμό του.

15.15 Ανακεφαλαίωση

Οι τύποι-δομές μας επιτρέπουν να παριστάνουμε και να διαχειριζόμαστε τιμές-αντικείμενα που απαρτίζονται από άλλες τιμές, άλλων τύπων και αναφέρονται στην ίδια φυσική οντότητα. Μπορούμε να διαχειριζόμαστε ένα αντικείμενο είτε ολόκληρο είτε κατά μέλη.

Εκτός από μορφοποιημένα αρχεία (text) μπορούμε να χρησιμοποιούμε και *μη μορφοποιημένα* στα οποία τα δεδομένα αντιγράφονται από τη μνήμη χωρίς να μετατραπούν σε χαρακτήρες. Κυριότερο πλεονέκτημα: η *ταχύτητα*, κυριότερο μειονέκτημα: *περιορίζεται η δυνατότητα μεταφοράς*.

Με τη μέθοδο *seekp()* (των *ofstream*, *fstream*) μπορείς να μετακινηθείς για να γράψεις σε οποιαδήποτε θέση του αρχείου. Παρομοίως, για να διαβάσεις από οποιαδήποτε θέση του αρχείου χρησιμοποίησε τη *seekg()* (των *ifstream*, *fstream*).

Στο παράδειγμα με τα χημικά στοιχεία πήραμε μια πρώτη γεύση για το πώς δουλεύουμε με *εξαιρέσεις*. Έτσι περίπου θα γράφουμε τα προγράμματά μας από εδώ και πέρα.

Το κεφάλαιο αυτό συνοδεύεται και από δύο projects. Να τα διαβάσεις οπωσδήποτε και να γυρίσεις στα προηγούμενα κεφάλαια για να ξαναδείς ό,τι δεν θυμάσαι. Το πρώτο (ξανα)δίνεται κυρίως για να το δούμε με εξαιρέσεις. Το δεύτερο θα ξαναδοθεί με άλλον τρόπο στη συνέχεια.

Ασκήσεις

Β Ομάδα

15-1 Μας δίνεται μη μορφοποιημένο (binary) αρχείο με όνομα `dates.dta` και άγνωστο πλήθος τιμών τύπου:

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
}; // Date
```

που έχουν φυλαχθεί με τη

```
void Date_save( const Date& a, ostream& bout )
{
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&a.year),
               sizeof(a.year) );
   bout.write( reinterpret_cast<const char*>(&a.month),
               sizeof(a.month) );
   bout.write( reinterpret_cast<const char*>(&a.day),
               sizeof(a.day) );
```

```
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::cannotWrite );
}; // Date_save
```

όπου `XXX_Xptn` κάποια κλάση εξαιρέσεων.

Γράψε πρόγραμμα που θα ζητάει από τον χρήστη μια ημερομηνία –ας την πούμε *limDate*– και διαβάζοντας το αρχείο θα δημιουργεί δύο νέα μορφοποιημένα (text) αρχεία στα οποία θα φυλάγει:

- στο ένα, με όνομα `odates.txt`, τις ημερομηνίες που είναι παλιότερες από τη *limDate* και
- στο άλλο, με όνομα `ndates.txt`, τις ημερομηνίες που είναι ίδιες με ή νεότερες από τη *limDate*.

Το πρόγραμμα θα υπολογίζει και θα μας λέει στο τέλος:

- το πλήθος των τιμών που έβαλε σε κάθε αρχείο,
- την πιο παλιά ημερομηνία που βρήκε και
- την πιο νέα ημερομηνία που βρήκε.

Γ Ομάδα

15-2 Ένας πίνακας που έχει πολλά στοιχεία του ίσα με 0 (μηδέν) λέγεται **αραιός** (sparse). Για έναν τέτοιο πίνακα μπορούμε, για οικονομία μνήμης, να μην φυλάγουμε όλα τα στοιχεία αλλά μόνον τα μη μηδενικά, το καθένα με τη θέση του. Ας πούμε ότι έχουμε έναν δισδιάστατο πίνακα τύπου `double` με *nR* γραμμές και *nC* στήλες. Για την αποθήκευσή του απαιτούνται

nR `*` *nC* `* sizeof(double)`

ψηφιολέξεις. Αν έχει *nNZ* μη μηδενικά στοιχεία και απόθηκεύσουμε το καθένα από αυτά σε ένα στοιχείο τύπου

```
struct ArrElmn
{
   unsigned int row;
   unsigned int col;
   double       value;
}; // ArrElmn
```

έχουμε οικονομία όταν

nNZ `* sizeof(`*ArrElmn*`) <` *nR* `*` *nC* `* sizeof(double)`

Σε ένα μη μορφοποιημένο αρχείο –με όνομα στον δίσκο `sprs017.dta`– έχουμε αποθηκευμένα τα στοιχεία ενός αραιού δισδιάστατου πίνακα ως εξής:

- Στην αρχή υπάρχει μια τιμή `unsigned int` που μας δίνει το πλήθος γραμμών (*nR*) του πίνακα.
- Ακολουθεί άλλη μια τιμή `unsigned int` που μας δίνει το πλήθος στηλών (*nC*) του πίνακα.
- Ακολουθούν *nR*`*`*nC* τιμές τύπου `double` που είναι οι τιμές των στοιχείων κατά γραμμές: πρώτα τα στοιχεία της γραμμής 0, μετά τα στοιχεία της γραμμής 1 κ.ο.κ.

Γράψε πρόγραμμα που θα αντιγράφει το αρχείο σε ένα άλλο, μη μορφοποιημένο, με όνομα στον δίσκο `sprs017cn.dta`, με το εξής περιεχόμενο:

- Στην αρχή θα υπάρχει μια τιμή `unsigned int`, το πλήθος γραμμών (*nR*) του πίνακα.
- Στη συνέχεια άλλη μια τιμή `unsigned int`, μας δίνει το πλήθος στηλών (*nC*) του πίνακα.
- Στη συνέχεια τιμές τύπου *ArrElmn*, μια για κάθε μη μηδενικό στοιχείο του αρχικού πίνακα.

project

1

Αυτοκίνητα στον Δρόμο

Περιεχόμενα:

Prj01.1 Το Πρόβλημα

Θα ξαναλύσουμε το πρόβλημα που λύσαμε στην §13.11 αλλά αυτήν τη φορά με χρήση εξαιρέσεων. Για διευκόλυνσή σου ξαναδίνουμε το πρόβλημα:

> *Ένα συνεργείο έκανε μέτρηση ροής οχημάτων σε κάποιο δρόμο. Σε αρχείο, text – με το όνομα στο δίσκο* **`autoflow.txt`***– καταγράφηκαν οι τιμές ροής σε οχήματα/min κάθε λεπτό. Το πλήθος των τιμών στο αρχείο είναι άγνωστο, αλλά σίγουρα θετικό.*
>
> *Οι πρώτες τρεις γραμμές του αρχείου έχουν την «ταυτότητα» της μέτρησης ως εξής:*

```
Υπεύθυνος:\t<όνομα>\t<επώνυμο>
Σημείο Μετρήσεων:\t<οδός-αριθμός>\t<περιοχή>\t<δήμος>
Αρχή:\tdd.mm.yyyy\thh:mm
```

> *Στις υπόλοιπες γραμμές δίνονται οι τιμές από τη μέτρηση, μια σε κάθε γραμμή. Κάθε τιμή είναι γραμμένη στις πέντε πρώτες θέσεις. Πέρα από τη δέκατη θέση μπορεί να υπάρχουν σχόλια που περιγράφουν συμβάντα κατά τη διάρκεια της μέτρησης. Να ένα παράδειγμα:*

```
Υπεύθυνος:    Ανδρέας       Νικολόπουλος
Σημείο Μετρήσεων: Τζαβέλλα 48    Νεάπολις      Αθήνα
Αρχή:  15.06.2009    05:00
   26
   30
    8
   28
. . .
   17
```

```
   19
    4
   12      / Αρχή λειτουργίας σηματοδότησης
   28
   17
. . .
   17
   31      / Βλάβη σηματοδότη Τζαβέλλα & Γιωργάκου
   23
   24
. . .
```

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει:
1. Το πλήθος τιμών του αρχείου καθώς και τη διάρκεια των μετρήσεων σε h και min.
2. Το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων.
3. Τη μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min.
Όλα αυτά θα γράφονται στην τελική έκθεση, που θα γραφεί σε αρχείο με το όνομα `report.txt`. *Στις πρώτες τρεις γραμμές του αρχείου θα αντιγραφούν οι τρεις πρώτες γραμμές του* `autoflow.txt`.
Ένα από τα ζητούμενα της δουλειάς είναι και η μελέτη της μεταβολής της ροής οχημάτων. Ως πρώτο βήμα μας ζητείται να δημιουργήσουμε ένα άλλο αρχείο με τις μεταβολές ροής ανά min.

Prj01.2 Η Δομή Εξαιρέσεων

Στην αρχική λύση μεταφέραμε την πληροφορία «κάτι δεν πάει καλά» με παραμέτρους "`bool& ok`". Τώρα θα τη μεταφέρουμε με εξαιρέσεις. Θα ρίχνουμε εξαιρέσεις τύπου:

```
struct ApplicXptn
{
   enum { . . . };
   char funcName[100];
   int  errCode;
// . . .
   ApplicXptn( const char* fn, int ec, . . . )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
   // . . .
   }
}; // ApplicXptn
```

όπως τον είδαμε στην §15.10.

Θα συμπληρώνουμε τη δομή όπως θα (ξανα)γράφουμε το πρόγραμμά μας και θα αντιμετωπίζουμε τις ανάγκες του.

Προφανώς θα ξεκινήσουμε από τις συναρτήσεις που έχουν παράμετρο *ok*.

Prj01.3 Η Συνάρτηση *openWrNoReplace()*

Η πρώτη συνάρτηση που θα δούμε είναι η *openWrNoReplace()*. Τη γράφουμε ως:

```
void openWrNoReplace( ofstream& newStream, string flNm )
{
   ifstream test( flNm.c_str() );                        // άνοιξε για διάβασμα
   if ( !test.fail() )                                    // υπάρχει το αρχείο,
   {
      test.close();                            // κλείσε το και μην το πειράξεις
      throw ApplicXptn( "openWrNoReplace", ApplicXptn::fileExists,
```

```
                        flNm.c_str() );
   }
   newStream.open( flNm.c_str() );                          // δημιούργησέ το
   if ( newStream.fail() )
      throw ApplicXptn( "openWrNoReplace", ApplicXptn::cannotCreate,
                        flNm.c_str() );
} // openWrNoReplace
```

Πρόσεξε ότι *η λογική της συνάρτησης είναι ίδια με αυτήν της αρχικής*. Γράφουμε σε σχόλιο «*κλείσε το και μην το πειράξεις*». Αυτό πώς διασφαλίζεται; Με το ότι ακολουθεί εντολή **throw** και εκεί θα διακοπεί η εκτέλεση της συνάρτησης και η "`newStream.open( flNm.c_str() )`" δεν θα εκτελεσθεί!

Εδώ χρησιμοποιούμε δύο κωδικούς σφάλματος που πρέπει να τους ορίσουμε στη δομή εξαιρέσεων:

```
   enum { fileExists, cannotCreate, . . . };
```

Ακόμη προκύπτει ανάγκη για δημιουργό με τρίτη παράμετρο πίνακα χαρακτήρων (ορμαθό της C):

```
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
```

Το *errStrVal* θα δηλωθεί ως μέλος της δομής:

```
   char errStrVal[100];
```

Prj01.4 Η Συνάρτηση *openFiles()*

Η *openWrNoReplace()* καλείται από την *openFiles()* που –και αυτή– έχει παράμετρο *ok*.

Ξαναδές στην §13.11.1 τη λογική της αρχικής συνάρτησης. Ανοίγουμε τα ρεύματα το ένα μετά το άλλο (*autoflow*, *differences*, *report*) εφόσον δεν υπήρξε πρόβλημα με το άνοιγμα των προηγουμένων. Και εδώ, αν γράψουμε:

```
   autoflow.open( autoFlNm.c_str() );
   if ( autoflow.fail() )
      throw ApplicXptn( "openFiles",
                        ApplicXptn::cannotOpen, autoFlNm.c_str() );
// . . .
```

πετυχαίνουμε το εξής: Αν δεν ανοίξει το *autoflow* ρίχνεται εξαίρεση και δεν προχωρούμε παρακάτω.

Το επόμενο ρεύμα (*differences*) ανοίγεται με την *openWrNoReplace()*. Τώρα, το «δεν άνοιξε το *differences*» σημαίνει ότι ρίχτηκε και μια εξαίρεση. Θα πρέπει να είμαστε έτοιμοι να την πιάσουμε:

```
   try
   {
      openWrNoReplace( differences, difFlNm );
   }
   catch( ApplicXptn& x )
   {
      autoflow.close();
      throw;
   } // catch
```

Τι κάνουμε εδώ; Αν πιάσουμε εξαίρεση κλείνουμε πρώτα το *autoflow* που είναι ήδη ανοικτό και ξαναρίχνουμε την εξαίρεση.

Παρομοίως θα μπορούσαμε να χειριστούμε και το άνοιγμα του *report*:

```
   try
   {
      openWrNoReplace( report, reprtFlNm );
   }
```

```
   catch( ApplicXptn& x )
   {
      autoflow.close();
      differences.close();
      throw;
   } // catch
```

Εδώ φυσικά θα πρέπει να κλείσουμε και το *differences*.

Αφού όμως η εξαίρεση κουβαλάει μαζί της και το όνομα του αρχείου που είχε το πρόβλημα (*x.errStrVal*), μπορούμε να γράψουμε:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm )
{
   autoflow.open( autoFlNm.c_str() );
   if ( autoflow.fail() )
      throw ApplicXptn( "openFiles",
                        ApplicXptn::cannotOpen, autoFlNm.c_str() );
// το autoflow ανοικτό
   try
   {
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
   }
   catch( ApplicXptn& x )
   {
      autoflow.close();
      if ( x.errStrVal == reprtFlNm )      // δεν άνοιξε το report
         differences.close();
      throw;
   } // catch
} // openFiles
```

Αν πάρουμε εξαίρεση από την *openWrNoReplace()* κλείνουμε οπωσδήποτε το *autoflow*. Αν προβληματικό αρχείο ήταν το `report.txt` τότε κλείνουμε και το ρεύμα *differences* που έχει ανοίξει πιο πριν.

Για να μην έχουμε τη συνάρτηση «κομμένη στα δύο», δεν θα μπορούσαμε να γράψουμε:

```
   try
   {
      autoflow.open( autoFlNm.c_str() );
      if ( autoflow.fail() )
         throw ApplicXptn( "openFiles",
                           ApplicXptn::cannotOpen,
                           autoFlNm.c_str() );
   // το autoflow ανοικτό
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
   }
   catch( ApplicXptn& x )
   { . . . }
```

Ναι, θα μπορούσαμε. Αλλά προσοχή: αν αποτύχει το άνοιγμα του *autoflow* η εξαίρεση θα συλληφθεί από την **catch** που έχουμε εδώ και η διαχείριση θα είναι πιο πολύπλοκη:

```
   catch( ApplicXptn& x )
   {
      if ( x.errStrVal == difFlNm )   // δεν άνοιξε το differences
         autoflow.close();
      else if ( x.errStrVal == reprtFlNm ) // δεν άνοιξε το report
      {
         autoflow.close();
         differences.close();
      }
      throw;
   } // catch
```

Αν το προτιμάς...

Από τη μετατροπή της *openFiles()* προκύπτει ανάγκη για έναν ακόμη κωδικό σφάλματος:

```
   enum { fileExists, cannotCreate, cannotOpen };
```

Prj01.5 Η Συνάρτηση *copyTitle()*

Στην *copyTitle()* η αντικατάσταση της *ok* με **throw** είναι απλή:

```
void copyTitle( ifstream& autoflow, ofstream& report )
{
   int  lineCount( 0 );   // μετρητής γραμμών
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
   if ( lineCount != 3 )
      throw ApplicXptn( "copyTitle", ApplicXptn::incomplete, lineCount );
} // copyTitle
```

Εδώ όμως, πέρα από την ανάγκη για νέο κωδικό σφάλματος (*incomplete*) προκύπτει η ανάγκη και για νέο δημιουργό εξαιρέσεων που να έχει τρίτη παράμετρο τύπου `int`:

```
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
```

Το *errIntVal* είναι μέλος της *ApplicXptn*:

```
   int  errIntVal;
```

Φυσικά, τώρα έχουμε:

```
   enum { fileExists, cannotCreate, cannotOpen, incomplete };
```

Prj01.6 Η Συνάρτηση *closeFiles()*

Η τελευταία συνάρτηση με παράμετρο *ok* είναι η *closeFiles()*. Την ξαναγράφουμε ως εξής:

```
void closeFiles( ifstream& autoflow, ofstream& differences, ofstream& report )
{
   autoflow.close();

   differences.close();
   if ( differences.fail() )
      throw ApplicXptn( "closeFiles", ApplicXptn::cannotClose, "διαφορές" );
   report.close();
   if ( report.fail() )
      throw ApplicXptn( "closeFiles", ApplicXptn::cannotClose, "έκθεση" );
} // closeFiles
```

Αυτή *δεν είναι λειτουργικώς ισοδύναμη με την αρχική*. Αν δεν μπορέσει να κλείσει το *differences* θα ρίξει εξαίρεση και δεν θα προσπαθήσει να κλείσει το *report*. Η αρχική θα προσπαθήσει να κλείσει και τα δύο. Δηλαδή είναι προτιμότερη η αρχική μορφή της συνάρτησης; Ναι, αν έχει νόημα το να πάρουμε το ένα μόνον αρχείο. Αν θέλουμε και τα δύο αρχεία θα πρέπει να ζητήσουμε να ξαναεκτελεσθεί το πρόγραμμα και στις δύο περιπτώσεις.

Πρόσεξε ακόμη ότι επειδή στη συνάρτηση αυτή δεν περνούν τα ονόματα των αρχείων *βάζουμε απλό κείμενο ως τρίτο όρισμα του δημιουργού*.

Prj01.7 Η *ApplicXptn* (τελικώς)

Ας δούμε τώρα πώς έγινε δομή εξαιρέσεων:

```
struct ApplicXptn
{
   enum { fileExists, cannotCreate, cannotOpen, incomplete, cannotClose };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   int  errIntVal;

   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
}; // ApplicXptn
```

Prj01.8 Και η `main`

Τώρα, η `main` θα είναι ως εξής:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.dta
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.txt
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   try
   {
      openFiles( autoflow, autoFlNm, differences, difFlNm,
                 report, reprtFlNm );
      process( autoflow, differences, report );
      closeFiles( autoflow, differences, report );
      cout << "Τέλος καλό, όλα καλά..." << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::fileExists:
            cout << "από την " << x.funcName << ": το αρχείο "
                 << x.errStrVal << " υπάρχει" << endl;
            break;
         case ApplicXptn::cannotCreate:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να δημιουργήσω το " << x.errStrVal << endl;
            break;
         case ApplicXptn::cannotOpen:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να ανοίξω το " << x.errStrVal << endl;
            break;
         case ApplicXptn::incomplete:
            cout << "από την " << x.funcName
                 << ": ελλιπή δεδομένα. Διαβάστηκαν "
                 << x.errIntVal << " γραμμές " << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να κλείσω το αρχείο " << x.errStrVal << endl;
            break;
```

```
         default:
            cout << "unexpected ApplicXptn from " << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Η **main** απλουστεύθηκε σε εντυπωσιακό βαθμό:

```
      openFiles( autoflow, autoFlNm, differences, difFlNm,
                 report, reprtFlNm );
      process( autoflow, differences, report );
      closeFiles( autoflow, differences, report );
      cout << "Τέλος καλό, όλα καλά..." << endl;
```

Αλλα τίποτε δεν είναι δωρεάν στον κόσμο αυτόν: Έφυγαν οι "`if (ok)…`" και μας ήλθε αυτή η μακροσκελής **catch**!

Prj01.9 Η *openFiles()* Αλλιώς

Στην §14.9 γράφαμε: «*Θα διαχειριζόμαστε πάντοτε τις εξαιρέσεις στη* **main***; Όχι. Στη συνέχεια, θα δούμε πώς αποφασίζουμε που και πώς μπορεί να γίνει η καλύτερη διαχείριση της κάθε εξαίρεσης.*» Εδώ διαχειριζόμαστε όλες τις εξαιρέσεις στη **main** αλλά, αν το ξανασκεφτούμε, αυτή δεν είναι η καλύτερη λύση.

Ας πάρουμε τις εξαιρέσεις που ρίχνουμε αν δεν μπορέσουμε να ανοίξουμε κάποιο αρχείο ή κάποια αρχεία· δεν υπάρχει λόγος να τις αφήσουμε να φτάσουν μέχρι τη **main**. Εκεί, αρκεί να φτάσει τον μήνυμα «δεν ανοίγουν τα αρχεία». Καλύτερα λοιπόν να (ξανα)χρησιμοποιήσουμε την *ok* ως εξής:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm,
                bool& ok )
{
   ok = false;
   try
   {
      autoflow.open( autoFlNm.c_str() );
      if ( autoflow.fail() )
         throw ApplicXptn( "openFiles", ApplicXptn::cannotOpen,
                           autoFlNm.c_str() );
   // το autoflow ανοικτό
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
      ok = true;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::fileExists:
         case ApplicXptn::cannotCreate:
            autoflow.close();
            if ( x.errStrVal == reprtFlNm )// δεν άνοιξε το report
               differences.close();
            cout << "από την " << x.funcName;
            if ( x.errCode == ApplicXptn::fileExists )
               cout << ": το αρχείο " << x.errStrVal << " υπάρχει" << endl;
            else // x.errCode == ApplicXptn::cannotCreate
               cout << ": δεν μπορώ να δημιουργήσω το " << x.errStrVal
                    << endl;
            break;
```

```
         case ApplicXptn::cannotOpen:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να ανοίξω το " << x.errStrVal << endl;
            break;
         default:
            throw;
      } // switch
   } // catch
} // openFiles
```

Αυτή η *openFiles()* πιάνει τις εξαιρέσεις που έχουν σχέση με το άνοιγμα των αρχείων: *ApplicXptn* με κωδικούς *fileExists*, *cannotCreate* και *cannotOpen* ενώ αφήνει να περνούν όλες οι άλλες.

Παρατήρηση: ▶

Στην §14.9 γράφαμε ακόμη: «*Οι εξαιρέσεις ρίχνονται μόνο από συναρτήσεις που καλούμε στην ομάδα* ***try****; Όχι! Αργότερα θα δεις παραδείγματα που θα έχουμε εντολές* `throw` *μέσα στην ομάδα της* ***try****.*» Εδώ έχουμε ένα τέτοιο παράδειγμα: περιμένουμε βέβαια εξαιρέσεις από τις δύο κλήσεις στην *openWrNoReplace()* αλλά υπάρχει και **throw** ακριβώς πάνω από αυτές.◀

Η αποτυχία στο άνοιγμα των αρχείων γνωστοποιείται προς τα έξω με τιμή `false` της *ok*. Το τι θα κάνει η **main** σε μια τέτοια περίπτωση είναι άλλη ιστορία. Για παράδειγμα, θα μπορούσε να γίνει κάτι σαν:

```
      do {
         openFiles( autoflow, autoFlNm, differences, difFlNm,
                    report, reprtFlNm, ok );
         if ( !ok )
            getFileNames( autoFlNm, difFlNm, reprtFlNm, quit );
      } while ( !ok && !quit );
      if ( ok )
      {
         process( autoflow, differences, report );
         closeFiles( autoflow, differences, report );
         cout << "Τέλος καλό, όλα καλά..." << endl;
      }
```

Η

```
void getFileNames( string& autoFlNm, string& difFlNm,
                   string& reprtFlNm, bool& quit )
```

(άσκηση για σένα) ζητάει από τον χρήστη να αλλάξει τα ονόματα (κάποιων) αρχείων ή να τα παρατήσει. Στην τελευταία περίπτωση η *quit* επιστρέφει τιμή `true`.

Με τον ίδιο τρόπο μπορούμε να χειριστούμε και την *closeFiles()*.

project

2

Διανύσματα στις 3 Διαστάσεις

Περιεχόμενα:

Prj02.1 Το Πρόβλημα

Το Project αυτό είναι ένα μεγάλο παράδειγμα επιφόρτωσης τελεστών. Είναι καλό να το διαβάσεις προσεκτικά διότι *αργότερα θα το ξαναδούμε και θα αλλάξουμε μερικά από αυτά πόυ θα δούμε εδώ.*

Ένα τρισδιάστατο (ελεύθερο) ***διάνυσμα*** *(vector) είναι μια διαταγμένη τριάδα πραγματικών αριθμών (x, y, z).*

Όρισε έναν τύπο δομής Vector3 που τα αντικείμενά της θα είναι διανύσματα του χώρου τριών διαστάσεων. Όρισε έναν ερήμην δημιουργό που θα μας δίνει ελεύθερο διάνυσμα με συνιστώσες (0, 0, 0) αλλά θα μπορεί να πάρει και αρχικές τιμές των συνιστωσών.

Α. Έστω ότι έχουμε δύο διανύσματα $\mathbf{v}_1 = (x_1, y_1, z_1)$ *και* $\mathbf{v}_2 = (x_2, y_2, z_2)$.

1. Τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *μπορεί να συγκριθούν για ισότητα και μόνον (δηλαδή έχουν νόημα μόνον οι "==" και "!="). Είναι ίσα αν και μόνον αν* $x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2$. *Επιφόρτωσε τους τελεστές σύγκρισης "==" και "!=" για τον Vector3.*

2. Μπορούμε ακόμη να τα προσθέσουμε ή τα αφαιρέσουμε και να πάρουμε ένα διάνυσμα $\mathbf{v}_1 \pm \mathbf{v}_2$ *με συνιστώσες* $(x_1 \pm x_2, y_1 \pm y_2, z_1 \pm z2)$.
Επιφόρτωσε τους τελεστές "+" και "-" για τον Vector3.
Επιφόρτωσε τους τελεστές "+=" και "-=" για τον Vector3.

3. Πολλαπλασιάζοντας ένα διάνυσμα $\mathbf{v}_1$ *επί έναν πραγματικό αριθμό α (*$\alpha\mathbf{v}_1$ *ή* $\mathbf{v}_1\alpha$*) παίρνουμε ένα νέο διάνυσμα με συνιστώσες* $(\alpha x_1, \alpha y_1, \alpha z_1)$.
Επιφόρτωσε καταλλήλως τον τελεστή "" για τον Vector3.*
Επιφόρτωσε τον τελεστή "=" για τον Vector3.*

4. Ειδική περίπτωση: το $(-1)\mathbf{v}_1$ *μπορεί να γραφεί και ως* $-\mathbf{v}_1$.
Επιφόρτωσε καταλλήλως τον ενικό τελεστή "-" για τον Vector3.

5. Μπορούμε να πολλαπλασιάσουμε τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *και να πάρουμε τον πραγματικό αριθμό:* $\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1x_2 + y_1y_2 + z_1z_2$ *που λέγεται* ***εσωτερικό γινόμενο*** *(inner ή dot product).*
*Επιφόρτωσε (άλλη μια φορά) καταλλήλως τον τελεστή "*****" για τον Vector3.*
6. Μπορούμε όμως να πάρουμε και το ***εξωτερικό γινόμενο*** *(outer ή cross product)* $\mathbf{v}_1 \times \mathbf{v}_2$ *που είναι διάνυσμα:* $\mathbf{v}_1 \times \mathbf{v}_2 = (y_1z_2-z_1y_2,\ x_2z_1-z_2x_1,\ y_2x_1-x_2y_1)$.
*Επιφόρτωσε καταλλήλως τον τελεστή "**^**"για τον Vector3 ώστε να υπολογίζει το εξωτερικό γινόμενο.*
*7. Καλό θα είναι να μπορούμε να γράψουμε στην οθόνη ή σε ένα αρχείο text την τιμή του διανύσματος με τον "**<<**".*
*Επιφόρτωσε τον "**<<**" για αντικείμενα τύπου Vector3.*

8. Τέλος, για κάθε διάνυσμα ορίζεται ένα ευκλείδιο μέτρο: $|\mathbf{v}| = \sqrt{x^2+y^2+z^2}$ *(=* $\sqrt{\mathbf{v}\cdot\mathbf{v}}$ *) που είναι το μήκος του διανύσματος.*
Γράψε συνάρτηση Vector3_abs που να δίνει το ευκλείδιο μέτρο ενός αντικειμένου τύπου Vector3.
Β. Για μια απλή εφαρμογή των παραπάνω, γράψε πρόγραμμα που θα υπολογίζει τη δύναμη Laplace $\mathbf{F}_L = q(\mathbf{v} \times \mathbf{B})$ *που ασκείται σε φορτίο* $q = 1\ \mu Cb$ *που εισέρχεται με ταχύτητα* $\mathbf{v} = (10^8, 0, 0)$ *σε μαγνητικό πεδίο* $\mathbf{B} = (0, 0, 10)$. *Επιβεβαίωσε ότι η δύναμη είναι κάθετη στα* $\mathbf{v}$ *και* $\mathbf{B}$ *(δηλαδή:* $\mathbf{F}_L \cdot \mathbf{v} = 0$ *και* $\mathbf{F}_L \cdot \mathbf{B} = 0$*).*
Το εμβαδόν παραλληλογράμμου που οι πλευρές του είναι παράλληλες προς τα ελεύθερα διανύσματα $\mathbf{a}$ *και* $\mathbf{b}$ *είναι το διάνυσμα* $\mathbf{S} = \mathbf{a} \times \mathbf{b}$. *Αν έχουμε το παραλληλόγραμμο μέσα σε ένα μαγνητικό πεδίο τότε η μαγνητική ροή που διέρχεται από αυτό είναι ίση με* $\mathbf{B} \cdot \mathbf{S}$. *Συμπλήρωσε το πρόγραμμά σου με εντολές που θα υπολογίζουν τη μαγνητική ροή που διέρχεται από παραλληλόγραμμο με πλευρές* $\mathbf{a} = (0.1, 0.2, 0.3)$ *και* $\mathbf{b} = (0.2, 0.3, 0.4)$ *όταν βρίσκεται στο μαγνητικό πεδίο που δώσαμε παραπάνω.*
Με τα $\mathbf{a}$ *και* $\mathbf{b}$ *επιβεβαίωσε την ταυτότητα:* $|\mathbf{a}|^2|\mathbf{b}|^2 = (\mathbf{a}\cdot\mathbf{b})^2 + |\mathbf{a}\times\mathbf{b}|^2$.

Prj02.2 Ο Τύπος *Vector3* και οι Δημιουργοί

Ο τύπος αυτός είναι σαν τον *complex* αλλά με τρία μέλη:

```
struct Vector3
{
   double x;
   double y;
   double z;
}; // Vector3
```

Γράφουμε τον ερήμην δημιουργό με αρχικές τιμές όπως κάναμε και στην *complex*:

```
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
```

Πρόσεξε ότι με αυτόν τον δημιουργό μπορούμε να κάνουμε τις εξής δηλώσεις:

```
   Vector3 v0;                                 // (0, 0, 0)
   Vector3 v1( 1 );                            // (1, 0, 0)
   Vector3 v2( sqrt(2), 3 );                   // (1.41421, 3, 0)
   Vector3 v3( sqrt(3), sqrt(5), 2*sqrt(2) ); // (1.73205, 2.23607, 2.82843)
```

Να λοιπόν ο ορισμός του τύπου μας:

```
struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
```

```
   {  x = ax;  y = ay;  z = az;  }
}; // Vector3
```

Prj02.3 Οι Τελεστές Σύγκρισης

Σύμφωνα με όσα είπαμε στην §14.6.4, ένας δυαδικός τελεστής ("`@`") επιφορτώνεται με μια συνάρτηση της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Για έναν τελεστή σύγκρισης, όπως είναι ο "`==`", *Trv* είναι ο `bool`. Για την περίπτωσή μας *Tl* και *Tr* είναι, κατ' αρχήν, ο *Vector3*. Θα μπορούσαμε λοιπόν να γράψουμε:

```
bool operator==( Vector3 lhs, Vector3 rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Αυτή είναι απολύτως σωστή, αλλά, παρ' όλα αυτά, θα την αλλάξουμε λιγάκι, ως προς τις παραμέτρους:

```
bool operator==( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Ποιο είναι το πλεονέκτημα της δεύτερης μορφής; Κάθε φορά που την καλούμε θα περάσουν, ως παράμετροι, δύο βέλη ενώ στην πρώτη μορφή θα περάσουν δυο τιμές *Vector3*. Σε ψηφιολέξεις αυτό θα μπορούσε να σημαίνει (ενδεικτικώς) 8:48· εξαπλάσιο! Ναι, αλλά το 48 είναι πολύ μικρό για να μας δημιουργήσει πρόβλημα. Υπάρχουν όμως και περιπτώσεις αντικειμένων που μπορεί να είναι πολύ μεγάλα· σκέψου, για παράδειγμα, ένα αντικείμενο τύπου *string* με τιμή το κείμενο ενός βιβλίου 1000 σελίδων. *Τα πολύ μεγάλα αντικείμενα δεν τα περνάμε ως παραμέτρους τιμής αλλά ως παραμέτρους αναφοράς με "`const`"*. Θα χρησιμοποιούμε λοιπόν αυτόν τον τρόπο παντού στις επιφορτώσεις τελεστών για να τον συνηθίσεις(!)

Ας έλθουμε τώρα στον "`!=`"· χρειάζεται να τον γράψουμε; Αφού έχουμε τον "`==`" θα μπορούμε να γράφουμε `!(a == b)` αντί για `a != b`. Φυσικά, αλλά γενικώς θα τηρούμε την εξής προγραμματιστική πρακτική:[1]

- ***Όταν επιφορτώνουμε έναν τελεστή σύγκρισης "@" επιφορτώνουμε και τον αντίθετό του με χρήση του "@".***

Έχουμε λοιπόν:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return !(lhs == rhs);
} // operator!=( Vector3
```

Έτσι, έχουμε και τους δύο τελεστές χωρίς ασυμβατότητες (σίγουρα). Αν θέλεις μπορείς να πας ένα βήμα παραπέρα και να αποφύγεις μια κλήση συνάρτησης:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x != rhs.x) ||
          (lhs.y != rhs.y) || (lhs.z != rhs.z);
} // operator!=( Vector3
```

Για τελεστές που αυτό το τελευταίο βήμα είναι πιο πολύπλοκο –και υπάρχει μεγάλη πιθανότητα λάθους– μην το κάνεις.

---

[1] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposite (such as "==" and "!="), it is appropriate to define both.*»

Prj02.4 Οι Τελεστές "+", "-", "*", "^"

Οι δύο προσθετικοί (δυαδικοί) τελεστές και ο "^" (εξωτερικό γινόμενο) θα επιφορτωθούν με συναρτήσεις της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Γι ά τις πράξεις αυτές έχουμε:

+: *Vector3* × *Vector3* ↛ *Vector3*

-: *Vector3* × *Vector3* ↛ *Vector3*

^: *Vector3* × *Vector3* ↛ *Vector3*

Τις θεωρούμε μερικές συναρτήσεις επειδή παίρνουμε υπόψη την περίπτωση υπερχείλησης. Στη συνέχεια, στην υλοποίηση, θα τις χειριστούμε σαν ολικές συναρτήσεις· δηλαδή δεν θα ρίχνουμε εξαιρέσεις.

Στις περιπτώσεις αυτές *Trv*, *Tl* και *Tr* είναι ο *Vector3*. Σύμφωνα όμως με αυτά που είπαμε παραπάνω, θα βάλουμε τους *Tl* και *Tr* `const Vector3&`:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x + rhs.x;
   fv.y = lhs.y + rhs.y;
   fv.z = lhs.z + rhs.z;
   return fv;
} // operator+( const Vector3&

Vector3 operator-( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x - rhs.x;
   fv.y = lhs.y - rhs.y;
   fv.z = lhs.z - rhs.z;
   return fv;
} // operator-( const Vector3&

Vector3 operator^( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.y*rhs.z - lhs.z*rhs.y;
   fv.y = rhs.x*lhs.z - rhs.z*lhs.x;
   fv.z = rhs.y*lhs.x - rhs.x*lhs.y;
   return fv;
} // operator^( const Vector3
```

Σημείωση:▶

Υπάρχει δυαδικός τελεστής "^" στη C++; Ναι, αλλά θα τον μάθουμε αργότερα. Στην §14.6 λέγαμε «*Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.*» Αυτό το σεβόμαστε με την επιφόρτωση που κάνουμε; Όχι, αλλά όπως θα καταλάβεις δεν δημιουργεί οποιαδήποτε σύγχυση.◀

Ο "*" θέλει να σκεφτούμε κάτι παραπάνω. Αν έχουμε δηλώσει:

```
Vector3 v1, v2;
double  a;
```

θα πρέπει να μπορούμε να γράψουμε είτε:

```
v2 = a*v1;
```

είτε:

```
v2 = v1*a;
```

Έχουμε δηλαδή δύο περιπτώσεις:

*: `double` × *Vector3* ↛ *Vector3*

*: *Vector3* × `double` ↛ *Vector3*

Θα κάνουμε λοιπόν διπλή επιφόρτωση του "*":

```
Vector3 operator*( double lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs * rhs.x;
   fv.y = lhs * rhs.y;
   fv.z = lhs * rhs.z;
   return fv;
} // operator*

Vector3 operator*( const Vector3& lhs, double rhs )
{
   Vector3 fv;
   fv.x = rhs * lhs.x;
   fv.y = rhs * lhs.y;
   fv.z = rhs * lhs.z;
   return fv;
} // operator*
```

Ο "`*`" πρέπει να επιφορτωθεί άλλη μια φορά για το εσωτερικό γινόμενο:

`*`: *Vector3* × *Vector3* ↛ `double`

που υλοποιείται με την

```
double operator*( const Vector3& lhs, const Vector3& rhs )
{
   return lhs.x*rhs.x + lhs.y*rhs.y + lhs.z*rhs.z;
} // operator*( const Vector3&
```

Prj02.5 Ο Ενικός Τελεστής "-"

Αν έχουμε δηλώσει:

```
   Vector3 v1, v2;
```

και δώσουμε:

```
   v2 = -v1;
```

το διάνυσμα $\mathbf{v}_2$ είναι το αντίθετο του $\mathbf{v}_1$, δηλαδή: $\mathbf{v}_1 + \mathbf{v}_2 = \mathbf{0}$.

Για τον "`-`" έχουμε:

`-`: *Vector3* → *Vector3*

Στην §14.6.4 λέγαμε ότι επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή "`@`" με μια συνάρτηση

```
   Trv operator@( T rhs )
```

και είδαμε ήδη μια εφαρμογή αυτού του κανόνα στον "`-`" για τον *complex* (§15.5). Τώρα, *Trv* και *T* είναι ο *Vector3*:

```
Vector3 operator-( const Vector3& rhs )
{
   return Vector3( -rhs.x, -rhs.y, -rhs.z );
} // operator-( const Vector3&
```

Prj02.6 Οι Τελεστές Εκχώρησης[2]

Τώρα θα επιφορτώσουμε τους τρεις τελεστές εκχώρησης "`+=`", "`-=`", "`*=`" όπως είπαμε στην §14.6.3. Εκεί είδαμε ότι ο "+=" για τον τύπο *T* επιφορτώνεται ως:

```
T& operator+=( T& lhs, const T& rhs )
```

Προσεξε ότι ο τύπος της πρώτης παραμέτρου δεν έχει `const`.

Στην περίπτωσή μας *T* είναι ο *Vector3* οπότε έχουμε:

```
Vector3& operator+=( Vector3& lhs, const Vector3& rhs )
{
```

[2] Αργότερα θα μάθουμε ότι ο πάγιος τρόπος επιφόρτωσης αυτών των τελεστών είναι διαφορετικός.

```
   lhs.x += rhs.x;
   lhs.y += rhs.y;
   lhs.z += rhs.z;
   return lhs;
} // operator+=( Vector3&
```

Παρομοίως γίνεται η επιφόρτωση και των άλλων δύο τελεστών:

```
Vector3& operator-=( Vector3& lhs, const Vector3& rhs )
{
   lhs.x -= rhs.x;
   lhs.y -= rhs.y;
   lhs.z -= rhs.z;
   return lhs;
} // operator-=( const Vector3&

Vector3& operator*=( Vector3& lhs, double rhs )
{
   lhs.x *= rhs;
   lhs.y *= rhs;
   lhs.z *= rhs;
   return lhs;
} // operator*=( const Vector3&
```

Prj02.7 Ο Τελεστής "<<"

Έχουμε ήδη επιφορτώσει τον "<<" για αρκετους τύπους. Αντιγράφοντας σχεδόν την επιφόρτωση για τον τύπο *complex* (§15.5) έχουμε:

```
ostream& operator<<( ostream& tout, const Vector3& rhs )
{
   return tout << "(" << rhs.x << ", " << rhs.y << ", "
               << rhs.z << ")";
} // operator<<
```

Prj02.8 ... και το Ευκλείδιο Μέτρο

Για το ευκλείδιο μέτρο δεν έχουμε κάποιον βολικό (από οπτική άποψη) τελεστή. Θα γράψουμε λοιπόν μια:

```
double Vector3_abs( const Vector3& lhs )
{
   return sqrt( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs
```

Prj02.9 Το Πρόγραμμα

Έχοντας αυτά τα εργαλεία το πρόγραμμα είναι τετριμμένο. Για να διευκολύνουμε το γράψιμο (και για να κάνουμε οικονομία στις πράξεις) της τελευταίας ερώτησης ορίζουμε μια επιπλέον συνάρτηση:

```
double Vector3_abs2( const Vector3& lhs )
{
   return ( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs2
```

που για ένα διάνυσμα **a** μας δίνει το $|\mathbf{a}|^2$.

Αυτό που περιμένουμε να δούμε είναι ο μηδενισμός της παράστασης:

```
Vector3_abs2(a)*Vector3_abs2(b) - ((a*b)*(a*b)+Vector3_abs2(a^b))
```

Γράφουμε λοιπόν το πρόγραμμα:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
   Vector3( const Vector3& rhs  )
   {  x = rhs.x;  y = rhs.y;  z = rhs.z;  }
}; // Vector3

bool operator==( const Vector3& lhs, const Vector3& rhs );
bool operator!=( const Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& rhs );
Vector3 operator+( const Vector3& lhs, const Vector3& rhs );
Vector3& operator+=( Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& lhs, const Vector3& rhs );
Vector3& operator-=( Vector3& lhs, const Vector3& rhs );
Vector3 operator*( double lhs, const Vector3& rhs );
Vector3 operator*( const Vector3& lhs, double rhs );
Vector3& operator*=( Vector3& lhs, double rhs );
double operator*( const Vector3& lhs, const Vector3& rhs );
Vector3 operator^( const Vector3& lhs, const Vector3& rhs );
ostream& operator<<( ostream& tout, const Vector3& rhs );
double Vector3_abs( const Vector3& lhs );
double Vector3_abs2( const Vector3& lhs );

int main()
{
   double q( 1e-6 ); // Cb
   Vector3 v( 1e8, 0, 0 );
   Vector3 B( 0, 0, 10 );
   Vector3 FL;

   FL = q*( v ^ B );
   cout << FL << endl;
   cout << FL*v << " " << FL*B << endl;

   Vector3 a( 0.1, 0.2, 0.3 ), b( 0.2, 0.3, 0.4 );

   cout << a << "  " << b << endl;
   cout << B*( a ^ b) << endl;

   cout << Vector3_abs2(a)*Vector3_abs2(b) -
           ((a*b)*(a*b)+Vector3_abs2(a^b)) << endl;
   cout << Vector3_abs2(a)*Vector3_abs2(b) << endl;
} // main
```

Αποτέλεσμα:

```
(0, -1000, 0)
0 0
(0.1, 0.2, 0.3)  (0.2, 0.3, 0.4)
-0.1
-3.27294e-018
0.0406
```

Οι τελευταίες δύο γραμμές χρειάζονται ένα σχόλιο. Η διαφορά που περιμένουμε μηδέν βγαίνει περίπου $-3.3 \cdot 10^{-18}$. Αλλά το γινόμενο $|\mathbf{a}|^2|\mathbf{b}|^2$ έχει τιμή $0.0406 \approx 4 \cdot 10^{-2}$. Όπως βλέπεις, κατ' απόλυτη τιμή, η διαφορά είναι 10^{16} φορές μικρότερη. Μπορούμε λοιπόν να τη θεωρήσουμε μηδέν (0).

κεφάλαιο

16

Δυναμική Παραχώρηση Μνήμης

Ο στόχος μας σε αυτό το κεφάλαιο:

Να γνωρίσεις το σύστημα διαχείρισης δυναμικής μνήμης της C++. Το σύστημα αυτό είναι εξαιρετικώς ευέλικτο και δίνει τη δυνατότητα στον προγραμματιστή να χρησιμοποιεί τη δυναμική μνήμη –με πίνακες ή δυναμικές δομές δεδομένων– με πολύ αποδοτικό τρόπο. Το βέλος είναι το βασικό εργαλείο.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς δυναμικούς πίνακες και δυναμικές δομές δεδομένων στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *τελεστές "`new`", "`new[]`"*
- *τελεστές "`delete`", "`delete[]`"*
- *δυναμική μεταβλητή*
- *δυναμικός πίνακας*
- *μεταβλητή-βέλος (pointer)*
- *RAII*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Όπως λέγαμε στην §11.1 «*σε κάθε πρόγραμμα παραχωρούνται τρεις περιοχές μνήμης:*

- *η* ***στατική****, όπου υλοποιούνται οι καθολικές μεταβλητές* (*και μερικές άλλες που θα δούμε στη συνέχεια*),
- *η* ***αυτόματη*** (*automatic*) *ή μνήμη* ***στοίβας*** (*stack*)*, όπου υλοποιούνται οι τοπικές μεταβλητές των σύνθετων εντολών και των συναρτήσεων και*
- *η* ***δυναμική*** (*dynamic*) *μνήμη που θα δούμε στη συνέχεια.*»

Η ποσότητα *στατικής μνήμης* που θα χρησιμοποιηθεί καθορίζεται όταν γράφουμε το πρόγραμμά μας, όταν δηλώνουμε τις καθολικές και τις στατικές μεταβλητές που χρησιμοποιεί.

Η ποσότητα *μνήμης στοίβας* που θα χρησιμοποιηθεί εξαρτάται

- από τις μεταβλητές που δηλώνουμε στις διάφορες συναρτήσεις και
- από τον τρόπο που θα κληθούν οι συναρτήσεις και πόσες φορές.

Αν, ας πούμε, έχουμε δύο συναρτήσεις *f()* και *g()* και τις καλέσουμε από τη `main` με τις εντολές:

```
f( . . . );
. . .
g( . . . );
```

τότε η μέγιστη ποσότητα αυτόματης μνήμης που θα απαιτηθεί είναι αυτή της `main` και η μέγιστη από τις απαιτούμενες για τις *f()* και *g()*. Αν όμως η `main` καλεί την *f()* και αυτή καλεί τη *g()*, τότε θα χρειαστούμε μνήμη για τη `main` και μνήμη για την *f()* και μνήμη για τη *g()* ταυτοχρόνως.

Αν η *f()* είναι αναδρομική και καλέσει *n* φορές τον εαυτόν της τότε θα χρειαστούμε μνήμη για τη `main` και *n*+1 φορές μνήμη για την *f()*.

Η C++, όπως και άλλες γλώσσες προγραμματισμού, μας επιτρέπει «να παίρνουμε» πρόσθετη μνήμη –τη **δυναμική** μνήμη– σύμφωνα με ανάγκες που παρουσιάζονται *όταν το πρόγραμμά μας εκτελείται*. Επιτρέπει δηλαδή **δυναμική παραχώρηση μνήμης** (dynamic memory allocation).

Το πρότυπο της C++ αναφέρεται στη δυναμική μνήμη με δύο διαφορετικά ονόματα: Αποκαλεί

- **μνήμη σωρού** (heap memory) αυτήν που χειριζόμαστε με τις συναρτήσεις της C (*malloc*, *calloc*, *realloc*, *free*) και
- **free store** αυτήν που χειριζόμαστε με τους τελεστές `new` και `delete` της C++.

Έτσι, αν κάνεις ένα πείραμα σαν αυτό της §14.8, με αυτά που θα δούμε στο κεφάλαιο αυτό, μπορεί να δεις διαφορετικές περιοχές διευθύνσεων. Αυτό δεν έχει και μεγάλη σημασία αρκεί να τηρείς τον πολύ βασικό κανόνα, που θα επαναλάβουμε και στη συνέχεια: *Μνήμη που παίρνεις με* `new` *θα απελευθερώνεται με* `delete` *και μνήμη που παίρνεις με malloc, calloc, realloc θα απελευθερώνεται με free.*

Θα πρέπει πάντως να επισημάνουμε μια διαφορά της C++ από τις «αδελφές» της απογόνους της C, τη Java και τη C#: Η C++ κράτησε τον τρόπο χειρισμού της δυναμικής μνήμης με *μεταβλητές-βέλη* (pointers) αν και έχει (και) νέα εργαλεία πέρα από αυτά της C.

16.1 Οι Τελεστές "new" και "delete"

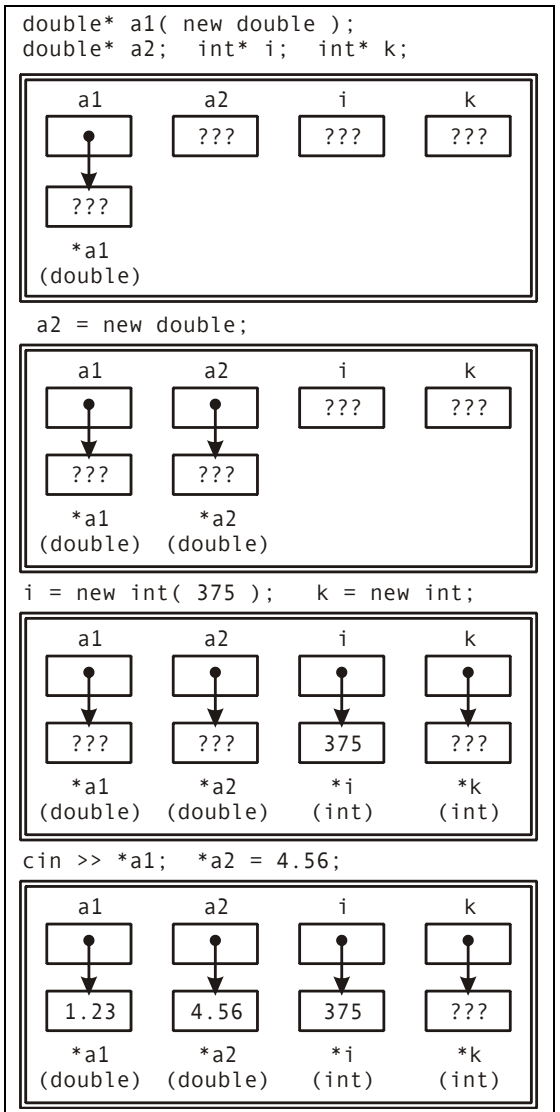
Η δυναμική παραχώρηση μνήμης γίνεται με τους τελεστές "`new`" και "`delete`" και με δύο κατηγορίες μεταβλητών:

- τις **μεταβλητές-βέλη** (pointer variables) –που ήδη ξέρουμε– και

- τις **δυναμικές** (dynamic) **μεταβλητές**.

Στις τιμές-βέλη αναφερθήκαμε για πρώτη φορά στο Κεφ. 2 (§2.8.3) λέγαμε ότι: «*γράφοντας* **&number** *παίρνουμε τη διεύθυνση μιας θέσης μνήμης όπου υπάρχει η πληροφορία που θέλουμε· παίρνουμε δηλαδή μια παραπομπή προς αυτό που μας ενδιαφέρει. Λέμε λοιπόν ότι η* **&number** *είναι μια* ***παραπέμπουσα*** *ή* ***αναφερόμενη*** (*referencing*) *τιμή –αφού παραπέμπει ή αναφέρεται σε κάτι– ή τιμή-**βέλος*** (*pointer*) *– αφού δείχνει κάτι.*» Στην ίδια παράγραφο λέγαμε ακόμη: «*Ας πούμε ότι έχουμε μια τιμή-βέλος p, δηλαδή μια διεύθυνση· πώς μπορούμε να δούμε την τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p; Ή, με άλλα λόγια, ποια είναι αντίστροφη πράξη της "&"; Η C++ τη συμβολίζει με "*****": η τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p παριστάνεται με "***p**". [...] Αν πάρουμε την* ***(&number)** *είναι σαν να παίρνουμε τη* **number**. *Λέμε ότι ο τελεστής "*****" **απο-παραμέμπει*** (*dereferences*) *την τιμή-βέλος στην οποία δρα.*»

Στο Κεφ. 12 είδαμε ότι «*Για τη C++, ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.*» Και (ενώ ένας πίνακας είναι ένα σταθερό βέλος) είδαμε ότι μπορούμε να έχουμε και μεταβλητές-βέλη. Πάντως, σε όλες τις περιπτώσεις οι τιμές και οι μεταβλητές-βέλη έδειχναν θέσεις συμβατικής (στατικής ή αυτόματης) μνήμης.



Σχ. 16-1 Δημιουργία δυναμικών μεταβλητών και εκχώρηση τιμών σε αυτές. Η δυναμική μεταβλητή δεν υπάρχει όταν το αντίστοιχο βέλος δεν έχει τιμή.

Τώρα ας δούμε μια άλλη δυνατότητα. Αν έχουμε δηλώσει:

```
T*   q;  ή  (T  *q;)
```

αν δηλαδή η *q* είναι μια *μεταβλητή-βέλος*, που δείχνει θέσεις μνήμης τύπου *T*, τότε η εκτέλεση της εντολής

```
q = new T;
```

έχει τα εξής αποτελέσματα:

- Παραχωρείται στο πρόγραμμά μας μια θέση μνήμης τύπου *T*. Αυτή είναι μια *δυναμική μεταβλητή* και μπορούμε να την χρησιμοποιούμε στο πρόγραμμά μας με το «όνομα» ***q**.
- Η *διεύθυνση* της ***q** είναι αποθηκευμένη ως τιμή της *q*.

Αν, αργότερα, δεν χρειαζόμαστε αυτή τη θέση, την «αναλυκλώνουμε» ή την «επιστρέφουμε» με την

```
delete q;
```

Πριν δώσουμε παράδειγματα όπου θα φαίνεται η χρησιμότητα της δυναμικής παραχώρησης μνήμης, θα εισαγάγουμε τα στοιχεία της γλώσσας που απαιτούνται με πιο απλά παραδείγματα.

Πλαίσιο 16.1

Τελεστής new

Αν έχουμε δηλώσει:

```
T*  pv;
```

1) η

```
pv = new T;
```

έχει ως αποτέλεσμα να παραχωρηθεί στο πρόγραμμα μια θέση μνήμης (μεταβλητή) τύπου *T*. Η μεταβλητή είναι η

```
*pv
```

και μπορούμε να τη διαχειριζόμαστε όπως οποιαδήποτε μεταβλητή τύπου *T*. Η διεύθυνση της θέσης που μας παραχωρήθηκε αποθηκεύεται ως τιμή της *pv*.

2) η

```
pv = new T[N];
```

έχει ως αποτέλεσμα να παραχωρηθούν στο πρόγραμμα *N* θέσεις μνήμης τύπου *T*, δηλαδή ένας *πίνακας* με *N* στοιχεία τύπου *T*. Τα στοιχεία του πίνακα είναι:

```
pv[0], pv[1],… pv[N-1]
```

και μπορούμε να τα διαχειριζόμαστε σαν να είχαμε δηλώσει στο πρόγραμμα:

```
T pv[N];
```

16.2 Συντακτικά και Βασικές Έννοιες

Ας πούμε ότι έχουμε κάποιον τύπο *T* και δηλώνουμε τη μεταβλητή:

```
T*  q;
```

Ή, αν θέλεις, ορίζουμε έναν τύπο:

```
typedef  T*   PT;
```

και δηλώνουμε:

```
PT  q;
```

Σε κάθε περίπτωση, η *q* είναι μια *μεταβλητή-βέλος*. Ο *T** ή ο *PT* είναι ένας **τύπος βέλους** με **τύπο στόχο** (target ή domain ή base type) τον *T*.

Όπως ξέρουμε *στην q αποθηκεύονται πληροφορίες που καθορίζουν μια διεύθυνση στην κύρια μνήμη του υπολογιστή*, συνήθως η ίδια η διεύθυνση. Στη διεύθυνση αυτήν, που μας δείχνει η τιμή της *q*, μπορεί να αποθηκευτεί μια τιμή τύπου *T*.

Για παράδειγμα, μετά τα

```
typedef  int*   PInt;
// . . .
   double *a1, *a2;
   PInt   i, j, k;
```

Οι *a*1, *a*2, *i*, *j*, *k* είναι μεταβλητές-βέλη· οι δύο πρώτες προς μεταβλητές τύπου `char`, οι τρεις τελευταίες προς μεταβλητές τύπου `int`.

Παρατήρηση: ▶

Πρόσεξε το εξής: Μετά τον ορισμό «*PInt* είναι ο `int*`», με τη δήλωση που κάνουμε, όλες οι μεταβλητές –*i*, *j*, *k* – είναι τύπου `int*`. Στην πρώτη δήλωση, αν γράφαμε "`double* a1, a2`" η *a*1 θα ήταν τύπου `double*` αλλά η *a*2 θα ήταν τύπου `double`.◀

Θα χρησιμοποιήσουμε αυτές τις (συμβατικές) μεταβλητές για να πάρουμε και να ελέγξουμε δυναμικές μεταβλητές.

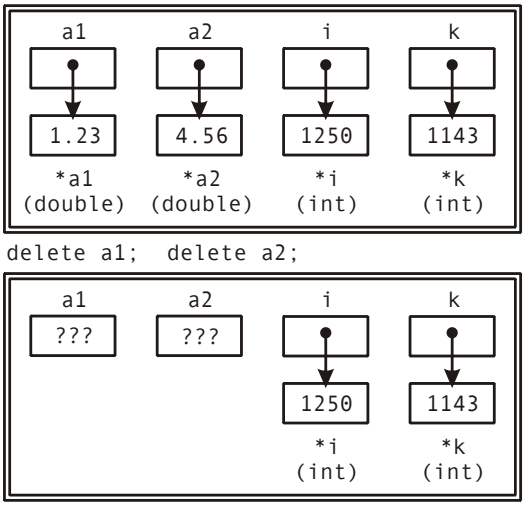
Μετά τη δήλωσή τους οι μεταβλητές αυτές είναι *αόριστες*. Θα τους δώσουμε τιμές με μια παράσταση **new** (Πλ. 16.1). Η πρώτη μορφή της παράστασης είναι γενικώς:

"`new`", *τύπος* ["`(`", *παράσταση*, "`)`"];

και επιστρέφει ένα βέλος προς μια μεταβλητή οργανωμένη όπως καθορίζει ο «*τύπος*». Αν υπάρχει η «*παράσταση*» θα πρέπει να δίνει αποτέλεσμα που μπορεί να μετατραπεί σε τιμή του τύπου της δυναμικής μεταβλητής. Στην πιο απλή περίπτωση μπορείς να δώσεις π.χ.:

```
k = new int;
```

Η μεταβλητη-βέλος *k* ορίζεται και δείχνει μια θέση μνήμης –μια δυναμική μεταβλητή– τύπου `int` που παραχωρήθηκε στο πρόγραμμά μας. Πώς τη χειριζόμαστε; Μα ως "`*k`"! *Μετά την εκτέλεση της εντολής ορίζεται η k αλλά η* `*`*k είναι αόριστη.*



Σχ. 16-2 Μετά τη `delete` η τιμή του βέλους είναι αόριστη και η αντίστοιχη δυναμική μεταβλητή δεν υπάρχει.

Μπορούμε να δώσουμε και

```
i = new int( 375 );
```

Τώρα, ορίζεται η μεταβλητη-βέλος *i* και δείχνει μια δυναμική μεταβλητή (`*`*i*) τύπου `int` αλλά είναι εξ αρχής ορισμένη και η `*`*i* που δημιουργήθηκε με αρχική τιμή "`375`".

Μπορούμε όμως να δώσουμε τη "`new`" και στη δήλωση:

```
double *a1( new double );
```

Τώρα η *a*1 είναι εξ αρχής ορισμένη. Φυσικά, μπορούμε να έχουμε εξ αρχής ορισμένη και τη δυναμική μεταβλητή (`*`*a*1) αν δώσουμε ως αρχική τιμή στο *a*1 "`new double(1.23)`".

Όλα αυτά μπορείς να τα δεις πιο παραστατικά στο Σχ. 16-1. Όπως βλέπεις, ενώ η *k*, ας πούμε, είναι μια *συμβατική μεταβλητή* η `*`*k* είναι πράγματι *δυναμική*. Οι μεταβλητές-βέλη των παραδειγμάτων υπάρχουν από τη στιγμή που αρχίζει να εκτελείται το πρόγραμμά μας (ή η συνάρτηση) όπου έχουν δηλωθεί. Αλλά οι δυναμικές μεταβλητές δεν υπάρχουν μέχρι να παραχωρηθούν από τον τελεστή `new`.

Αν έχουμε τύπο δομής, ας πούμε τον *Date*, μπορούμε να γράψουμε:

```
Date* pd( new Date(Date(2008, 7, 9)) );
```

Δηλαδή: δώσε μου μνήμη για μια δυναμική μεταβλητή τύπου *Date* (που θα τη δείχνει το βέλος *pd*) με αρχική τιμή αυτήν που δημιουργεί ο δημιουργός του τύπου *Date* αν τροφοδοτηθεί με τις τιμές 2008, 7, 9. Πάντως μπορείς να έχεις το ίδιο αποτέλεσμα και με πιο απλό γράψιμο:

```
Date* pd( new Date(2008, 7, 9) );
```

Κατά τα άλλα:

- ***Χειριζόμαστε τις δυναμικές μεταβλητές όπως όλες τις άλλες του ίδιου τύπου.***

Μπορούμε, για παράδειγμα, να τους δίνουμε τιμή ή να αλλάζουμε την τιμή που έχουν με εντολή εκχώρησης

```
*i = (*i)*2 + 500;
```

ή με εντολή εισόδου από αρχείο ή απο την οθόνη:

```
cin >> *k;
```

Για να μιλάμε όμως για δυναμική μνήμη δεν φτάνει μόνον η δυνατότητα παραχώρησης· πρέπει να έχουμε και δυνατότητα επιστροφής όταν δεν τη χρειαζόμαστε. Ο «αντίστροφος» του `new` είναι ο τελεστής `delete` (Πλ. 16.2).

Ας πούμε ότι μετά τη χρήση των δυναμικών μεταβλητών που βλέπουμε στο Σχ. 16-1 βάζουμε τις εντολές

```
delete a1;   delete a2;
```

Πλαίσιο 16.2

Τελεστής `delete`

Αν έχουμε δηλώσει:

```
T*  pv;
```

και έχουμε πάρει δυναμική μνήμη με την

```
pv = new T;
```

η

```
delete pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει η δυναμική μεταβλητή **pv* β) η *pv* να γίνει αόριστη.

Αν έχουμε πάρει δυναμική μνήμη με την:

```
pv = new T[N];
```

η

```
delete[] pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει ο δυναμικός πίνακας που έδειχνε η *pv* β) η *pv* να γίνει αόριστη.

Στο Σχ. 16-2 βλέπεις δυο στιγμιότυπα από τη συνέχεια της εκτέλεσης.

Οι a1 και a2 δεν είναι πια ορισμένες. Η "`delete a1`" «εξαφανίζει» τη θέση μνήμης (μεταβλητή) **a1* και η "`delete a2`" την **a*2. Τί θα πει «εξαφανίζει»; Οι **a1* και **a2* δεν ελέγχονται πια από το πρόγραμμά μας αλλά από το μηχανισμό διαχείρισης της δυναμικής μνήμης. Αυτό σημαίνει ότι η θέση αυτή μπορεί να παραχωρηθεί ξανά με μια επόμενη "`new` …".

Και τώρα προσοχή:

- ***Αμέσως μετά τη "`delete q`" η *q δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.***

Δυστυχώς, η C++ θα αφήσει την τήρηση αυτού του κανόνα στον προγραμματιστή και δεν θα αποπειραθεί να αποτρέψει μια τέτοια παρανομία.

16.3 Τιμές Βελών και Δυναμικών Μεταβλητών

Όπως ήδη ξέρεις, η C++ μας επιτρέπει να τυπώσουμε την τιμή ενός βέλους ή οποιαδήποτε διεύθυνση. Ας πάμε λοιπόν σε αυτά που είδαμε στο Σχ. 16-1 για να δούμε τι θα μπορούσαν να είναι εκείνα τα "???":

```
#include <iostream>
using namespace std;
int main()
{
   double* a1( new double );
   double* a2;  int* i;  int* k;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
```

Αποτέλεσμα; Κάτι σαν

```
0x3e2478  0x34  0x22ffa8  0x7c800000
```

Αν προσπαθήσεις να κάνεις αποπαραπομπές, για παράδειγμα με την:

```
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

είναι πολύ πιθανό να έχεις διακοπή της εκτέλεσης του προγράμματός σου.

Οι αποπαραπομπές γίνονται με ασφάλεια μόνον όταν έχεις ορίσει όλες τις μεταβλητές-βέλη:

```
   a2 = new double;
   i = new int( 375 );   k = new int;
```

```
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

Αποτέλεσμα κάτι σαν

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
0  1.66976e-307  375  4064328
```

Συγκρίνοντας τα τελευταία αποτελέσματα με τα αρχικά βλέπουμε τα εξής: Μετά τις "`new`"

- Οι διευθύνσεις που είναι αποθηκευμένες στα τέσσερα βέλη είναι «κοντινές», στην ίδια περιοχή της μνήμης.
- Οι αποπαραπομπές γίνονται με ασφάλεια αλλά, φυσικά, η μόνη τιμή που έχει νόημα είναι αυτή του `*`*i* (375).

Την πρώτη φορά, η μόνη διεύθυνση που είχε νόημα ήταν η τιμή της *a*1 που ήταν ορισμένη από τη δήλωση. Στις άλλες μεταβλητές-βέλη τι είχαμε; *Το τυχαίο περιεχόμενό τους ερμηνευμένο ως διεύθυνση*. Έτσι, όταν ζητούσαμε να γίνει μια αποπαραπομπή αν η υποτιθέμενη διεύθυνση ήταν έξω από τον *χώρο διευθύνσεων* του προγράμματος είχαμε διακοπή της εκτέλεσής του.

Σημείωση: ▶

Χώρος διευθύνσεων (address space) ενός προγράμματος είναι το σύνολο των διευθύνσεων που μπορεί «νομίμως» να χρησιμοποιήσει όταν εκτελείται. Στα σύγχρονα ΛΣ, που χρησιμοποιούν τεχνολογίες **εικονικής** ή **υπερβατικής** (virtual) μνήμης αυτές οι διευθύνσεις δεν είναι φυσικές αλλά εικονικές που απεικονίζονται στις φυσικές με κάποιον τρόπο[1]. Αυτό εξηγεί και το ότι σε διαφορετικές εκτελέσεις του ίδιου προγράμματος μπορεί να βλέπεις μερικές (εικονικές) διευθύνσεις –όπως εδώ η τιμής της *a*1– να παραμένουν ίδιες.

Δηλαδή, ολη η μνήμη που χρειάζεται το πρόγραμμά μας υπάρχει στον δίσκο. Στην κύρια μνήμη παραχωρείται μια μικρότερη περιοχή (που μπορεί να είναι και *πολύ μικρότερη*). Κατά την εκτέλεση του προγράμματος, αναλόγως των αναγκών, φορτώνονται στην κύρια μνήμη κομμάτια της μνήμης από τον δίσκο ενώ φορτωμένα κομμάτια φυλάγονται στον δίσκο· έχουμε δηλαδή **ανταλλαγές μνήμης** (memory swapping) .◀

Συνεχίζουμε με το πρόγραμμά μας: Δίνουμε τιμές σε όλες τις δυναμικές μεταβλητές:

```
   cin >> *a1;  *a2 = 4.56;  cin >> *k;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και βλέπουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Και τώρα πρόσεξε: Δίνουμε

```
   delete a1;  delete a2;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και παίρνουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Παρά τις "`delete`" οι *a*1 και *a*2 κρατούν τις τιμές τους και οι αποπαραπομπές τους δεν έχουν το παραμικρό πρόβλημα! Και να ήταν μόνο αυτό... Δίνουμε στη συνέχεια τα εξής:

```
   a1 = new double;
   cout << a1 << "  " << *a1 << endl;
   *a1 = 7.13;
   cout << *a2 << endl;
```

και παίρνουμε:

```
0x3e24e0  4.56
```

[1] Έτσι, μπορεί να δείς και τον όρο «virtual address space»

7.13

Με την "`a1 = new double`" η `*a1` υλοποιείται στην ίδια θέση με την `*a2` (που όμως έχουμε «καταργήσει»). Και φυσικά όταν δίνουμε τιμή στην `*a1` αλλάζουμε την τιμή (της καταργημένης) `*a2`.

Ας τα πάρουμε τώρα από την αρχή για να δούμε μερικά κρίσιμα «πρέπει» και «δεν πρέπει».

- *Δεν πρέπει να προσπαθήσεις να κάνεις αποπαραπομπή σε κάποιο βέλος αν δεν υπάρχει η αντίστοιχη δυναμική μεταβλητή-στόχος*. Πολύ περισσότερο, δεν πρέπει να προσπαθήσεις να αλλάξεις την τιμή της (ανύπαρκτης) δυναμικής μεταβλητής· αν η τυχαία τιμή που υπάρχει στο βέλος είναι «νόμιμη» διεύθυνση μπορεί να κάνεις ζημιά! Πώς αντιμετωπίζουμε πρακτικώς αυτό το πρόβλημα;
- *Σε κάθε μεταβλητή-βέλος που δηλώνεις πρέπει να δίνεις οπωσδήποτε αρχική τιμή. Αν δεν την ξέρεις δώσε τιμή* "`0`" ("`NULL`"). (Να υπενθυμίσουμε ότι στην §12.3.2 είδαμε και άλλους τρόπους –εκτός από μια παράσταση **new**– για να δώσουμε αρχική τιμή σε ένα βέλος. Εκεί είδαμε και την τιμή "`0`".)
- *Αν σε κάποιο σημείο του προγράμματός σου δεν έχεις τη σιγουριά ότι μια μεταβλητή-βέλος p δείχνει «νόμιμο» στόχο, πριν κάνεις αποπαραπομπή, πρέπει να ελέγξεις με μια* "`if( p != 0 ) …`"
- *Μετά από μια* "`delete p`", *αν δεν δίνεις μια άλλη νόμιμη τιμή στο βέλος p, πρέπει να βάλεις οπωσδήποτε μια* "`p = 0`" (για να έχει νόημα ο έλεγχος που υποδεικνύεται πιο πάνω). Η "`delete p`"
  - δεν μηδενίζει αυτομάτως την τιμή της *p*,
  - δεν «καθαρίζεται» η τιμή της δυναμικής μεταβλητής `*p`· στο σημείο αυτό θα επανέλθουμε.[2]

16.4 Δυναμικοί Πίνακες

Το ενδιαφέρον της δυναμικής παραχώρησης μνήμης δεν βρίσκεται στο να πάρουμε μια μεταβλητή τύπου `int` όταν εκτελείται το πρόγραμμα αλλά

1. στη δυνατότητα υλοποίησης δυναμικών δομών στοιχείων (λίστες, δένδρα κλπ)
2. στη διαχείριση μεγάλων πινάκων.

Εδώ θα ασχοληθούμε με τη δεύτερη περίπτωση που χρειάζεται τη δεύτερη μορφή των **new** και **delete**. Έστω ότι στο πρόγραμμά μας δηλώνουμε:

```
double*  p( 0 );
```

Αν στη συνέχεια ζητήσουμε:

```
p = new double[100];
```

θα πάρουμε μια περιοχή μνήμης με 100 θέσεις τύπου `double` που μπορούμε να τις διαχειριστούμε σαν να είχαμε δηλώσει:

```
double p[100];
```

με την εξής διαφορά: αντί για το 100, θα μπορούσαμε να είχαμε βάλει στη `new` οποιαδήποτε παράσταση που θα μας έδινε μια θετική ακέραιη τιμή. Αυτή *η τιμή καθορίζεται τη στιγμή που εκετελείται η εντολή και όχι όταν μεταγλωττίζεται*, όπως γίνεται με τη δήλωση ενός συμβατικού πίνακα.

---

[2] Δηλαδή η "`delete`", παρά το όνομά της, δεν κάνει διαγραφές· απλώς παραδίδει στην *ανακύκλωση* τη δυναμική μνήμη που είχαμε σε χρήση.

Παρατήρηση: ▶
«Θετική ακέραιη τιμή»! Δηλαδή δεν μπορεί να είναι μηδέν; Η C++ δεν έχει αντίρρηση αλλά υπάρχουν κάτι προβληματάκια που μπορεί να μας έλθουν από τη C. Τα συζητούμε παρακάτω.◀

Χειριζόμαστε τον δυναμικό πίνακα όπως τους συμβατικούς. Για παράδειγμα τα στοιχεία του p είναι τα p[0], p[1], …, p[99] και χειριζόμαστε το κάθε ένα από αυτά όπως μια μεταβλητή τύπου `double`.

Όταν δεν μας χρειάζεται η μνήμη, μπορούμε να τη επιστρέψουμε με την εντολή:

```
delete[] p;
```

Αλλά εδώ υπάρχει κάτι που χρειάζεται ιδιαίτερη προσοχή: δεν επιτρέπεται να ανακατεύεις τις δύο μορφές των **new** και **delete**.

- ***Όταν παίρνεις δυναμική μνήμη με "`p = new T`" θα την επιστρέφεις με "`delete p`" και όταν παίρνεις δυναμική μνήμη με "`p = new T[...]`" θα την επιστρέφεις με "`delete[] p`".***

Έτσι, *θα ήταν λάθος αν προσπαθούσαμε να επιστρέψουμε τον δυναμικό πίνακα* p *με μια* "`delete p`". Αν πάλι είχαμε δηλώσει

```
double q( new double );
```

είναι λάθος να προσπαθήσουμε να ανακυκλώσουμε την `*q` *με* "`delete[] q`".

Ιδιαίτερο ενδιαφέρον έχουν οι πολυδιάστατοι δυναμικοί πίνακες. Θα τα πούμε στη συνέχεια. Προς το παρόν ας ξαναδούμε ένα παράδειγμα από τα παλιά.

Παράδειγμα
Θα ξαναγράψουμε το πρόγραμμα που «μετράει» το αρχείο alturing.txt, που είδαμε στο Παράδ. 1 της §8.10, *αλλά*

- αφού φορτώσουμε το περιεχόμενο του αρχείου στη μνήμη (§15.12.1),
- σε δυναμικό πίνακα.

Κατ' αρχάς θα γράψουμε μια

```
void loadText( string flNm, char*& toText, size_t& tLength )
```

που τροφοδοτείται με το όνομα ενός αρχείου (*flNm*), φορτώνει το περιεχόμενό του σε έναν δυναμικό πίνακα (με στοιχεία τύπου `char`) και μας επιστρέφει βέλος προς την αρχή του πίνακα (*toText*) και το πλήθος στοιχείων του πίνακα (*tLength*). Η συνάρτηση θα ρίχνει εξαίρεση αν δεν μπορεί να ανοίξει ή να διαβάσει το αρχείο ή αν δεν μπορεί να πάρει δυναμική μνήμη.

```
void loadText( string flNm, char*& toText, size_t& tLength )
{
   ifstream tin( flNm.c_str(), ios_base::binary|ios_base::ate );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotOpen,
                        flNm.c_str() );
   tLength = tin.tellg();
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
   tin.seekg( 0 );
   tin.read( toText, tLength );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotRead,
                        flNm.c_str() );
   tin.close();
   toText[tLength] = '\0';
} // loadText
```

Σε σχέση με αυτά που είδαμε στην §15.12.1 το νέο στοιχείο είναι η δυναμική μνήμη:

```
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
```

```
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
```

Γιατί παίρνουμε *tLength*+1; Για να χωρέσει και ο φρουρός (`'\0'`) που βάζουμε στην τελευταία (παραπανίσια) θέση.

Σε μια άλλη συνάρτηση βάζουμε την επεξεργασία (μέτρημα):

```
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits )
{
   int pos( 0 );
   char ch;      // χαρακτήρας που διαβάζουμε
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
// Επεξεργάσου το αρχείο
   ch = toText[pos];
   while ( pos < tLength )
   {
      while ( pos < tLength && ch != '\n' )
      {
         if ( isupper(ch) ) ++nUpCase;
         else if ( isdigit(ch) )  ++nDigits;
         ++pos;  ch = toText[pos];
      }
      ++nRows;
      if ( pos < tLength )  {  ++pos;  ch = toText[pos];  }
   } // while
} // countText
```

Πρόσεξε τα εξής:

- Αν *pos* είναι ο δείκτης του χαρακτήρα *ch* που επεξεργαζόμαστε (toText`[`*pos*`]` `==` *ch*) τότε η πρώτη "`t.get(ch)`" γίνεται: "`pos = 0; ch = toText[pos];`" και η επαναλαμβανόμενη: "`++pos; ch = toText[pos]`".
- Η "`!t.eof()`" γίνεται "`pos < tLength`". Θα μπορούσε να γίνει και "`ch != '\0'`"; Ναι, αλλά η πρώτη είναι προτιμότερη αφού δουλεύει και στην περίπτωση που υπάρχουν `'\0'` μέσα στο αρχείο. Να τονίσουμε, βέβαιως, ότι δεν περιμένουμε να βρούμε τέτοιους χαρακτήρες σε ένα αρχείο text.
- Ο έλεγχος τέλους γραμμής παραμένει: "`ch != '\n'`". Θα πρέπει να τον αλλάξουμε αν έχουμε αρχείο text που οι γραμμές του διαχωρίζονται από ακολουθία χαρακτήρων που δεν περιέχει τον `'\n'`.
- Αυτή η συνάρτηση δεν ρίχνει κάποια εξαίρεση; Θα μπορούσαμε να βάλουμε ελέγχους όπως "`toText == 0`" ή "`tLength < 0`" αλλά για ένα τόσο απλό πρόγραμμα δεν έχει νόημα.

Έχοντας δηλώσει:

```
   char*  toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t tLength;     // μήκος κειμένου
   int    nRows;       // μετρητής γραμμών
   int    nUpCase;     // μετρητής κεφαλαίων
   int    nDigits;     // μετρητής ψηφίων
```

καλούμε τις δύο συναρτήσεις ως εξής:

```
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
```

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <new>

using namespace std;
```

```
struct ApplicXptn
{
   enum { cannotOpen, cannotRead, allocFailed };
   char funcName[100];
   int  errCode;
   char errStrVal[100];

   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ApplicXptn

void loadText( string flNm, char*& toText, size_t& tLength );
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits );

int main()
{
   char*     toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t    tLength;     // μήκος κειμένου
   int       nRows;       // μετρητής γραμμών
   int       nUpCase;     // μετρητής κεφαλαίων
   int       nDigits;     // μετρητής ψηφίων

   try
   {
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
      delete[] toText;  toText = 0;
   // Λέγε τα αποτελέσματα
      cout << " Διάβασα " << nRows << " γραμμές" << endl;
      cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
           << nDigits << " ψηφία" << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

16.5 * Η Τρίτη Μορφή του "new"

Υπάρχει και μια τρίτη μορφή χρήσης του τελεστή "**new**" που δεν έχει σχέση με δυναμική παραχώριση μνήμης. Δες ένα παράδειγμα: Οι εντολές

```
   int buf[100];
   double* d( new (buf) double );
   cout << &buf[0] << endl;
   cout << d << endl;
```

δίνουν:

```
0x22fde0
0x22fde0
```

Δηλαδή: η μεταβλητή **d* δημιουργείται στην αρχή του πίνακα buf.

Όπως φαίνεται και από το παράδειγμα, αυτή η μορφή του "**new**" έχει ως αποτέλεσμα *την δημιουργία μιας μεταβλητής σε μια συγκεκριμένη θέση της μνήμης*· δεν υπάρχει παραχώρηση δυναμικής μνήμης στο πρόγραμμά μας και επομένως δεν υπάρχει αντίστοιχη μορφή "**delete**". Αυτή η μορφή λέγεται **new τοποθέτησης** (placement new).

Πρόσεξε ένα άλλο παράδειγμα:

```
   char* buf( new char[100] );
   double* d( new (buf) double );
// . . .
   delete[] buf;
```

Εδώ γιατί βάλαμε **delete**; Για να ανακυκλώσουμε τη buf που πήραμε με **new** *δεύτερης μορφής* ("**new char[100]**"). Φυσικά, ανακυκλώνεται και η **d*, αφού δημιουργήθηκε στην αρχή του πίνακα buf.

Σε μικρά μονοχρηστικά συστήματα μπορείς να χρησιμοποιήσεις αυτήν τη μορφή της "**new**" για να βάλεις μεταβλητές κατάλληλου τύπου σε συγκεκριμένες διευθύνσεις για να διαχειριστείς με το πρόγραμμά σου ενταμιευτές συσκευών ή σχετικούς καταχωρητές. Αν δουλεύεις σε Linux ή Windows ή Unix ή άλλο παρόμοιο σύστημα μη διανοηθείς να χρησιμοποιήσεις απόλυτες διευθύνσεις μνήμης. Όπως είπαμε και πιο πριν οι διευθύνσεις που βλέπει το πρόγραμμά σου είναι **εικονικές**.

16.6 Η Εξαίρεση *bad_alloc*

Οι σημερινοί υπολογιστές διαθέτουν τεράστιες ποσότητες μνήμης (σε σύγκριση με αυτά που ξέραμε μέχρι πριν από λίγα χρόνια). Με τα συστήματα εικονικής μνήμης τα όρια φτάνουν τη χωρητικότητα των δίσκων σου. Παρ' όλα αυτά, ένα καλό πρόγραμμα θα πρέπει να είναι προετοιμασμένο για την περίπτωση που θα αποτύχει μια **new**, δηλαδή δεν θα μας δοθεί η μνήμη που ζητάμε. Στην περίπτωση αυτή ρίχνεται μια εξαίρεση τύπου *bad_alloc*.

Αυτό σημαίνει ότι η **new** θα πρέπει να βρίσκεται πάντοτε μέσα σε μια ομάδα **try** που θα ακολουθείται από μια "**catch(bad_alloc&)**". Για να μπορέσεις να χρησιμοποιήσεις τη *bad_alloc* στο πρόγραμμά σου θα πρέπει να έχεις βάλει μια:

```
#include <new>
```

Κατ' αρχήν λοιπόν έχουμε ένα σχέδιο της μορφής:

```
// . . .
#include <new>
// . . .
   try
   {
      double dr( new double[1000] );
// . . .
   }
// . . .
   catch( bad_alloc& )
   {
```

```
   // διαχείριση της εξαίρεσης
   }
// . . .
```

Στη συνέχεια θα βλέπεις να κάνουμε μια κάπως διαφορετική αντιμετώπιση: Θα πιάνουμε την εξαίρεση *bad_alloc* και θα ρίχνουμε μια δική μας,

```
// . . .
   double dr( 0 );
   try {  dr = new double[1000];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "thisFunc", MyProgXptn::allocFailed );  }
// . . .
```

Και τι θα μπορούσαμε να κάνουμε όταν πιάσουμε μια τέτοια εξαίρεση; Πιθανότατα να ανακυκλώσουμε κάποιον (ή κάποιους) τεράστιο πίνακα (-ες) που δεν μας χρειάζονται πια.

16.6.1 Μια Εξήγηση για τις Εξαιρέσεις μας

Τώρα μπορούμε να εξηγήσουμε και ένα χαρακτηριστικό των δικών μας εξαιρέσεων που μπορεί να σου φαίνεται περίεργο.

Γιατί δηλώνουμε

```
   char        funcName[100];
```

και όχι:

```
   string      funcName;
```

όπως συνήθως; Διότι ένα από τα προβλήματα που έχουμε να διαχειριστούμε είναι η έλλειψη δυναμικής μνήμης (*allocFailed*). Η *string* χρησιμοποιεί *δυναμική* μνήμη για να αποθηκεύσει το κείμενο. Αν λοιπόν το πρόβλημά μας είναι ότι *δεν μπορεί να παραχωρηθεί δυναμική μνήμη* και προσπαθήσουμε να ρίξουμε *εξαίρεση που θα χρειαστεί δυναμική μνήμη* – που πιθανότατα δεν θα μπορεί να πάρει– θα προκαλέσουμε «βίαιη» διακοπή της εκτέλεσης του προγράμματός μας. Χρησιμοποιώντας πίνακα τύπου `char` που υλοποιείται στη στοίβα τα πράγματα είναι ασφαλή.

16.7 Τα Προβλήματα της Δυναμικής Μνήμης

Η παραχώρηση δυναμικής μνήμης είναι ένα πολύ καλό εργαλείο αλλά έχει και ορισμένα προβλήματα η αντιμετώπιση των οποίων χρειάζεται την προσοχή μας όταν γράφουμε το πρόγραμμα.

Ξεκινάμε αντιγράφοντας –με ορισμένες προσαρμογές– ένα σχήμα και μερικά πράγματα από την §12.3.2:

Ας πούμε ότι δηλώνουμε:

```
   int* a( new int(10) );
   int* b( new int(20) );
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 16-3(α): το βέλος *a* δείχνει την `*a`, που έχει τιμή 10, και το βέλος *b* δείχνει την `*b` που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή "`*a = *b`". Η εικόνα της μνήμης είναι αυτή του Σχ. 16-3 (β). Η τιμή της `*a` γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

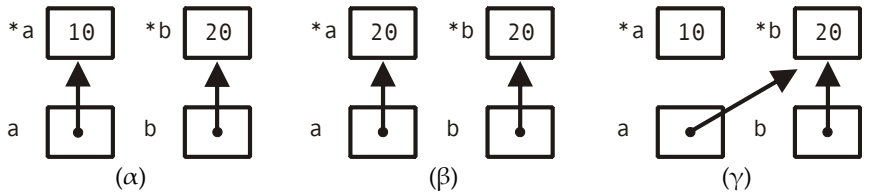
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: "`a = b`". Οι τιμές των μεταβλητών `*a` και `*b` δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *a*· το βέλος *a* δείχνει εκεί που δείχνει και το *b*: τη `*b` (Σχ. 16-3 (γ)).

Φυσικά, και στις δύο περιπτώσεις, αν δώσουμε την εντολή:

```
   cout << *a << "  " << *b << endl;
```

θα πάρουμε αποτέλεσμα:

```
20  20
```



Σχ. 16-3 (τροποποίηση του Σχ. 12-3) Αρχικώς, μετά την εκτέλεση των δηλώσεων (με αρχικές τιμές) `int* a(new int(10)); int* b(new int(20));` έχουμε την εικόνα (α). Η εικόνα (β) προκύπτει αν στη συνέχεια εκτελεσθεί η εκχώρηση: `*a = *b`. Η εικόνα (γ) προκύπτει αν, μετά την (α), εκτελεσθεί η εκχώρηση `a = b`.

αλλά το «παρασκήνιο» είναι διαφορετικό· και στη δεύτερη περίπτωση δημιουργείται το εξής πρόβλημα: *το πρόγραμμά μας έχει μια δυναμική μεταβλητή, την* `*a`*, αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί* (*ούτε να την ανακυκλώσει*).

Αυτό που περιγράψαμε πιο πάνω, είναι γνωστό ως **απώλεια** ή **διαρροή μνήμης** (memory leakage). Μπορεί να εμφανισθεί στις εξής περιπτώσεις:

- Ένα βέλος *p* είναι το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης και χωρίς να την ανακυκλώσουμε, αλλάζουμε την τιμή του *p*, όπως είδαμε παραπάνω.
- Ένα βέλος *p*, όντας το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης, είναι τοπική μεταβλητή σε μια συνάρτηση. Η εκτέλεση της συνάρτησης τελειώνει χωρίς να ανακυκλώσουμε τη δυναμική μνήμη το βέλος *p* «χάνεται». Αυτό φαίνεται στο παρακάτω παράδειγμα.

Έστω ότι έχουμε τη συνάρτηση:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
} // f
```

Ας πούμε λοιπόν ότι

- καλείται η *f*,
- εκτελείται επιτυχώς η "`pi = new int[ni]`" και παίρνει μνήμη για τον pi, αλλά
- αποτυγχάνει να πάρει μνήμη για τον pd και ρίχνεται μια *MyProgXptn* από τη δεύτερη `throw`.
- Διακόπτεται η εκτέλεση της *f* και παύουν να υπάρχουν τα βέλη *pi* και *pd* (επιστρέφονται στη στοίβα).

Έτσι, το πρόγραμμά μας έχει δεσμεύσει μνήμη για *ni* θέσεις τύπου `int` αλλά δεν έχει πρόσβαση σε αυτήν. Τι θα έπρεπε να κάνουμε; Πριν ρίξουμε την εξαίρεση θα έπρεπε να ανακυκλώσουμε τη μνήμη που ήδη πήραμε:

```
// . . .
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
```

Δεν τελειώσαμε όμως· αν μπορεί να εγερθεί εξαίρεση από αυτά που ακολουθούν θα πρέπει να ανακυκλώσουμε και τους δύο πίνακες:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try
   {
   // άλλες εντολές
   }
   catch( ... )
   {
      delete[] pi;
      delete[] pd;
      throw;
   }
} // f
```

Πρόσεξε την τελευταία `catch`:

- πιάνει οποιαδήποτε εξαίρεση μπορεί να προέλθει από τις «*άλλες εντολές*»,
- ανακυκλώνει τους δύο πίνακες και
- την ξαναρίχνει!

Ένα άλλο πρόβλημα, από μια άποψη το «αντίστροφο» αυτού που είδαμε, είναι το πρόβλημα του **μετέωρου βέλους** (pending ή dangling pointer). Ας πούμε ότι είμαστε στην κατάσταση (γ) του Σχ. 16-3 και δίνουμε

```
   delete b;  b = 0;
```

Όπως βλέπεις, ανακυκλώνουμε τη **b* και –τηρώντας τον κανόνα που βάλαμε– μηδενίζουμε το βέλος *b*. Καλά όλα αυτά, αλλά η **b* ήταν ταυτοχρόνως και **a*· έτσι τώρα το βέλος *a* είναι *μετέωρο* αφού δείχνει σε θέση μνήμης που δεν ανήκει στο πρόγραμμά μας.

Και αυτό το πρόβλημα μπορεί να προέλθει από απρόσεκτο γράψιμο συναρτήσεων. Δες την παρακάτω:

```
void g( double* dAr, int n )
{
   int nn;
// . . .
   if ( n < nn )
   {
      double* dArL( new double[nn] );
// . . .
      delete[] dAr;  dAr = dArL;
      n = nn;
   } // if
// . . .
} // g
```

Αυτή η συνάρτηση τροφοδοτείται με έναν δυναμικό πίνακα. Σε κάποιο σημείο μπορεί να ανακαλύψει ότι το μέγεθος του πίνακα δεν είναι κατάλληλο και τον αλλάζει.

Προσοχή! Η παράμετρος-βέλος *dAr* είναι παράμετρος τιμής. Το ότι δεν έχουμε βάλει "`const`" μας επιτρέπει να αλλάζουμε τις τιμές των στοιχείων του πίνακα αλλά το *dAr* είναι αντίγραφο.της πραγματικής παραμέτρου· αν αλλάξουμε την τιμή του βέλους η αλλαγή είναι τοπική μέσα στη συνάρτηση.

Ας πούμε λοιπόν ότι καλούμε τη συνάρτηση ως εξής:

```
   double* bd( 0 );
   int bn( 0 );
// . . .
```

```
g( bd, bn );
```

Όταν αρχίσει η εκτέλεση της συνάρτησης το βέλος *dAr* είναι αντίγραφο του *bd* και έτσι δείχνουν και τα δύο την ίδια θέση της μνήμης. Αν η συνάρτηση βρει ότι το μέγεθος του πίνακα δεν είναι αρκετό παίρνει την κατάλληλη μνήμη (με το βέλος *dArL*) και ανακυκλώνει την παλιά ("`delete[] dAr`") –αφού, πιθανότατα, αντιγράψει το περιεχόμενό της.

- Στην περίπτωση αυτήν το *bd*, που έδειχνε τη μνήμη που ανακυκλώθηκε, είναι πια *μετέωρο*.
- Μετά το τέλος εκτέλεσης της συνάρτησης το *τοπικό* βέλος *dArL* χάνεται (επιστρέφει στη στοίβα) και δεν έχουμε εργαλείο για να χειριστούμε τη δυναμική μνήμη που έδειχνε. Έχουμε δηλαδή *διαρροή μνήμης*.

Το πρόβλημα εξαφανίζεται αν περάσουμε τον δυναμικό πίνακα με *παραμέτρους αναφοράς*:

```
void g( double*& dAr, int& n )
```

Τα επόμενα προβλήματα έχουν σχέση με τον τελεστή "`delete`".

Το πρώτο είναι αυτό που ήδη είπαμε (§16.4): *Η μορφή του "`delete`" με την οποίαν θα ανακυκλώνουμε τη δυναμική μνήμη κάθε φορά πρέπει να είναι αντίστοιχη της μορφής του "`new`" που χρησιμοποιήσαμε για την παραχώρησή της.* Αν παραβιάσεις αυτόν τον κανόνα μπορεί να κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης (heap).

Δεν υπάρχει κάποια «συνταγή» για να διασφαλίσεις την τήρηση αυτού του κανόνα. Η συνηθισμένη σχετική υπόδειξη λέει: *Για κάθε κομμάτι δυναμικής μνήμης που παίρνεις φρόντισε η παράσταση* **new** *και η αντίστοιχη παράσταση* **delete** *να βρίσκονται στην ίδια συνάρτηση.* Αν φροντίζεις να γράφεις και μικρές συναρτήσεις τότε κάτι μπορεί να γίνει... Πάντως, όπως θα καταλάβεις και από την πείρα σου, όταν γράφεις προγράμματα με δυναμική μνήμη, δεν είναι και τόσο απλό να τηρείς αυτόν τον κανόνα.

Ένα άλλο πρόβλημα με τη "`delete`" είναι το εξής: *αν, κατά λάθος φυσικά, αποπειραθείς να ανακυκλώσεις μνήμη που έχεις ήδη ανακυκλώσει* (και επομένως δεν ελέγχεται από το πρόγραμμά σου) *κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης*. Φυσικά, δεν λέει κανείς ότι θα πας να γράψεις στο πρόγραμμά σου

```
delete p;
delete p;
```

Είναι όμως πολύ πιθανό να γράψεις κάτι σαν:

```
if ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
while ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
f( ..., p, ... );
delete p;
```

και μέσα στην *f* εκτελείται άλλη μια "`delete p`".

Αν τηρείς τον κανόνα «*μηδένισε το βέλος μετά τη* `delete`» γλυτώνεις από αυτό το πρόβλημα. Αν έχουμε:

```
delete p;  p = 0;
```

```
delete p;
```

η δεύτερη "`delete p`" είναι στήν πραγματικότητα "`delete 0`" και

- ***Η "`delete 0`" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "`delete[] 0`".***

Το τελευταίο που έχουμε να πούμε είναι το εξής: *Μην προσπαθήσεις να ανακυκλώσεις μνήμη που δεν είναι δυναμική*. Θα δείξουμε πώς μπορεί να συμβεί κάτι τέτοιο με ένα παράδειγμα. Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση που θα τροφοδοτείται με έναν πίνακα a με στοιχεία τύπου `int` και μια τιμή *v* τύπου `int`. Η συνάρτηση θα ψάχνει να βρει τη *v* μέσα στον a και αν δεν τη βρει θα την εισάγει. Τι θα κάνεις αν ο πίνακας είναι γεμάτος;

- Αν *ο πίνακας δεν είναι δυναμικός* η συνάρτηση θα πρέπει να βγάζει ένα μήνυμα – πιθανότατα μια εξαίρεση– για το πρόβλημα.
- Αν *ο πίνακας είναι δυναμικός* η συνάρτηση θα προσπαθήσει να τον «μεγαλώσει». Όπως θα δούμε στη συνέχεια, αυτό σημαίνει αντιγραφή σε έναν μεγαλύτερο πίνακα και ανακύκλωση του παλιού.

Όταν γράφεις τη συνάρτηση δεν ξέρεις με τι είδους πίνακες θα χρησιμοποιείται και αν ξεχάσεις την πρώτη απο τις παραπάνω δυνατότητες είναι πολύ πιθανό να κάνεις προσπάθεια να ανακυκλώσεις μη δυναμικό πίνακα.

Πώς λύνεται το πρόβλημα; Με διαχωρισμό των στόχων:

- Γράψε μια συνάρτηση που αναζητεί μια τιμή σε έναν πίνακα είτε αυτός είναι δυναμικός είτε όχι.
- Γράψε μια συνάρτηση που εισάγει μια τιμή σε έναν πίνακα που *δεν είναι γεμάτος* είτε αυτός είναι δυναμικός είτε όχι.

Αυτές μπορεί να είναι συναρτήσεις γενικής χρήσης· μπορεί να είναι και περιγράμματα.

Ανάμεσα στις κλήσεις αυτών των δύο συναρτήσεων βάλε τις εντολές –που εξαρτώνται από το πρόβλημα– για τις κατάλληλες ενέργειες όταν ο πίνακας είναι γεμάτος.

Αργότερα, όταν θα δούμε την STL, θα δούμε ότι υπάρχουν εργαλεία που μας απαλλάσσουν από αυτά τα προβλήματα. Τέτοια είναι:

- τα **έξυπνα βέλη** (smart pointers) που δεν έχουν αυτά τα προβλήματα και
- το *vector* και τα άλλα περιγράμματα **περιεχόντων** (containers).

16.7.1 RAII: Μια Καλύτερη Λύση

Τώρα, θα ξαναγυρίσούμε στη συνάρτηση *f* που είδαμε ως παράδειγμα για τη διαρροή μνήμης μέσα σε συνάρτηση· ναι αυτήν με τις τρεις **try-catch**! Θα τη χρησιμοποιήσουμε τώρα ως παράδειγμα για να παρουσιάσουμε μια άλλη λύση, σαφώς καλύτερη, με μια πάγια τεχνική της C++ που ονομάζεται «*Resource Acquisition Is Initialization*» (RAII), δηλαδή: *πρόσκτηση πόρων σημαίνει εκκίνηση*.

Πριν προχωρήσεις καλό θα είναι να κάνεις μια επανάληψη σε όσα είπαμε –έστω και ακροθιγώς– για δημιουργούς (§15.3) και καταστροφείς (§15.3.1).

Για το παράδειγμά μας τώρα, ορίζουμε έναν τύπο ως εξής:

```
struct IntDArr
{
   int* da;
   IntDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "IntDArr", MyProgXptn::allocFailed );  }
   }
   ~IntDArr()  {  delete[] da;  }
}; // IntDArr
```

Κάθε μεταβλητή αυτού του τύπου «κρύβει» μέσα της έναν δυναμικό πίνακα με στοιχεία τύπου `int`. Μέσα στη συνάρτηση, αντί για "`int* pi(0)`" βάζουμε τη δήλωση:

```
   IntDArr    pi( ni );
```

Για την εκτέλεσή της καλείται ο δημιουργός που παίρνει τη μνήμη που απαιτείται για τον δυναμικό πίνακα `pi.da`.

Σημείωση: ▶

Εδώ όμως έχουμε και καινούρια πράγματα: *Εξαίρεση από δημιουργό*! Ναι! Αν κάτι «πάει στραβά» (δεν πήραμε μνήμη) ρίχνεται εξαίρεση και *δεν δημιουργείται το αντικείμενο*!

Αυτό βέβαια μας βάζει και ιδέες· δεν θα μπορούσαμε να βάλουμε ελέγχους στον δημιουργό της *Date* –ας πούμε– και να ρίχνουμε εξαίρεση αν μας δώσουν μήνα 37; Ναι και θα το κάνουμε αργότερα. ◀

Στη συνέχεια, μπορούμε να χρησιμοποιούμε τον πίνακα, αλλά για το στοιχείο *k* θα πρέπει να γράφουμε `pi.da[k]` αντί για `pi[k]`. Αν διακοπεί η εκτέλεση της συνάρτησης για κάποιο λόγο –είτε διότι φτάσαμε στο τέλος της είτε λόγω εξαίρεσης– η μεταβλητή `pi` θα καταστραφεί με *αυτόματη κλήση του καταστροφέα*. Ο καταστροφέας, όπως βλέπεις, ανακυκλώνει τον πίνακα και έτσι δεν έχουμε διαρροή μνήμης.

Πολύ ωραία, αλλά για να δούμε την εξής περίπτωση: Ας πούμε ότι κάποια από τις παραμέτρους της συνάρτησης που δεν βλέπουμε είναι "`int*& ipp`" και –αν όλα πάνε καλά– θέλουμε να δείχνει στον νέο πίνακα (που έτσι θα «επιζήσει» μετά την εκτέλεση της συνάρτησης). Κανένα πρόβλημα:

```
   int* psv( pi.da );
   pi.da = ipp;
   ipp = psv;
```

Δηλαδή: αντιμεταθέτουμε τις τιμές των βελών `pi.da` και `ipp` και έτσι το `ipp` δείχνει τον νέο δυναμικό πίνακα ενώ, όταν καταστραφεί η `pi`, θα ανακυκλωθεί ο πίνακας που έδειχνε αρχικώς η `ipp` (αν δεν είχε τιμή 0).

Για να χειριστούμε παρομοίως και τον δυναμικό πίνακα τύπου `double` ορίζουμε:

```
struct DoubleDArr
{
   int* da;
   DoubleDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "DoubleDArr", MyProgXptn::allocFailed ); }
   }
   ~DoubleDArr()  {  delete[] da;  }
}; // DoubleDArr
```

και δες πώς γίνεται η *f*:

```
void f( int ni, int nd, . . . )
{
   IntDArr    pi( ni );
   DoubleDArr pd( nd );

// . . . όπως πριν αλλά
// αντικαθιστώντας το κάθε pi[k] με pi.da[k] και
// το κάθε pd[k] με pd.da[k]
} // f
```

Αγνώριστη και πολύ καλύτερη!

Αργότερα, όταν εξοικειωθείς περισσότερο με δημιουργούς και καταστροφείς θα κατάνοήσεις καλύτερα την τεχνική και θα καταλάβεις ότι είναι πολύτιμη σε πολλές περιπτώσεις.

Οι *IntDArr* και *DoubleDArr* λέγονται **δομές** (ή **κλάσεις**) **περιτυλίγματος** (wrapper structures ή classes). Θα τις ξαναδούμε και για άλλες δουλειές.

16.8 Προβλήματα και στις Δομές

Τώρα θα επισημάνουμε άλλο ένα πρόβλημα αλλά θα αφήσουμε τη λύση του για αργότερα.

Ας πούμε ότι κάνουμε τις εξής αλλαγές στον τύπο *Address*:

```
struct AddressD
{
   char* country;
   char* city;
   int   areaCode;
   char* street;
   int   number;
}; // AddressD
```

με σκοπό να κάνουμε οικονομία και να μην σπαταλούμε τον ίδιο χώρο για την οδό "Κώ" και την οδό "Αγίου Κωνσταντίνου". Όλα δυναμικά!

Πρόσεξε τώρα: ας πούμε ότι έχουμε:

```
   AddressD a1, a2;
```

και –αφού πάρουμε την απαραίτητη δυναμική μνήμη– δίνουμε τιμές σε όλα τα μέλη του *a*1.

Τι αποτέλεσμα θα έχει η εκχώρηση "`a2 = a1`" στη συνεχεια; Τα τρία *βέλη* *a*1.*country*, *a*1.*city*, *a*1.*street* θα αντιγραφούν στα αντίστοιχα *βέλη* του a2 αλλά *δεν θα έχουμε αντιγραφές κειμένων*. Έτσι, θα έχουμε τρεις δυναμικούς πίνακες χαρακτήρων που ο καθένας του στοχεύεται από δύο βέλη. Αλλάζουμε την πόλη στο *a*1 αλλάζει και η πόλη στο *a*2· αλλάζουμε τη χώρα στο *a*2 αλλάζει και η χώρα στο *a*1. Σπανίως θέλουμε να συμβαίνει κάτι τέτοιο.

Η διόρθωση αυτής της συμπεριφοράς γίνεται με τη σωστή επιφόρτωση του τελεστή εκχώρησης. Αυτό θα το δούμε αργότερα.[3] Προς το παρόν, αν θέλεις να μην σπαταλάς μνήμη και να κάνεις εύκολα τη δουλειά σου, τη λύση τη ξέρεις: χρησιμοποίησε τον τύπο *string*.[4]

16.9 Δισδιάστατοι Δυναμικοί Πίνακες

Θα δούμε τώρα πώς μπορούμε να έχουμε δισδιάστατους δυναμικούς πίνακες.

Ας πούμε ότι μας δίνεται ένα μορφοποιημένο αρχείο (όνομα στον δίσκο `egr63e2.txt`) όπου υπάρχουν, σίγουρα, οι τιμές των στοιχείων (πραγματικοί) ενός δισδιάστατου πίνακα. Στην πρώτη γραμμή υπάρχουν δύο θετικοί ακέραιοι που δίνουν το πλήθος γραμμών και το πλήθος στηλών του πίνακα. Σε κάθε μια από τις επόμενες γραμμές του αρχείου υπάρχουν οι τιμές των στοιχείων μιας γραμμής του πίνακα. Θέλουμε να διαβάσουμε το αρχείο και να αποθηκεύσουμε το περιεχόμενό του σε έναν δυναμικό δισιδιάστατο πίνακα.

Αρχίζουμε ανοίγοντας το αρχείο. Αμέσως μετά διαβάζουμε τους αριθμούς γραμμών και στηλών:

```
   ifstream tin( "egr63e2.txt" );
   int      nR, nC;
   tin >> nR >> nC;
```

Ο πίνακάς μας θα είναι *nR*×*nC*.

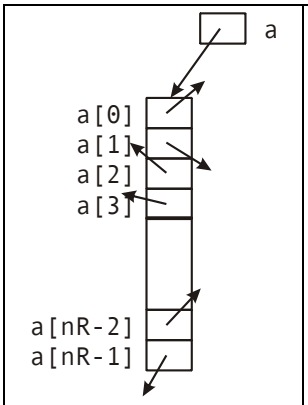
Αν δούμε τον δισδιάστατο δυναμικό πίνακα ως *πίνακα γραμμών –πίνακα πινάκων–* θα έχουμε *nR* μονοδιάστατους δυναμικούς πίνακες που καθένας τους θα έχει *nC* στοιχεία τύπου `double`. Για κάθε έναν από αυτούς τους πίνακες χρειαζόμαστε ένα βέλος τύπου `double*`· επομένως θα χρειαστούμε έναν δυναμικό πίνακα με *nR* στοιχεία τύπου `double*`:

```
   (double*)* a;
```

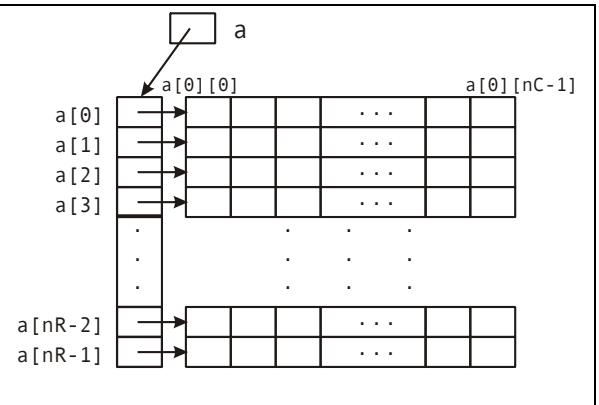
---

[3] Μην προσπαθήσεις να επιφορτώσεις τον "`=`" για τύπο δομής με αυτά που είπαμε στην §14.6.3. Αργότερα θα μάθουμε πώς γίνεται.

[4] Ε, δεν μπορεί να τρόμαξες από αυτά που είπαμε για τη *save()* και τη *load()*! Αφού, όπως είδες, όλα διορθώνονται.



Σχ. 16-4 Τι έχουμε μετά την εκτέλεση της εντολής
`a = new double*[nR]`



Σχ. 16-5 Τι έχουμε μετά την εκτέλεση της εντολής
`for ( int r(0); r < nR; ++r )`
`  a[r] = new double[nC];`

ή απλούστερα:

```
double** a;
```

Έχουμε δηλαδή ένα *διπλό βέλος* ή *βέλος προς βέλος*.

Μόλις μάθουμε το πλήθος γραμμών *nR* του πίνακα μπορούμε να πάρουμε μνήμη για αυτόν τον πίνακα βελών:

```
a = new double*[nR];
```

Στο Σχ. 16-4 βλέπεις μια εικόνα του πίνακα a μετά την εκτέλεση της παραπάνω εντολής. Τα στοιχεία (βέλη) του πίνακα, τα `a[`*r*`]`, *r*: 0..*nR*-1, δεν είναι ορισμένα· αυτό παριστάνεται με τα βέλη που δείχνουν σε τυχαίες διευθύνσεις, νόμιμες ή όχι.

Το τελευταίο βήμα είναι η υλοποίηση των γραμμών με δυναμική μνήμη. Αυτό γίνεται με την:

```
for ( int r(0); r < nR; ++r ) a[r] = new double[nC];
```

Τη μορφή του πίνακα μετά από αυτήν την εντολή τη βλέπεις στο Σχ. 16-5. Το στοιχείο *c* της γραμμής *r* είναι το "`(a[r])[c]`" ή απλούστερα "`a[r][c]`". Αυτόν τον συμβολισμό τον ξέρουμε ήδη από τους μη δυναμικούς δισδιάστατους πίνακες.

Αν θελήσουμε να διαβάσουμε τα στοιχεία του πίνακα από το αρχείο, αυτό θα γίνει κατά τα γνωστά:

```
for ( int r(0); r < nR; ++r )
{
   for ( int c(0); c < nC; ++c ) tin >> a[r][c];
}
tin.close();
```

Σε όλα αυτά που είπαμε δεν βάλαμε ελέγχους για να αναδείξουμε αυτά που μας ενδιαφέρουν εδώ. Στη συνέχεια θα τα ξαναδείς με τους ελέγχους τους (τους ελέγχους για την ανάγνωση από το αρχείο τους ξέρεις καλά.)

Και πώς ανακυκλώνουμε τη μνήμη όταν δεν τη χρειαζόμαστε; Πρώτα ανακυκλώνουμε τις γραμμές:

```
for ( int r(0); r < nR; ++r ) delete[] a[r];
```

και μετά τον πίνακα των βελών:

```
delete[] a;
```

Αφού όλα αυτά διαφέρουν μόνον ως προς τον τύπο των στοιχείων του πίνακα μπορούμε να τα βάλουμε σε δύο περιγράμματα. Εδώ θα βάλουμε και ελέγχους και καλό είναι να τους προσέξεις. Θα χρησιμοποιήσουμε μια δομή εξαιρέσεων που είναι κάτι σαν:

```
struct MyTmpltLibXptn
{
   enum {  domainError, noArray, allocFailed  };

   char funcName[100];
```

```
   int  errCode;
   int  errVal1, errVal2;
   MyTmpltLibXptn( const char* fn, int ec,
                   int erv1 = 0, int erv2 = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errVal1 = erv1;  errVal2 = erv2;  }
}; // MyTmpltLibXptn
```

Θα τη δούμε σε επόμενο κεφάλαιο.

Το πρώτο περίγραμμα –το ονομάζουμε *new2d*– τροφοδοτείται με τα πλήθη γραμμών και στηλών και επιστρέφει βέλος τύπου *T*`**` (*T* ο τύπος-παράμετρος):

```
 0: template< typename T >
 1: T** new2d( int nR, int nC )
 2: {
 3:    if ( nR <= 0 || nC <= 0 )
 4:       throw MyTmpltLibXptn( "new2d",
 5:                             MyTmpltLibXptn::domainError,
 6:                             nR, nC );
 7:    T** fv;
 8:    try {  fv = new T*[nR];  }
 9:    catch( bad_alloc& )
10:    {  throw MyTmpltLibXptn( "new2d",
11:                             MyTmpltLibXptn::allocFailed );  }
12:    for ( int r(0); r < nR; ++r )
13:    {
14:       try {  fv[r] = new T[nC];  }
15:       catch( bad_alloc& )
16:       {
17:          for ( int k(0); k < r; ++k )  delete[] fv[k];
18:          delete[] fv;
19:          throw MyTmpltLibXptn( "new2d",
20:                                MyTmpltLibXptn::allocFailed );
21:       } // catch
22:    } // for ( int r
23:    return fv;
24: } // new2d
```

Στη γρ. 8 προσπαθούμε να πάρουμε μνήμη για τα βέλη των γραμμών· αν δεν τα καταφέρουμε ρίχνουμε εξαίρεση και τελειώσαμε. Στη γρ. 14, που είναι μέσα στην περιοχή επανάληψης της **for** (γρ. 12) προσπαθούμε να πάρουμε μνήμη για τη γραμμή *r*. Μέχρι το σημείο αυτό έχουμε πάρει ήδη μνήμη για τα βέλη των γραμμών αλλά και για τις γραμμές 0 .. *r*-1. Αν δεν μπορέσουμε να πάρουμε μνήμη για τη γραμμή *r* πρέπει –πριν ρίξουμε την εξαίρεση– να ανακυκλώσουμε όλη αυτή τη μνήμη που ήδη πήραμε. Αυτό ακριβώς κάνουμε στις γρ. 17 και 18.

Η *delete2d()* είναι πολύ πιο απλή:

```
template< typename T >
void delete2d( T**& a, int nR )
{
   if ( a == 0 && nR > 0 )
      throw MyTmpltLibXptn( "delete2d", MyTmpltLibXptn::noArray );
   for ( int r(0); r < nR; ++r )  delete[] a[r];
   delete[] a;  a = 0;
} // delete2d
```

Πρόσεξε μόνον τον τύπο της πρώτης παραμέτρου: "`T**&`" διπλό βέλος αναφοράς! Εντυπωσιακό μεν αλλά απολύτως φυσικό: το a είναι ένα διπλό βέλος του οποίου η τιμή αλλάζει μέσα στη συνάρτηση.

Πρόσεξε ότι κάνουμε και αυτό που δεν κάνει η C++: μηδενίζουμε το βέλος μετά την ανακύκλωση του στόχου του.

Με αυτά τα εργαλεία, το παράδειγμα που ξεκινήσαμε γράφεται:

```
   tin >> nR >> nC;
   a = new2d<double>( nR, nC );
```

```
   for ( int r(0); r < nR; ++r )
   {
      for ( int c(0); c < nC; ++c ) tin >> a[r][c];
   }
   tin.close();
// . . .
   delete2d( a, nR );
```

Πρόσεξε ότι η *new2d()* δεν έχει κάποια παράμετρο με βάση την οποία θα μπορούσε να γίνει αυτόματη εξειδίκευση. Έτσι η κλήση θα πρέπει να γίνει υποχρεωτικώς ως: "`new2d<double>`".

Για να συγκρίνεις τους δυναμικούς δισδιάστατους πίνακες με τους συμβατικούς θα γράψουμε για τέταρτη (!) φορά το πρόγραμμα πολλαπλασιασμού πινάκων που πρωτοείδαμε στο Παράδ 4 της §12.4 (χωρίς συναρτήσεις), ξαναείδαμε στο Παράδ. 6 της §13.9.3 (συνάρτηση με παράμετρο τον «γραμμοποιημένο» πίνακα) και τέλος στην §14.7.1, όπου δείξαμε και ένα τέχνασμα με τις παραμέτρους περιγραμμάτων.

Τώρα θα το δούμε με δυναμικούς πίνακες:

```
   const int l( 3 ),  m( 5 ),  n( 2 );

   int** a( new2d<int>(l, m) );
   int** b( new2d<int>(m, n) );
   int** d( new2d<int>(l, n) );
```

Πρόσεξε πώς γράφονται τώρα οι συναρτήσεις εισόδου/εξόδου των πινάκων:

```
void input2DAr( istream& tin, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
} // input2Dar

void output2DAr( ostream& tout, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Ο πίνακας περνάει στη συνάρτηση με: (διπλό) βέλος, πλήθος γραμμών, πλήθος στηλών. Μέσα στις συναρτήσεις χρησιμοποιείται με τον συνήθη συμβολισμό των δισδιάστατων πινάκων (χωρίς «γραμμοποιήσεις»).

Οι συναρτήσεις καλούνται ως εξής: Διαβάζουμε με τις:

```
   input2DAr( atx, a, l, m );
   input2DAr( atx, b, m, n );
```

και γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr( cout, a, l, m );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr( cout, b, m, n );
   cout << " Στοιχεία του πίνακα d" << endl;
   output2DAr( cout, d, l, n );
```

Στο τέλος, ανακυκλώνουμε τη μνήμη που πήραμε με τις:

```
   delete2d( a, l );
   delete2d( b, m );
   delete2d( d, l );
```

Για όποιον δεν το πρόσεξε να το επισημάνουμε: *Οι γραμμές ενός δισδιάστατου δυναμικού πίνακα δεν είναι απαραίτητο να έχουν το ίδιο μήκος!* Ας δούμε ένα

Παράδειγμα

Θέλουμε έναν δυναμικό δισδιάστατο πίνακα που κάθε γραμμή του θα έχει το όνομα ενός μήνα. Δηλώνουμε:

```
   char** monthName( 0 );
```

και παίρνουμε μνήμη ως εξής:

```
   monthName = new char*[12];

   monthName[0] = new char[strlen("January")+1];
   strcpy( monthName[0], "January" );
   monthName[1] = new char[strlen("February")+1];
   strcpy( monthName[1], "February" );
// . . .
   monthName[10] = new char[strlen("November")+1];
   strcpy( monthName[10], "November" );
   monthName[11] = new char[strlen("December")+1];
   strcpy( monthName[11], "December" );
```

Όπως βλέπεις, για κάθε μήνα παίρνουμε ακριβώς τη μνήμη που χρειαζόμαστε (το "`+1`" για τον φρουρό).

Όταν θέλουμε να ανακυκλώσουμε τη μνήμη, η

```
   delete2d( monthName, 12 );
```

κάνει τη δουλειά μια χαρά.

Παρατήρηση:

Αυτά για το παράδειγμα. Σε πραγματικές καταστάσεις. πολύ πιο βολική λύση είναι η

```
   string* sMonthName( new string[12] );
// . . .
   delete[] sMonthName;
```

Αν πάρουμε υπόψη μας ότι σε μια τιμή *string* το κείμενο αποθηκεύεται σε δυναμική μνήμη, έχουμε και πάλι δυναμικό πίνακα δυναμικών πινάκων.

16.10 * Τύπος Βέλους: "`void*`"

Η C δεν υποστηρίζει περιγράμματα συναρτήσεων και για να διευκολύνει τους προγραμματιστές στην προσπάθειά τους να γράψουν συναρτήσεις ανεξάρτητες από τύπο (δίνουμε ένα παράδειγμα στη συνέχεια) δίνει και άλλο ένα εργαλείο: τον τύπο γενικού βέλους "`void*`".

Σε ένα βέλος αυτού του τύπου μπορείς να δώσεις ως τιμή τη διεύθυνση αντικειμένου οποιοδήποτε τύπου. Για παράδειγμα:

```
   int iv;
   void* pv1( &iv );
   double dv;
   pv1 = &dv;
   Date dt;
   pv1 = &dt;
```

Η αντίστροφη εκχώρηση δεν μπορεί να γίνει χωρίς τυποθεώρηση, π.χ.:

```
   int* pInt( reinterpret_cast<int*>(pv1) );
```

Στα βέλη `void*` δεν μπορούμε να κάνουμε αποπαραπομπή αν προηγουμένως δεν κάνουμε ερμηνευτική τυποθεώρηση (`reinterpret_cast`).

Αναφέραμε τα παραπάνω μόνο για προετοιμασία για την επόμενη παράγραφο και δεν θα δεις να τα χρησιμοποιούμε ξανά.

Ας δούμε ένα παράδειγμα γενικής συνάρτησης της C ώστε να δεις και χρήση του "`void*`" αλλιώτικο από αυτό που θα δεις στην επόμενη παράγραφο. Βάζοντας στο πρόγραμμά σου "`#include <cstdlib>`" μπορείς να χρησιμοποιήσεις τη συνάρτηση της C

```
void qsort( void* base, size_t num, size_t size,
            int (*compar)(const void*, const void*) );
```

που ταξινομεί –με τον αλγόριθμο Quicksort– *num* στοιχεία ενός πίνακα ξεκινώντας από αυτό που δείχνει το βέλος *base*. Η παράμετρος *size* περνάει το μέγεθος (σε ψηφιολέξεις) ενός στοιχείου του πίνακα. Η τελευταία παράμετρος είναι (βέλος προς) μια συνάρτηση. Όταν καλείται από την *qsort()* επιστρέφει:

- *Αρνητική* τιμή αν το στοιχείο που δείχνει η *πρώτη* παράμετρος *προηγείται* του στοιχείου που δείχνει η δεύτερη παράμετρος.
- *Μηδέν* αν το στοιχείο που δείχνει η *πρώτη* παράμετρος είναι *ίσο* με το στοιχείο που δείχνει η δεύτερη παράμετρος.
- *Θετική* τιμή αν το στοιχείο που δείχνει η *δεύτερη* παράμετρος *προηγείται* του στοιχείου που δείχνει η πρώτη παράμετρος.

Εδώ βλέπεις έναν άλλον τρόπο να γράφεις γενικές συναρτήσεις: με διευθύνσεις (βέλη) και μεγέθη περιοχών στη μνήμη. Έχουμε δηλαδή προγραμματισμό χαμηλού επιπέδου, που, όπως έχουμε πει, είναι δύσκολος. Πάντως υπάρχει και ένα (μικρό) κέρδος: ενώ, όπως είπαμε, ο μεταγλωττιστής θα δημιουργήσει μια συνάρτηση για κάθε στιγμιότυπο του περιγράμματος που θα βρει στο πρόγραμμά μας, η *qsort()* –και οποιαδήποτε συνάρτηση γραμμενη με αυτόν τον τρόπο– θα μεταγλωττισθεί μια φορά μόνο. Για κάθε κλήση της θα πρέπει να γράψουμε μια (το πολύ) συνάρτηση *compar()*.

Ας πούμε λοιπόν ότι σε κάποιο πρόγραμμα έχουμε:

```
int    intArr[ 50 ];
double dblArr[ 100 ];
```

και θέλουμε να ταξινομήσουμε ολόκληρον τον dblArr, κατ' *αύξουσα* τάξη, με χρήση της *qsort()*. Θα την καλέσουμε ως εξής:

```
qsort( dblArr, 100, sizeof(double), compareDblInc );
```

Το πρώτο όρισμα θα μπορούσε, αντί για "`dblArr`", να είναι "`&dblArr[0]`". Σε κάθε περίπτωση αυτό αντιγράφεται στο *base*. Πώς θα είναι η *compareDblInc()*; Έτσι:[5]

```
int compareDblInc( const void* a, const void* b )
{
   return ( *reinterpret_cast<const double*>(a) -
            *reinterpret_cast<const double*>(b) );
} // compareDblInc
```

Αφού θέλουμε ταξινόμηση κατ' αύξουσα τάξη το "`7.1`" προηγείται του "`11.7`". Έτσι η κλήση "`compareDblInc(&dblArr[35],&dblArr[71])`", όπου τα δύο στοιχεία έχουν αντιστοίχως τις παραπάνω τιμές, θα επιστρέψει τιμή "`-4`" (`== int(-4.6)`). Η *qsort()* θα κάνει πολλές τέτοιες κλήσεις.

Αν θέλουμε να ταξινομήσουμε το κομμάτι `intArr[11]` μέχρι `intArr[20]` κατά *φθίνουσα* τάξη θα πρέπει να γράψουμε:

```
qsort( &intArr[11], 10, sizeof(int), compareIntDec );
```

όπου:

```
int compareIntDec( const void* a, const void* b )
{
   return ( *reinterpret_cast<const int*>(b) -
            *reinterpret_cast<const int*>(a) );
} // compareIntDec
```

Πρόσεξε ότι με αυτήν το "`11`" προηγείται του "`7`".

---

[5] Στη C θα γράφανε: "`return ( *(double*)a - *(double*)b );`".

16.11 * Αναμνήσεις από τη C: *malloc(), free(), realloc()*

Η C++, έχοντας τη C ως υποσύνολό της, δίνει τη δυνατοτητα διαχείρισης της δυναμικής μνήμης και με τις συναρτήσεις *malloc()* (*calloc()*), *free()* και *realloc()*. Επειδή είναι πολύ πιθανό να τις συναντήσεις σε προγράμματα αξίζει να τις δούμε, έστω και εν συντομία. Για να τις χρησιμοποιήσεις θα πρέπει να περιλάβεις στο πρόγραμμά σου το `cstdlib`.

Η επικεφαλίδα της *malloc()* είναι:

```
void* malloc( size_t size );
```

Το "`void*`" λέει ότι η συνάρτηση επιστρέφει βέλος χωρίς τύπο. Μέσω της παραμέτρου πρέπει να περάσουμε την ποσότητα της ζητούμενης μνήμης σε ψηφιολέξεις. Αν έχεις δηλώσει:

```
   int* pInt;
```

και ζητήσεις:

```
   pInt = reinterpret_cast<int*>( malloc(72*sizeof(int) );
```

θα σου παραχωρηθεί που χωράει 72 τιμές τύπου `int` (πίνακας με 72 στοιχεία τύπου `int`) Το βέλος *pInt* δείχνει την αρχή της μνήμης που παραχωρήθηκε. Αν δεν υπάρχει διαθέσιμη η μνήμη που ζητείται η τιμή που επιστρέφεται είναι 0 (μηδέν).

Δεν είναι παράνομο να καλέσεις τη *malloc()* με όρισμα 0 αλλά το αποτέλεσμα εξαρτάται από την εγκατάσταση:

- Μπορεί να σου επιστρέψει βέλος 0.
- Μπορεί να σου επιστρέψει μη μηδενικό βέλος στο οποίο όμως δεν μπορείς να κάνεις αποπαραπομπή.

Παρόμοια με τη *malloc()* είναι η:

```
void* calloc( size_t nItems, size_t size );
```

Αυτή ταιριάζει πιο πολύ σε πίνακες: σου επιτρέπει να ζητήσεις *nItems* θέσεις μνήμης που κάθε μια τους θα πιάνει *size* ψηφιολέξεις. Το παραπάνω παράδειγμα θα μπορούσε να γραφεί:

```
   pInt = reinterpret_cast<int*>( calloc(72, sizeof(int) );
```

Όπως καταλαβαίνεις, αυτές οι δύο συναρτήσεις παίζουν τον ρόλο του τελεστή `new`.

Τον ρόλο του `delete` τον παίζει η συνάρτηση

```
void free( void* block );
```

που επιστρέφει στον σωρό τη μνήμη που δείχνει το βέλος *block*.

Πολύ συχνά, όταν δουλεύουμε με δυναμικούς πίνακες, έχουμε ανάγκη να μεγαλώσουμε (ή και να μικρύνουμε) κάποιον τον πίνακα. Για την περίπτωση αυτήν η C++ (C) μας δίνει τη συνάρτηση

```
void* realloc( void* block, size_t size );
```

που επεκτείνει ή συρρικνώνει το κομμάτι δυναμικής μνήμης που δείχνει το βέλος *block* σε *size* ψηφιολέξεις. Συνήθως αυτό γίνεται με παραχώρηση νέου τμήματος μνήμης στο οποίο η *realloc()* φροντίζει να αντιγράψει το περιεχόμενο του αρχικού τμήματος.

Το παρακάτω παράδειγμα δείχνει τη χρήση των συναρτησεων.

```
 0: #include <iostream>
 1: #include <cstdlib>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    int*    ip( 0 );
 8:    double* dp( 0 );
 9:
10:    ip = reinterpret_cast<int*>( malloc(3*sizeof(int)) );
11:    dp = reinterpret_cast<double*>( calloc(3,sizeof(double)));
12:    for ( int k(0); k <= 2; ++k )
```

```
13:    {
14:       ip[k] = k+1;
15:       dp[k] = 1.1*ip[k];
16:    }
17:    for ( int k(0); k <= 2; ++k )
18:       cout << ip[k] << "  " << dp[k] << endl;
19:    cout << "==========" << endl;
20:    ip = reinterpret_cast<int*>( realloc(ip, 7*sizeof(int)) );
21:    dp = reinterpret_cast<double*>(
22:                               realloc(dp, 7*sizeof(double)) );
23:    for ( int k(3); k <= 6; ++k )
24:    {
25:       ip[k] = k+1;
26:       dp[k] = 1.1*ip[k];
27:    }
28:    for ( int k(0); k <= 6; ++k )
29:       cout << ip[k] << "  " << dp[k] << endl;
30:    free( dp );
31:    free( ip );
32: } // main
```

Αποτέλεσμα:

```
1  1.1
2  2.2
3  3.3
==========
1  1.1
2  2.2
3  3.3
4  4.4
5  5.5
6  6.6
7  7.7
```

Αρχικώς, με τις *malloc()* και *calloc()*, πήραμε μνήμη για πίνακες με τρία στοιχεία (γρ. 10-11). Στη συνέχεια, με τη *realloc* (γρ. 20-22) επεκτείναμε τους πίνακες σε επτά στοιχεία για τον καθένα. Στο τέλος, με τη *free()* (γρ. 30-31), επιστρέφουμε τη μνήμη που πήραμε.

Πριν τελειώσουμε να πούμε ότι είναι *σαφώς προτιμότερη η χρήση των* `new` *και* `delete` και να επαναλάβουμε ότι, σε κάθε περίπτωση:

- ***Μνήμη που παίρνεις με*** `new` ***θα απελευθερώνεται με*** `delete` ***και μνήμη που παίρνεις με malloc(), calloc(), realloc() θα απελευθερώνεται με free().***

και ακόμη περίσσότερο:

- ***Μη χρησιμοποιείς στο ίδιο πρόγραμμα τις συναρτήσεις δυναμικής μνήμης της C και τους*** `new` ***και*** `delete`***.***

Αυτό βέβαια δεν εύκολο όταν μέσα στο πρόγραμμά σου χρησιμοποιείς βιβλιοθήκες ή έτοιμα κομμάτια προγράμματος. Στην περίπτωση αυτή περιορίσου στην τήρηση του πρώτου κανόνα.

Τέλος, να πούμε κάτι που μπορεί να συμβαίνει και να κάνει τα παραπάνω να φαίνονται περίεργα: Σε ορισμένες υλοποιήσεις της C++ οι `new` και `delete` υλοποιούνται με τις *malloc()* και *free()*.

- Αν αναπτύξεις μια εφαρμογή σε ένα τέτοιο περιβάλλον μπορεί να μην έχεις πρόβλημα όσο και αν ανακατέψεις τους τελεστές της C++ με τις συναρτήσεις της C.
- Σε μια τέτοια εγκατάσταση μπορεί το πρόβλημα της κλήσης "`malloc(0)`" να μεταφερθεί και στον τελεστή `new`.

16.12 Για να Μη Ζηλεύουμε τη *realloc()*

Όπως είπαμε και παραπάνω, θα πρέπει να προτιμάς τους **new** και **delete** που είναι πιο βολικοί από τις συναρτήσεις. Βέβαια υπάρχει και η *realloc*, που χρειάζεται αρκετά συχνά. Αργότερα θα μάθουμε ότι η C++ μας δίνει πολύ καλά εργαλεία[7], όπως το *vector*, το *map*, το *list*, που σου επιτρέπουν να έχεις δυναμικές δομές δεδομένων χωρίς να ανησυχείς για "**new**" και "**delete**".

Προς το παρόν όμως το πρόβλημα μπορεί να σου το λύσει το παρακάτω περίγραμμα συνάρτησης, που

- παίρνει το βέλος (*p*) προς έναν δυναμικό πίνακα με στοιχεία τύπου *T* και
- το αλλάζει σε βέλος προς νέον δυναμικό πίνακα με *nf* στοιχεία του ίδιου τύπου. Στα πρώτα *ni* στοιχεία του νέου πίνακα αντιγράφονται τα πρώτα *ni* στοιχεία του παλιού.

Η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*domainError*)[8] αν κληθεί χωρίς να ισχύει η συνθήκη $0 \le ni \le nf$.

Η συνάρτηση θα δεχθεί βέλος 0 (μηδέν) αρκεί να μην έχει θετική τιμή η *ni*. Αλλιώς (*p* == 0 && *ni* > 0) θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*noArray*).

Φυσικά, η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*allocFailed*) αν δεν μπορέσει να πάρει την απαραίτητη μνήμη.

```
template< typename T >
void renew( T*& p, int ni, int nf )
{
   if ( ni < 0 || nf < ni )
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::domainError, ni, nf );
// 0 <= ni <= nf
   if ( p == 0 && ni > 0 )
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::noArray );
// (0 <= ni <= nf) && (p != 0 || ni == 0)
   try
   {
      T* temp( new T[nf] );
      for ( int k(0);  k < ni; ++k )  temp[k] = p[k];
      delete[] p;   p = temp;
   }
   catch( bad_alloc& )
   {
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed );
   }
} // void renew
```

Παρατηρήσεις: ▶

1. Οτιδήποτε και να πάει στραβά, αν ριχτεί εξαίρεση, δεν θα πειραχτεί ο αρχικός πίνακας.
2. Η *bad_alloc* μπορεί να προέλθει όχι μόνο από τη "**new T[nf]**" αλλά και από τις αντιγραφές "**temp[k] = p[k]**". Αυτό μπορεί να γίνει αν ο *T* έχει αντικείμενα που χρησιμοποιούν δυναμική μνήμη, όπως λέγαμε στην §16.8. Θα πρέπει να κάνουμε κάτι σαν αυτό που κάναμε στη *new2d*. Θα το μάθουμε αργότερα. ◀

Στη συνέχεια, ξαναδίνουμε το παράδειγμα που είδαμε πιο πάνω αλλά με τα εργαλεία της C++:

```
#include <iostream>
#include <new>

using namespace std;

template< typename T >
void renew( T*& p, int ni, int nf );
```

[7] Πρόκειται για περιγράμματα κλάσεων.

[8] Είδαμε τη *MyTmpltLibXptn* πιο πριν, στην §16.9.

```
int main()
{
   int*    ip( 0 );
   double* dp( 0 );

   ip = new int [3];  dp = new double [3];
   for ( int k = 0; k <= 2; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 2; ++k )
      cout << ip[k] << "  " << dp[k] << endl;
   cout << "==========" << endl;
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
   for ( int k = 3; k <= 6; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 6; ++k )
      cout << ip[k] << "  " << dp[k] << endl;
   delete [] dp;
   delete [] ip;
} // main
```

Η `renew(ip, 3, 7)` αλλάζει την τιμή του βέλους προς έναν δυναμικό πίνακα με 7 στοιχεία τύπου `int`. Στα 3 πρώτα στοιχεία αυτού του πίνακα αντιγράφονται τα στοιχεία του πίνακα που έδειχνε το *ip* πριν από την κλήση της συνάρτησης. Παρόμοια κάνει και η `renew(dp, 3, 7)` στον dp.

Η C++ ελαχιστοποιεί την ανάγκη για χρήση της *realloc* με το περίγραμμα κλάσης *vector* που υπάρχει στην πάγια βιβλιοθήκη περιγραμμάτων (Standard Template Library, STL).

16.13 Παραδείγματα

Θα δώσουμε τώρα τρία παραδείγματα χρήσης δυναμικής μνήμης που είναι τρεις παραλλαγές του δεύτερου προγράμματος (§15.14.2) της §15.14. Πριν προχωρήσουμε όμως θα πρέπει να τονίσουμε ότι *τα παραδείγματα δίνονται επειδή κρίνονται κατάλληλα για την επίδειξη χρήσης όσων είπαμε παραπάνω. Σε καμιά περίπτωση δεν εννοούμε ότι αυτές οι λύσεις είναι καλύτερες από την αρχική.*

Παράδειγμα 1

Αντί να διαβάζουμε τα δεδομένα του κάθε χημικού στοιχείου που μας ζητείται και να τα φυλάγουμε μετά τη συμπλήρωση/διόρθωση του στοιχείου αυτού

- *διαβάζουμε ολόκληρο το αρχείο σε έναν δυναμικό πίνακα,*
- *κάνουμε όλες τις ενημερώσεις στον πίνακα και*
- *φυλάγουμε στο αρχείο τον ενημερωμένο πίνακα.*

Αυτό το πρόγραμμα είναι απλούστερο από το αρχικό. Δηλώνουμε:

```
   GrElmn* grElmnTbl( 0 );
```

και μόλις μάθουμε τον μέγιστο ατομικό αριθμό –που είναι ίσος με το πλήθος των στοιχείων– ζητούμε την απαραίτητη μνήμη:

```
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
```

Για την περίπτωση αποτυχίας προσθέτουμε, μετά την ομάδα **try**, άλλη μια:

```
   catch ( bad_alloc& )
   {
      cout << "not enough memeory to load data" << endl;
   }
```

Στη συνέχεια φορτώνουμε το περιεχόμενο του αρχείου στον πίνακα με την:

```
      loadAllData( bInOut, grElmnTbl, maxAtNo );
```

όπου:

```
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekg( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_load( grElmnTbl[k], bInOut );
} // loadAllData
```

Η *editGrName* δεν μας βολεύει πια. Την τροποποιούμε:

```
void editGrNameMM( GrElmn& a )
{
   GrElmn_display( a, cout );
   string newGrName;
   cout << "new greek name: "; getline( cin, newGrName, '\n' );
   if ( !newGrName.empty() )
   {
      GrElmn_setGrName( a, newGrName );
   }
} // editGrNameMM
```

Όπως βλέπεις, δεν έχει πια στις παραμέτρους το ρεύμα απο/προς το αρχείο αφού η δουλειά γίνεται στην κύρια μνήμη (βάλαμε το "*MM*" για να μας θυμίζει τη «Main Memory»).

Την καλούμε, αφού πρώτα διαβάσουμε το ατομικό αριθμο, ως εξής:

```
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
```

(θυμίσου ότι το στοιχείο με ατομικό αριθμό *aa* βρίσκεται στη θέση *aa*-1 του πίνακα.)

Τελικώς, φυλάγουμε τον πίνακα με την:

```
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekp( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_save( grElmnTbl[k], bInOut );
} // saveAllData
```

Οι συναρτήσεις *readRandom()* και *writeRandom()* δεν μας χρειάζονται. Οι *loadAllData()* και *saveAllData()* επεξεργάζονται το αρχείο ως σειριακό.

Το πρόγραμμά μας θα είναι κάπως έτσι:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
struct GrElmn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
// ΝΕΕΣ ΣΥΝΑΡΤΗΣΕΙΣ
void editGrNameMM( GrElmn& a );
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );

int main()
```

```
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );

   try
   {
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
      loadAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
      } while ( aa != 0 );
      openFile( flNm, bInOut );
      saveAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch ( bad_alloc& )
   {
      cout << "not enough memory to load data" << endl;
   }
   catch( ApplicXptn& x )
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Όπως βλέπεις:

- μια φορά παίρνουμε μνήμη (`grElmnTbl = new GrElmn[maxAtNo]`),
- μια φορά ανακυκλώνουμε (`delete[] grElmnTbl`).

Έτσι, δεν υπάρχει λόγος να πιάσουμε τη *bad_alloc* και να ρίξουμε δική μας.

Παράδειγμα 2

Θα ξαναλύσουμε το προηγούμενο πρόβλημα με έναν άλλον τρόπο: *Θα κρατούμε σε πίνακα μόνο τις εγγραφές που ζητούνται από τον χρήστη και με το τέλος χρήσης του προγράμματος θα ενημερώνουμε το αρχείο.*

Η ουσιαστική διαφορά από την προηγούμενη περίπτωση είναι η εξής:

- Στο προηγούμενο πρόγραμμα με το άνοιγμα του αρχείου μπορούσαμε να υπολογίσουμε το μέγεθος του δυναμικού πίνακα.
- Τώρα το μέγεθος του δυναμικού πίνακα εξαρτάται από το πόσο θα δουλέψει ο χρήστης! Κάθε φορά που ο χρήστης αποφασίζει να διορθώσει ένα στοιχείο το μέγεθος του δυναμικού πίνακα θα αυξάνεται κατά 1.

Αυτό το τελευταίο μπορεί και να μην είναι έτσι ακριβώς. Ας πούμε, ότι ο χρήστης παίρνει το στοιχείο 74 (Tungsten), το μεταφράζει «Τουνγκστένιο» και μετά θυμάται ότι στα ελληνικά το λέμε «Βολφράμιο»! Τα δεδομένα για το στοιχείο αυτό βρίσκονται στον πίνακα

και δεν χρειάζεται να τα ξαναφορτώσει! Άρα μπορεί να κάνει και διορθώσεις χωρίς να βάζει και άλλα στοιχεία στον πίνακα.

Αυτό όμως βάζει μια άλλη απαίτηση: αναζήτηση στα στοιχεία του πίνακα με βάση τον ατομικό αριθμό. Αυτό μπορούμε να το αντιμετωπίσουμε: Θα πάρουμε τη *linSearch()* –όπως την κάναμε στην §12.2.1– και θα τη μετατρέψουμε σε περίγραμμα (§16.13.1). Αλλά, για τη χρησιμοποιήσουμε θα πρέπει να μπορούμε να κάνουμε εκχώρηση (τελεστής "=") και σύγκριση "!=". Θα πρέπει λοιπόν να επιφορτώσουμε τον τελεστή "!=" (και τον αντίθετό του "==") για τον *GrElmn*. Αυτό είναι πολύ απλό αν σκεφτούμε ότι αντικείμενα τύπου *GrElmn* είναι ίσα αν και μόνον αν έχουν ίδιο ατομικό αριθμό:

```
bool operator!=( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber != rhs.geANumber );  }

bool operator==( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber == rhs.geANumber );  }
```

Να σκεφτούμε τώρα πώς θα διαχειριστούμε τον δυναμικό πίνακα. Θα μπορούσαμε να καλούμε τη *renew()* κάθε φορά που ζητείται κάποιο στοιχείο που δεν υπάρχει στον πίνακα. Για σκέψου όμως τι σημαίνει αυτό:

- Όταν βάλουμε το δεύτερο στοιχείο θα κάνουμε μια αντιγραφή (του πρώτου).
- Όταν βάλουμε το τρίτο θα κάνουμε δύο αντιγραφές (του πρώτου και του δεύτερου).

κ.ο.κ. Έτσι, αν ο χρήστης, σε μια εκτέλεση του προγράμματος διορθώσει 12 διαφορετικά στοιχεία η *renew()* θα κάνει:

$$1 + 2 + 3 + \ldots + 11 = 66 \text{ αντιγραφές}$$

Η άλλη ακραία επιλογή είναι να κρατήσουμε χώρο για ολόκληρον τον πίνακα αλλά να χρησιμοποιήσουμε μόνο 12 από τις θέσεις του.

Ας εξετάσουμε μια ενδιάμεση περίπτωση: Να αυξάνουμε το μέγεθος τού πίνακα κατά πεντάδες. Έτσι, όταν πάρουμε τη δεύτερη πεντάδα θα έχουμε 5 αντιγραφές και όταν πάρουμε την τρίτη άλλες 10. Συνολικώς θα κάνουμε 15 αντιγραφές ενώ θα πάρουμε μόνον τρεις παραπάνω θέσεις.

Η λύση στο πρόβλημά μας είναι κάτι τέτοιο. Αλλά, θα αυξάνουμε κατά 5 ή κατά 10 ή κατά 80; Η απάντηση εξαρτάται από το συγκεκριμένο πρόβλημα κάθε φορά. Όπως είδες, με την υπόθεση εργασίας «*σε μια εκτέλεση του προγράμματος διορθώνει 12 διαφορετικά στοιχεία*», το «κβάντο» αύξησης 5 φαίνεται να δουλεύει μια χαρά. Αν λέγαμε ότι θα διορθωσει 35 με 40 στοιχεία ένα «κβάντο» 10 ή 15 θα ήταν πιο κατάλληλο.

Και πριν προχωρήσουμε παρακάτω να υπενθμίσουμε ότι η *linSearch* χρησιμοποιεί και φρουρό. Για να έχουμε, λοιπόν, *n* στοιχεία αποθηκευμένα και να κάνουμε και αναζητήσεις χρειαζόμαστε πίνακα με *n*+1 θέσεις.

Για να ανταποκριθούμε σε όλες τις απαιτήσεις που βάλαμε πιο πάνω θα πρέπει να δηλώσουμε για τον δυναμικό πίνακα τα εξής:

```
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων
```

και όταν πάρουμε για πρώτη φορά μνήμη:

```
      grElmnTbl = new GrElmn[incr];
      nReserved = incr;
'     nElmn = 0;
```

Κάθε φορά που θα θέλουμε να εισαγάγουμε ένα νέο στοιχείο –με ατομικό αριθμό *aa*– στον πίνακα θα δουλεύουμε ως εξής:

```
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
```

```
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
```

Πρόσεξε τα εξής σημεία:

- Αυξάνουμε το μέγεθος του πίνακα όταν *nElmn*+1 == *nReserved*. Το «**+1**» μας επιτρέπει να έχουμε μια διαθέσιμη θέση στο τέλος για να βάζει η *linSearch* τον φρουρό.
- Καλούμε τη *renew()* με την **renew(grElmnTbl, nElmn, nReserved+incr)**. Το νέο μέγεθος του πίνακα θα είναι **nReserved+incr**. Από τον αρχικό πίνακα θα αντιγραφούν τα **nElmn** στοιχεία.
- Το νέο στοιχείο διαβάζεται από το αρχείο στη θέση **nElmn** του πίνακα. Μετά την αύξηση της τιμής της **nElmn** η θέση θα είναι **nElmn-1**.

Όλα αυτά τα κρύβουμε μέσα σε μια συνάρτηση που σιγουρεύει ότι τα δεδομένα του στοιχείου που μας ενδιαφέρει βρίσκονται στον πίνακα:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos )
{
   GrElmn oneElmn( aa );
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = nElmn - 1;
   }
   else
      pos = lPos;
} // elmntInTable
```

Να δούμε τι γίνεται με αυτήν τη συνάρτηση:

- Κατ' αρχάς έχει πολλές (7) παραμέτρους. Οι τέσσερις από αυτές έχουν να κάνουν με τον δυναμικό πίνακα και οι τρεις από αυτές είναι αναφοράς. Πράγματι:
  - Είναι πολύ πιθανό να κάνει εισαγωγή νέου στοιχείου στον πίνακα οπότε μπορεί να αλλάξει το πλήθος στοιχείων *nElmn*.
  - Μπορεί να χρειαστεί να μεγαλώσει ο πίνακας –με κλήση της *renew*– οπότε αλλάζει το βέλος *grElmnTbl* και το πλήθος των δεσμεμένων στοιχείων *nReserved*.
- Πριν από οτιδήποτε άλλο, καλείται η *linSearch()* για να μας φέρει στην *lPos* τη θέση του στοιχείου, αν υπάρχει στον πίνακα. Η *linSearch()* περιμένει ότι η τιμή που θα αναζητήσει θα είναι του ίδιου τύπου με τα στοιχεία του πίνακα. Για αυτόν ακριβώς τον λόγο δηλώνουμε την *oneElmn*.
- Αφού η σύγκριση "**!=**" γίνεται με χρήση μόνο του ατομικού αριθμού, για να κάνουμε τη δουλειά μας αρκεί να βάλουμε *oneElmn.geANumber* τη τιμή της παραμέτρου *aa*, Αυτό μπορούμε να το κάνουμε:
  - Με την εντολή "**oneElmn.geANumber = aa**".
  - Με τη δήλωση της *oneElmn* αν έχουμε τον κατάλληλο δημιουργό. Προτιμήσαμε αυτή τη λύση και γράψαμε τον

```
struct GrElmn
{
// . . .
   GrElmn( int aAN=0 ) {  geANumber = aAN;  }
}; // GrElmn
```

- Αν έχουμε αυτόν τον δημιουργό δεν χρειάζεται να δηλώσουμε τη μεταβλητή *oneElmn* αλλά μπορούμε να καλέσουμε τη *linSearch()* ως εξής:

```
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, GrElmn(aa)));
```

- Είδαμε παραπάνω τι κάνουμε αν το στοιχείο που ζητάει ο χρήστης δεν υπάρχει στον πίνακα (*lPos* < 0). Στην περίπτωση αυτή τα δεδομένα φορτώνονται από το αρχείο στην τελευταία χρησιμοποιούμενη θέση του πίνακα και επιστρέφουμε ως τιμή της *pos* το `nElmn - 1`.
- Αν τα δεδομένα υπάρχουν στη θέση *lPos* αυτή επιστρέφεται ως τιμή της *pos*.

Όταν ο χρήστης τελειώσει τη δουλειά του το περιεχόμενο του πίνακα φυλάγεται στο αρχείο με την:

```
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
```

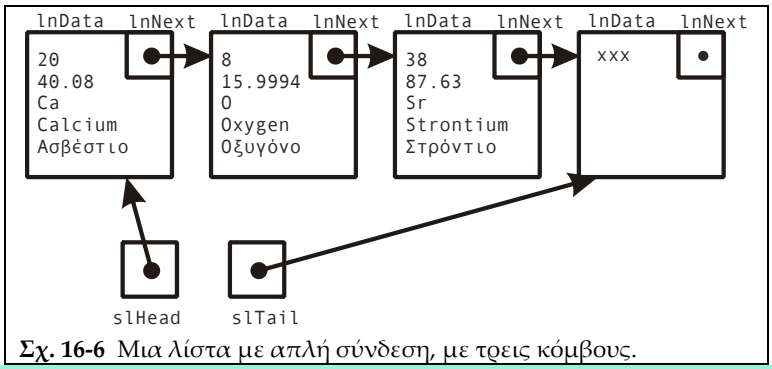
όπου:

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn )
{
   for ( int k(0); k < nElmn; ++k )
      writeRandom( grElmnTbl[k], bInOut );
} // saveUpdateTable
```

Τώρα το πρόγραμμα θα είναι ως εξής:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

template< typename T >
void renew( T*& p, int ni, int nf );
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x );

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
   GrElmn( int aAN=0 ) {  geANumber = aAN;  }
}; // GrElmn

// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
bool operator!=( const GrElmn& lhs, const GrElmn& rhs );
bool operator==( const GrElmn& lhs, const GrElmn& rhs );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
```

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn );
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos );

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων

   try
   {
      try {  grElmnTbl = new GrElmn[incr];  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "main", ApplicXptn::allocFailed );  }
      nReserved = incr;
      nElmn = 0;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            unsigned int pos;
            elmntInTable( grElmnTbl, nElmn, nReserved, incr,
                          bInOut, aa, pos );
            editGrNameMM( grElmnTbl[pos] );
         }
      } while ( aa != 0 );
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
      // ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
         case MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
```



Σχ. 16-6 Μια λίστα με απλή σύνδεση, με τρεις κόμβους.

```
            break;
         case MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στο πρόγραμμα αυτό παίρνουμε δυναμική μνήμη στη **main** (μια φορά) και στη *renew()*. Παρ' όλο που δεν υπάρχει περίπτωση σύγχυσης, αφού από τη *renew()* ρίχνονται εξαιρέσεις τύπου *MyTmpltLibXptn*, προτιμήσαμε να πιάνουμε και στη **main** τη *bad_alloc* και να ρίχνουμε μια δική μας *ApplicXptn*(::*allocFailed*).

Στη διαχείριση των εξαιρέσεων *ApplicXptn* προσθέσαμε μια επι πλέον περίπτωση *ApplicXptn*::*allocFailed*.

Παράδειγμα 3

Τώρα θα ξαναγράψουμε το πρόγραμμα του Παραδ. 2 με την εξής διαφορά: *Θα κρατούμε τις εγγραφές που ζητάει ο χρήστης όχι σε δυναμικό πίνακα αλλά σε μια λίστα με (απλή) σύνδεση.*

Η **λίστα με απλή σύνδεση** (simply linked list) –μια υλοποίηση της *ακολουθίας* (sequence)– είναι μια ευρύτατα χρησιμοποιούμενη δομή δεδομένων. Θα δούμε εδώ ένα τμήμα υλοποίησης μιας τέτοιας λίστας με λογική στοίβας.

Η παράσταση που θα υλοποιήσουμε φαίνεται στο Σχ. 16-6. Όπως βλέπεις, η λίστα είναι μια ακολουθία από *κόμβους* (nodes). Ο «κόμβος-φρουρός» στο τέλος απλουστεύει ορισμένους αλγόριθμους διαχείρισης της λίστας.

Κάθε κόμβος έχει δεδομένα (στην περίπτωσή μας τύπου *GrElmn*) και ένα βέλος-σύνδεσμο προς τον επόμενο κόμβο. Μπορούμε να τον υλοποιήσουμε στο πρόγραμμά μας με αντικείμενα τύπου:

```
struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode
```

Ένα βέλος δείχνει την αρχή της λίστας. Η ύπαρξη και δεύτερου βέλους που δείχνει τον φρουρό στο τέλος της λίστας απλουστεύει –όπως και ο ίδιος ο φρουρός, ορισμένους αλγόριθμους. Μπορούμε να πούμε λοιπόν ότι θα έχουμε:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

Στο Σχ. 16-7 βλέπεις μια κενή λίστα. Από αυτήν την εικόνα μπορούμε εύκολα να βγάλουμε τον ερήμην δημιουργό:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // Slist
```

Εδώ βλέπουμε ξανά να ρίχνεται εξαίρεση από δημιουργό. Βλέπουμε όμως και κάτι ακόμη: πώς παιρνουμε μνήμη για έναν κόμβο της λίστας. Με τον ίδιο τρόπο θα παίρνουμε και για τους υπόλοιπους κόμβους. Αλλά τι θα γίνει όταν θελήσουμε να ανακυκλώσουμε αυτή τη μνήμη; Και *η ανακύκλωση θα γίνει κομβο προς κόμβο*· δεν υπάρχει "`delete`" που να καθαρίζει όλη τη λίστα.

Να δούμε τώρα τι θέλουμε να κάνουμε με μια τέτοια λίστα:

- Εισαγωγή δεδομένων ενός στοιχείου στη λίστα (σε νέο κόμβο).
- Αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου.
- Φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.
- Ανακύκλωση όλων των κόμβων της λίστας.

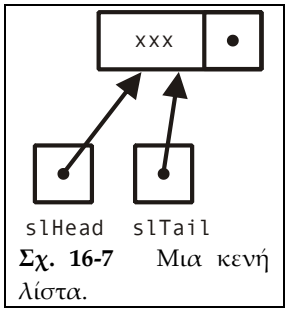
Ξεκινούμε με τη συνάρτηση εισαγωγής δεδομένων στη λίστα που δεν είναι και τόσο μπερδεμένη:[9]

```
void SList_push_front( SList& lst, const GrElmn& aData )
{
   ListNode* pnln( 0 );
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "SList_push_front",
                        ApplicXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = lst.slHead;
   lst.slHead = pnln;
} // SList_push_front
```

Ας δούμε τη λειτουργία της με ένα παράδειγμα από το «κτίσιμο» της λίστας του Σχ. 16-6: Έχουμε μια λίστα στην οποίαν υπάρχει μόνο το Στρόντιο και θέλουμε να εισαγάγουμε και το Οξυγόνο. Τα δεδομένα του Οξυγόνου υπάρχουν στην παράμετρο *aData*.

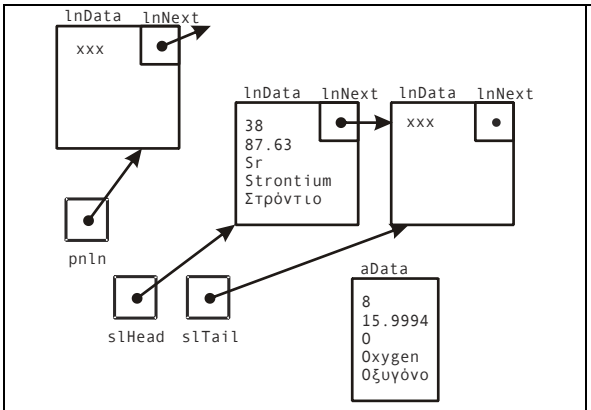
Αν προσπαθήσουμε να πάρουμε δυναμική μνήμη κάπως απρόσεκτα με μια "`lst.slHead = new ListNode`" *χάνουμε την αρχή της λίστας*. Επιλέγουμε λοιπόν να κάνουμε το εξής:

- Παίρνουμε μνήμη χρησιμοποιώντας ένα βοηθητικό βέλος, το *pnln* με την "`pnln = new ListNode`". Στο Σχ. 16-8α βλέπεις την

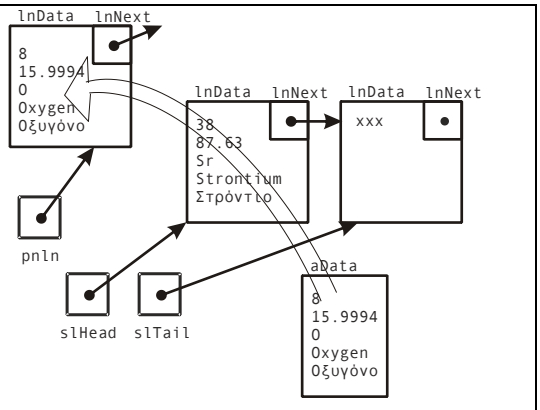


Σχ. 16-7 Μια κενή λίστα.

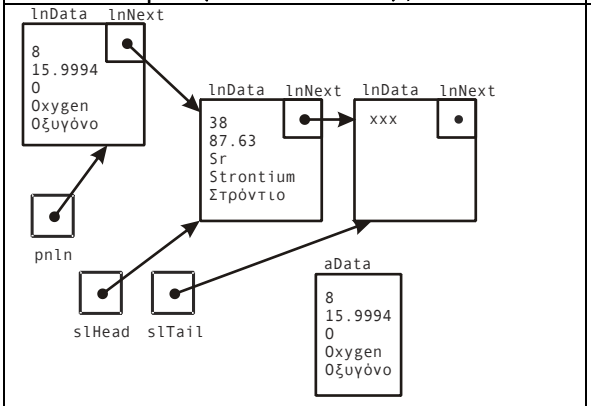
[9] Το όνομα από την STL.



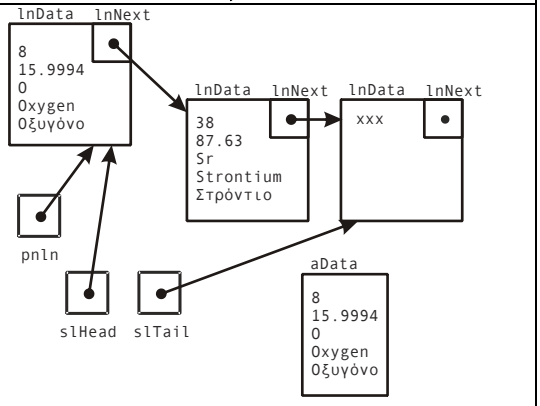
Σχ. 16-8α Μετά την εκτέλεση της εντολής:
`ListNode* pnln( new ListNode );`



Σχ. 16-8β Μετά την εκτέλεση της εντολής:
`pnln->lnData = aData;`



Σχ. 16-8γ Μετά την εκτέλεση της εντολής:
`pnln->lnNext = lst.slHead;`



Σχ. 16-8δ Μετά την εκτέλεση της εντολής:
`lst.slHead = pnln;`

κατάσταση που διαμορφώνεται. Η λίστα δεν έχει πειραχτεί ακόμη.

- Στον νέο κόμβο, στο μέλος *lnData* αντιγράφουμε τα δεδομένα που θέλουμε να εισαγάγουμε στη λίστα με την "`pnln->lnData = aData`". Η λίστα δεν έχει πειραχτεί ακόμη. Η κατάσταση φαίνεται στο Σχ. 16-8β.
- Με την "`pnln->lnNext = lst.slHead`" συμπληρώνεται ο νέος κόμβος. Το βέλος *lnNext* δείχνει τον μέχρι τώρα πρώτο κόμβο της λίστας. Όπως βλέπεις στο Σχ. 16-8γ έχουμε μια «νέα είσοδο» στη λίστα που κατά τα άλλα δεν έχει πειραχτεί.
- Με την "`lst.slHead = pnln`" ο νέος κόμβος ενσωματώνεται στη λίστα (Σχ. 16-8δ): Ξεκινώντας από την *slHead* περνάς από τον νέο κόμβο και από αυτόν πηγαίνεις στον προηγούμενο πρώτο (με το Στρόντιο).

Με το τέλος εκτέλεσης της συνάρτησης το βέλος *pnln* παύει να υπάρχει.

Σε δύο περιπτώσεις, μετά από αντιγραφές βελών, είχαμε δύο βέλη να δείχνουν τον ίδιο στόχο: στα Σχ. 16-8γ και 16-8δ, αλλά δεν υπήρχε οποιοδήποτε πρόβλημα.

Ο τρόπος αυτός δεν είναι ο μοναδικός· να και ένας άλλος το ίδιο σωστός:

```
ListNode* pnln( lst.slHead );
lst.slHead = new ListNode;
(lst.slHead)->lnData = aData;
(lst.slHead)->lnNext = pnln;
```

Η «*αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου*» μπορεί να γίνει στη λίστα όπως περίπου γίνεται στον πίνακα με τη *linSerch*:

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   ListNode* p( lst.slHead );
   while ( p->lnData != aData && p->lnNext != lst.slTail )
      p = p->lnNext;
   return (p->lnData == aData ? p : 0 );
```

```
} // SList_listSearch
```

Το βέλος *p* ξεκινάει δείχνοντας το στοιχείο που δείχνει και το *lst.slHead* και προχωρεί από στοιχείο σε στοιχείο με την "`p = p->lnNext`". Διασχίζει τη λίστα κόμβο προς κόμβο με την "`p = p->lnNext`" μέχρι

- Να βρει τα στοιχεία, αν υπάρχουν ("`p->lnData != aData`") ή
- Να φτάσει στον φρουρό ("`p->lnNext != lst.slTail`").

Αν έχει βρει την τιμή ("`p->lnData == aData`") η τιμή που επιστρέφει η συνάρτηση είναι *p* · αλλιώς επιστρέφει `0`.

Αν στον κόμβο-φρουρό αντιγράψουμε την τιμή που ψάχνουμε δεν θα απαλλαγούμε από τη μια σύγκριση της **while**; Βεβαίως, αυτή η τεχνική είναι η αντίστοιχη αυτής που μάθαμε στη γραμμική αναζήτηση σε πίνακα. Φυσικά, αυτό απαγορεύεται λόγω του "`const SList& lst`" αλλά ξέρουμε πώς θα αντιμετωπίσουμε: με τυποθεώρηση **const**.

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   SList& ncLst( const_cast<SList&>(lst) );
   ncLst.slTail->lnData = aData;
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
   return (p != lst.slTail ? p : 0 );
} // SList_listSearch
```

Η *ncLst* είναι η *lst* αλλά τύπου *SList*`&` –χωρίς "`const`"– και μπορούμε να δώσουμε:

```
   ncLst.slTail->lnData = aData;
```

Έτσι, η **while** γίνεται:

```
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
```

και η **return**:

```
   return (p != lst.slTail ? p : 0 );
```

(επίστρεψε το *p* αν δεν δείχνει τον φρουρό, αλλιώς το 0)

Η *linSearch()* επιστρέφει δείκτη στοιχείου πίνακα. Η *SList_listSearch* επιστρέφει βέλος προς κόμβο που έχει μέλος τύπου *GrElmn*. Έτσι, τώρα θα καλέσουμε την *editGrNameMM* αλλά κάπως διαφορετικά:

```
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
```

Εδώ είδαμε και πώς διασχίζουμε ολόκληρη τη λίστα: «*Το βέλος p ξεκινάει δείχνοντας το στοιχείο που δείχνει και το lst.slHead και προχωρεί από στοιχείο σε στοιχείο με την* "`p = p->lnNext`" *... μέχρι ... να φτάσει στον φρουρό* ("`p->lnNext != lst.slTail`").»

Πώς θα κάνουμε λοιπόν τη «*φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.*» Τη διασχίζουμε και φυλάγουμε το περιεχόμενο του κάθε κόμβου:

```
void saveUpdateList( fstream& bInOut, const SList& lst )
{
   for ( ListNode* p(lst.slHead); p != lst.slTail; p = p->lnNext )
      writeRandom( p->lnData, bInOut );
} // saveUpdateList
```

Παρομοίως γίνεται και η «*ανακύκλωση όλων των κόμβων της λίστας.*»

```
void SList_deleteAll( SList& lst )
{
   while ( lst.slHead != lst.slTail )
   {
      ListNode* p( lst.slHead );
      lst.slHead = (lst.slHead)->lnNext;
      delete p;
   }
   delete lst.slHead;
```

```
   lst.slTail = lst.slHead = 0;
} // SList_deleteAll
```

Πρόσεξε πώς γίνεται η διαγραφή του (εκάστοτε) πρώτου κόμβου:

- Φυλάγουμε στο *p* την τιμή του *lst.slHead* που δείχνει τον πρώτο κόμβο.
- Προχωρούμε το *lst.slHead* στον επόμενο κόμβο.
- Ανακυκλώνουμε αυτόν που δείχνει το *p*.

Αυτή η διαδικασία τελειώνει όταν "`lst.slHead == lst.slTail`": και τα δύο βέλη δείχνουν τον φρουρό. Η "`delete lst.slHead`" ανακυκλώνει και τον φρουρό.

Σε τι κατάσταση βρίσκεται η λίστα τώρα; Είναι άδεια; Όχι! *Δεν υπάρχει λίστα!* Δηλαδή δεν μπορείς να ξανακάνεις εισαγωγή στοιχείου με *SList_push_front()*. Αυτή η κατάσταση είναι ανιχνεύσιμη μετά την εκτέλεση των εκχωρήσεων:

```
   lst.slTail = lst.slHead = 0;
```

Να δούμε τώρα το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

// Συναρτήσεις GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode

struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // SList

void SList_push_front( SList& lst, const GrElmn& aData );
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData );
void SList_deleteAll( SList& lst );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
void saveUpdateList( fstream& bInOut, const SList& lst );
void elmntInList( SList& lst,
                  fstream& bInOut, int aa, ListNode*& pos );
```

```
int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      SList   lst;
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      SList_deleteAll( lst );
   }
   catch( ApplicXptn& x )
   // ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στις λίστες θα ξαναγυρίσουμε.

Πρόσεξε τώρα ένα σημαντικό πρόβλημα που έχουν αυτές οι τρεις λύσεις: *η καθυστέρηση που υπάρχει από τη στιγμή που κάνεις την ενημέρωση μέχρι να περάσει στο αρχείο.* Όταν λέμε «*καθυστέρηση*» την εννοούμε με όρους ανθρώπου και όχι υπολογιστή, π.χ. συμπληρώνω πέντε στοιχεία πάω για καφέ και τσιγάρο και επιστρέφω για τα υπόλοιπα. Αν λοιπόν συμβεί κάποιο είδος «system crash» χάνεις όλη τη δουλειά που έχεις κάνει εν τω μεταξύ (ενώ με το αρχικό πρόγραμμα το πολύ που μπορείς να χάσεις είναι η τελευταία εγγραφή).

Όμως, εκτός από αυτήν την περίπτωση, πρόσεξε ότι η *GrElmn_save()* μπορεί να ρίξει εξαίρεση *ApplicXptn*(::*cannotWrite*). Πώς μπορεί να γίνει αυτό; Για παράδειγμα, έχεις το αρχείο σε memory stick και κατά λάθος βγήκε από τη θέση του! Παρόμοια ζημιά μπορεί να γίνει –στα Παραδ. 2 και 3– και στην περίπτωση που η *GrElmn_load()* ρίξει εξαίρεση *ApplicXptn* (::*cannotRead*). Όπως είναι γραμμένα τα προγράμματα δεν υπάρχει πρόβλεψη για τέτοια καταστροφή. Για σκέψου όμως, κάτι θα μπορέσεις να σκαρφιστείς...

Ειρήσθω εν παρόδω ότι στις περιπτώσεις αυτές έχουμε και «διαρροή μνήμης» αλλά, αν έχεις χάσει όλη τη δουλειά που έκανες, μάλλον δεν σε ενδιαφέρει και τόσο.

16.13.1 Το Περίγραμμα *linSearch()*

Μετατρέπουμε σε περίγραμμα τη *linSearch* όπως την είδαμε στην §12.2.1

```
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
{
   if ( v == 0 && n > 0 )
      throw MyTmpltLibXptn( "linSearch",
                            MyTmpltLibXptn::noArray );
   int fv( -1 );
   if ( v != 0 && (0 <= from && from <= upto && upto < n) )
   {
      T* ncv( const_cast<T*>(v) );
      T  save( v[upto+1] );  // φύλαξε το v[upto+1]
      ncv[upto+1] = x;       // φρουρός
      int k( from );
      while ( v[k] != x )  ++k;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
      ncv[upto+1] = save;    // όπως ήταν στην αρχή
   // (from <= fv <= upto && v[fv] == x) ||
   //                (fv == -1 && (∀j:from..upto • v[j] != x))
   }
   return fv;
} // linSearch
```

Πέρα από τις αλλαγές από "`int`" σε "*T*" πρόσεξε τα εξής σημεία:

- Ελέγχουμε μήπως "`v == 0`". Στην περίπτωση αυτήν, αν το πλήθος στοιχείων του πίνακα *n* είναι 0 επιστρέφουμε τιμή "`-1`" ("δεν το βρήκα") αλλιώς ρίχνουμε εξαίρεση.
- Για να μπορεί να γίνει εξειδίκευση στον τύπο *T*, θα πρέπει να έχουμε για τον τύπο αυτόν:
  - Τελεστή εκχώρησης που να δουλεύει σωστά (αν δεν καταλαβαίνεις τι θα πει αυτό διάβασε, για παράδειγμα, αυτά που λέμε στην §16.8) για τις εκχωρήσεις "`ncv[upto+1] = x`" και "`ncv[upto+1] = save`". Στην περίπτωσή μας αυτός που υπάρχει αυτομάτως δουλεύει σωστά.
  - Τελεστή σύγκρισης "`!=`" για τη σύγκριση "`v[k] != x`" στη **while**.
  - *Δημιουργό αντιγραφής* που να δουλεύει σωστά! Τι είναι πάλι αυτό; Θα το μάθουμε αργότερα. Απλώς να πούμε ότι αυτός είναι που εκτελείται για τη δήλωση "`T save(v[upto+1])`".[10] Στην περίπτωσή μας και αυτός, όπως ο τελεστής εκχώρησης, υπάρχει αυτομάτως και δουλεύει σωστά.

Πάντως θα πρέπει να επιστήσουμε την προσοχή σου στις πολλές αντιγραφές που –αν τα αντικείμενα του τύπου *T* είναι μεγάλα– μπορεί να καθυστερούν την εκτέλεση του προγράμματός σου. Και να σκεφτείς ότι για τη σύγκριση χρειαζόμαστε μόνο το κλειδί, που συνήθως είναι πολύ πιο μικρό...

16.13.2 Χωρίς τη *linSearch()*

Η *linSearch* είναι ένα καλό εργαλείο από εκπαιδευτική άποψη, αλλά η C++ μας προσφέρει ένα περίγραμμα συνάρτησης –με το όνομα (`std::`)*find*– που κάνει την ίδια δουλειά: γραμμική αναζήτηση στα στοιχεία ενός πίνακα (και όχι μόνο). Θα το περιγράψουμε τώρα με τους περιορισμούς που υπάρχουν από αυτά που ξέρουμε μέχρι τώρα. Αργότερα, θα τη δούμε πληρέστερα.

Μπορείς, προς το παρόν, να σκέφτεσαι το περίγραμμα ως εξής:

```
template< typename T >
```

[10] Αν δεν θέλεις να ανησυχείς για δημιουργούς αντιγραφής και άλλα παρόμοια γράψε:

```
      T  save;
      save = v[upto+1];  // φύλαξε το v[upto+1]
```

```
T* find( T* first, T* last, T value )
```

όπου:

- Ο τύπος *T* έχει ορισμένη τη σύγκριση για ισότητα ("`==`").
- Η `[` *first*`..`*last* `)` είναι μια περιοχή βελών που μπορούμε να διασχίσουμε –ξεκινώντας από τη *first*– με μια μεταβλητή-βέλος `T* p` με την πράξη "`++p`". *Η last είναι η πρώτη θέση μετά την περιοχή αναζήτησης και δεν περιλαμβάνεται σε αυτήν.*
- *value* είναι η τιμή που αναζητούμε.

Η *find()* θα επιστρέψει βέλος προς την πρώτη θέση *fPos* για την οποίαν θα ισχύει η `*fPos == value`. Αν δεν βρει τη *value* επιστρέφει *last*.

Αν λοιπόν έχουμε έναν πίνακα ar –συμβατικό ή δυναμικό– με στοιχεία τύπου *T* για να ψάξουμε με τη *linSearch()* θα πρέπει να δώσουμε:

```
   unsigned int size, from, upto;
   int          ndx;
   T            value;
// . . .
   ndx = linSearch( ar, size, from, upto, value );
   if ( ndx >= 0 ) // βρέθηκε
   // . . .
```

ενώ για να ψάξουμε με τη *find* θα πρέπει να δώσουμε:

```
   unsigned int from, upto,;
   T*           pos;
   T            value;
// . . .
   pos = find( ar+from, ar+upto, value );//[11]
   if ( pos != ar+upto ) // βρέθηκε
   // . . .
```

Γυρνώντας στο Παράδ. 2, θα μπορούσαμε να γράψουμε τη *elmntInTable* ως εξής:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, GrElmn*& pos )
{
   GrElmn oneElmn( aa );
   GrElmn* lPos( find(grElmnTbl, grElmnTbl+nElmn, GrElmn(aa)) );
   if ( lPos == grElmnTbl+nElmn ) // αν δεν υπάρχει
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = &grElmnTbl[nElmn-1];
   }
   else
      pos = lPos;
} // elmntInTable
```

Να επισημάνουμε τις διαφορές:

- Η τελευταία παράμετρος είναι τώρα: `GrElmn*& pos`.
- Παρομοίως, η *lPos* είναι τύπου *GrElmn*`*`.
- Πρόσεξε πώς καλούμε τη *find* για να αναζητήσει τη `GrElmn(aa)` σε ολόκληρον τον πίνακα *grElmnTbl*:
  - Πρώτη παράμετρος ο πίνακας (η αρχή του),

[11] Ή, αν προτιμάς:

```
pos = find( &ar[from], &ar[upto], value );
```

- Δεύτερη παράμετρος μια θέση μετά το τελευταίο στοιχείο, Να υπενθυμίσουμε ότι: `grElmnTbl+nElmn` σημαίνει `&grElmnTbl[nElmn]`.

Μετά την κλήση ελέγχουμε το *αν δεν βρέθηκε η τιμή* ελέγχοντας τη συνθήκη *lPos* `==` *grElmnTbl*`+`*nElmn*.

Μπορούμε να χρησιμοποιήσουμε τη *find* για αναζητήσεις στη λίστα; Όχι για δύο λόγους:

- Δεν διασχίζουμε τη λίστα με την "`++p`" αλλά με την "`p = p->lnNext`".
- Αυτό που αναζητούμε δεν είναι ολόκληρος ο κόμβος αλλά μέλος του κόμβου.

Αργότερα θα γνωρίσουμε τα κατάλληλα εργαλεία για να το καταφέρουμε και αυτό.

16.13.3 "*reserved* + *incr*" ή "2**reserved*"

Ο τρόπος διαχείρισης της δυναμικής μνήμης δεν είναι ο καλύτερος. Ας πούμε ότι ξεκινούμε με *reserved* == *incr* στοιχεία στον πίνακα και τελικώς εισάγουμε *n* τιμές. Θα πάρουμε μνήμη *n*/*incr* φορές.[12] Την *k*-οστή φορά θα πάρουμε μνήμη για *k***incr* στοιχεία και θα κάνουμε (*k*-1)**incr* αντιγραφές.

Συνολικώς, το πλήθος των αντιγραφών θα είναι:

$$0*incr + 1*incr + 2*incr + \ldots + ((n/incr)-1)*incr = (n/incr)*((n/incr)-1)*incr/2 = O(n^2)$$

Στη βιβλιοθήκη της C++ (STL), κάθε φορά που χρειάζεται δυναμική μνήμη, παίρνει διπλάσια από αυτήν που ήδη έχει. Έτσι, αν ξεκινήσει με μνήμη για *incr* στοιχεία, την *k*-οστή φορά θα πάρει μνήμη για $2^{k-1}*incr$ στοιχεία και θα κάνει $2^{k-2}*incr$ αντιγραφές ($k \geq 2$: την πρώτη φορά δεν γίνονται αντιγραφές). Την τελευταία φορά θα έχουμε πάρει μνήμη για $n == 2^{p-1}*incr$ στοιχεία. Από αυτήν έχουμε: $p = log_2(n/incr)+1$.

Οι αντιγραφές που θα γίνουν:

$$2^0*incr + 2^1*incr + 2^2*incr + \ldots + 2^{p-2}*incr = incr*(2^{p-1}-1) = O(n)$$

Με αυτόν τον τρόπο δουλεύουν οι *παραχωρητές μνήμης* (memory allocators) των *περιεχόντων* (containers) της βιβλιοθήκης της C++.

Εμείς θα χρησιμοποιούμε τον «αργό τρόπο» μέχρι να γυρίσουμε στην STL, μια και όταν χρησιμοποιούμε δυναμική παραχώρηση μνήμης άλλα είναι αυτά που θα θέλουμε να δείξουμε. Πάντως, εσύ, αν θέλεις, μπορείς να χρησιμοποιείς τον «γρήγορο τρόπο». Πιο πάνω, στο Παράδ. 2, είδαμε τις εντολές:

```
renew( grElmnTbl, nElmn, nReserved+incr );
nReserved += incr;
```

Μπορείς να τις αλλάξεις σε:

```
renew( grElmnTbl, nElmn, 2*nReserved );
nReserved *= 2;
```

16.14 Προβλήματα Ασφάλειας

Στην παράγραφο αυτή θα ασχοληθούμε με δυο συστάσεις του (CERT 2009):

- ***Καθάρισε ευαίσθητες πληροφορίες αποθηκευμένες σε ανακυκλώσιμους πόρους που επιστρέφονται για να ξαναχρησιμοποιηθούν.***[13]

και

- ***Εξασφάλισε ότι δεν γράφονται στον δίσκο ευαίσθητα δεδομένα.***[14]

---

[12] Για να μη μπερδεύεσαι με ατελείς διαιρέσεις θεώρησε ότι $n = 2^N*incr$.

[13] Σύσταση *MEM*03: "Clear sensitive information stored in reusable resources returned for reuse."

[14] Σύσταση *MEM*06: "Ensure that sensitive data is not written out to disk."

Θα ξεκινήσουμε ξαναδίνοντας ένα παράδειγμα που είδαμε στην §16.3. Με τη βοήθεια δύο βελών:

```
   double* a1( new double );
   double* a2;
```

είχαμε υλοποιήσει δύο δυναμικές μεταβλητές *a1, *a2. Αφού τους δώσαμε τιμές, η

```
   cout << *a1 << "  " << *a2 << endl;
```

μας έδωσε:

```
1.23  4.56
```

Στη συνέχεια τις ανακυκλώσαμε:

```
   delete a1;  delete a2;
```

ξαναϋλοποιήσαμε την *a1 και είδαμε την τιμή της πριν προσπαθήσουμε να την ορίσουμε:

```
   a1 = new double;
   cout << *a1 << endl;
```

Αποτέλεσμα:

```
4.56
```

Πώς έγινε αυτό; Η *a1 υλοποιήθηκε τη δεύτερη φορά στη μνήμη που ελευθερώθηκε με την ανακύκλωση της *a2. Φυσικά, αυτό στηρίζεται στο ότι:

- Ο "`delete`", παρά το όνομά του, δεν κάνει οποιαδήποτε διαγραφή.
- Ο "`new`", παρά το όνομά του μπορεί να μας φέρει «παλιά» πράγματα.

Παρόμοια πράγματα μπορεί να συμβούν και με τη μνήμη στοίβας:

```
#include <iostream>
using namespace std;

void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
}

void f2()
{
   char q[5];
   cout << "from f2:  q = " << q << endl;
}

int main()
{
   f1();
   f2();
}
```

Αυτό το πρόγραμμα θα δώσει:

```
from f1:  m = abcd
from f2:  q = abcd
```

Τι έγινε εδώ; Με την κλήση της *f1()* η τοπική μεταβλητή *m* υλοποιείται σε μνήμη που δίνεται από τη στοίβα και παίρνει τιμή `"abcd"`. Με το τέλος της εκτέλεσης της *f1()* η μνήμη αυτή απελευθερώνεται. Εδώ δεν έχουμε ούτε "`new`" ούτε "`delete`"· όλα γίνονται αυτομάτως. Όταν στη συνέχεια ενεργοποιείται η *f2()*, με τον τρόπο που λειτουργεί η στοίβα, η μνήμη που είχε δοθεί στην *m* της *f1()* δίνεται στην *q* της *f2()*.

Τι κοινό έχουν τα παραπάνω παραδείγματα; *Και στις δύο περιπτώσεις πήραμε μνήμη για να κάνουμε τη δουλειά μας και μετά την ελευθερώσαμε. Αλλά τα δεδομένα που αποθηκεύτηκαν έχουν παραμείνει.*

Αν τα δεδομένα είναι «ευαίσθητα», για παράδειγμα κάποια *συνθηματικά πρόσβασης* (passwords), θα θέλαμε να έχουμε τη σιγουριά ότι όταν δεν τα χρησιμοποιούμε στο πρόγραμμά μας δεν υπάρχουν στη μνήμη του υπολογιστή. Ένας απλός τρόπος να το σιγουρέ-

ψουμε είναι να σβήσουμε πριν ανακυκλωθούν. Υπάρχει μια πάγια τεχνική για τη δουλειά αυτή: γεμίζουμε την περιοχή της μνήμης που μας ενδιαφέρει με κάποιον χαρακτήρα –συνήθως τον `'\0'`– με τη συνάρτηση της C *memset()*:[15]

```
   memset( a2, '\0', sizeof(double) );  delete a2;
```

και στην *f*1:

```
void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
   memset( m, '\0', strlen(m) );
}
```

Σημείωση: ▶

Η *memset()* έχει επικεφαλίδα:

```
void* memset( void* s, int c, size_t n );
```

και λειτουργεί ως εξής: βάζει *n* αντίγραφα του χαρακτήρα *c* στην περιοχή της μνήμης που αρχίζει από τη διεύθυνση *s*. Επιστρέφει την τιμή του *s*. ◀

Πάντως τα πράγματα μπορεί και να μη γίνουν όπως τα περιμένεις: Μερικοί μεταγλωττιστές, υπερβολικώς «ευφυείς», αν βρουν εντολές που αλλάζουν το περιεχόμενο μιας περιοχής της μνήμης που στη συνέχεια ανακυκλώνεται –χωρίς να μεσολαβει χρήση του περιεχόμενου– *δεν μεταγλωττίζουν τις εντολές αλλαγής για να κάνουν το πρόγραμμα ταχύτερο!* Το ΛΣ μπορεί να έχει εργαλεία για τη λύση του προβλήματος.[16]

Η δεύτερη σύσταση έχει να κάνει με τη μνήμη συνολικώς και όχι μόνον τη δυναμική:

- Στην §16.3 μιλήσαμε για τη διαδικασία ανταλλαγών μνήμης και είπαμε ότι όλη η μνήμη που απαιτείται για το κάθε πρόγραμμα βρίσκεται κατ' αρχήν στον δίσκο. Αν λοιπόν κάποιος «κακός» ξέρει καλά το ΛΣ αλλα δεν μπορεί να «σπάσει» τις προστασίες του Συστήματος Διαχείρισης Βάσεων Δεδομένων (DBMS) μπορεί να «κλέβει» στιγμιότυπα μνήμης προγραμμάτων που χρησιμοποιούν τα δεδομένα που τον ενδιαφέρουν.
- Ορισμένα ΛΣ, όταν έχουν *μη κανονικό* τερματισμό εκτέλεσης προγράμματος δίνουν μια **απόρριψη** (περιεχόμενου της) **μνήμης** του (memory dump) σε κάποιο αρχείο, για να διευκολύνουν τον εντοπισμό του προβλήματος. Ο «κακός», που λέγαμε παραπάνω, θα μπορούσε να προκαλεί μη κανονικούς τερματισμούς προγραμμάτων και να κλέβει τα αρχεία με τα περιεχόμενα της μνήμης.

  Αυτά τα προβλήματα μπορεί να αντιμετωπισθούν μόνον μέσω του ΛΣ.

16.15 Ανακεφαλαίωση

Στο πρόγραμμά μας μπορούμε να παίρνουμε μνήμη για να υλοποιήσουμε μεταβλητές ή πίνακες σύμφωνα με τις ανάγκες που προκύπτουν κατά τη διάρκεια της εκτέλεσης. Αν η *p* είναι μεταβλητή-βέλος τύπου *T*`*` τότε:

- Με την εντολή "`p = new T`" παίρνουμε μνήμη για την υλοποίηση μιας *δυναμικής μεταβλητής* τύπου *T*. Η *p* δείχνει αυτήν τη μεταβλητή που τη χειριζόμαστε ως "`*p`".
- Η `*`*p* είναι όπως οποιαδήποτε μεταβλητή τύπου *T* και ως τέτοια τη χειριζόμαστε.
- Όταν δεν χρειαζόμαστε την `*`*p* την ανακυκλώνουμε με την "`delete p`". Αμέσως μετά τη "`delete p`" η `*`*p* δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.
- Με την "`p = new T[`*Π*`]`" ζητούμε μνήμη για έναν δυναμικό πίνακα με στοιχεία τύπου *T*. Το πλήθος τους καθορίζεται από την τιμή *n* της παράστασης *Π* που θα πρέπει να είναι ακέραιου τύπου και θετική.

---

[15] Θα πρέπει να έχεις δώσει "`#include <string>`" για να τη χρησιμοποιήσεις.

[16] Για παράδειγμα, στο Windows API υπάρχει η συνάρτηση *SecureZeroMemory()*.

- Τα στοιχεία του πίνακα είναι `p[0]`, `p[1]`, … `p[`*n*`-1]`. Χειριζόμαστε τον πίνακα όπως έναν συνήθη πίνακα με *n* στοιχεία τύπου *T*.
- Όταν δεν χρειαζόμαστε τον δυναμικό πίνακα τον ανακυκλώνουμε με την "`delete p[]`".
- Όταν ένα βέλος δεν δείχνει συγκεκριμένο στόχο –π.χ. μετά από δήλωση χωρίς αρχική τιμή ή μετά από ανακύκλωση του στόχου– βάζουμε στο βέλος τιμή "`0`" (ή "`NULL`" ή (*std*`::`)"*nullptr*").
- Η απόπειρα ανακύκλωσης ήδη ανακυκλωμένης μνήμης ("`delete p; delete p`") είναι σοβαρό λάθος. Η "`delete 0`" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "`delete[] 0`".

Έχεις δυνατότητα να χειριστείς τη δυναμική μνήμη με τα εργαλεία της C. Αλλά:

- Μνήμη που παίρνεις με *malloc(), calloc()* θα πρέπει να ανακυκλώνεται με *free()* και όχι με `delete`.
- Μνήμη που παίρνεις με `new`, `new … […]` θα πρέπει να ανακυκλώνεται με `delete`, `delete[]` αντιστοίχως.

Ακόμη καλύτερα, αν αυτό είναι δυνατόν:

- Μην αναμειγνύεις τους δύο τρόπους στο ίδιο πρόγραμμα.

Εκτός από το «παράπτωμα» της επαναλαμβανόμενης ανακύκλωσης τα σοβαρότερα προβλήματα με τη χρήση δυναμικής μνήμης είναι τα εξής:

- Η *διαρροή μνήμης*, δηλαδή η απώλεια ελέγχου σε τμήμα δυναμικής μνήμης που παραμένει δεσμευμένη από το πρόγραμμά μας αλλά δεν μπορούμε να τη χειριστούμε διότι δεν υπάρχει βέλος που να τη δείχνει.
- Το *μετέωρο βέλος*, δηλαδή βέλος που δείχνει τμήμα δυναμικής μνήμης που έχει ήδη ανακυκλωθεί.

Ασκήσεις

Α Ομάδα

16-1 Στην §14.7.3 γράψαμε τη *swap()* για ορμαθούς της C χρησιμοποιώντας ως ενδιάμεσο ενταμιευτή μια μεταβλητή τύπου *string*. Τότε δεν ξέραμε από δυναμική μνήμη και πήραμε βοήθεια από τον τύπο *string* (που ξέρει). Τώρα, που έμαθες τη χρήση της δυναμικής μνήμης, να την ξαναγράψεις. Προσεκτικά...

16-2 Ας ξαναδούμε το πρόγραμμα με το οποίο δοκιμάζαμε τον αλγόριθμο του Horner για την τιμή πολυωνύμου (§9.4, Παράδ. 1). Στην πρώτη μορφή, είχαμε δηλώσει:

```
const int N = 20;
double a[N];
```

έναν αρκετά μεγάλο πίνακα a που θα μπορούσε να χωρέσει συντελεστές πολυωνύμου μέχρι και 19ου βαθμού. Έτσι όμως, από τη μια δεν μπορούμε να υπολογίσουμε τιμές πολυωνύμου 20ου βαθμού και από την άλλη αν έχουμε πολυώνυμα μικρού βαθμού πολλά στοιχεία του πίνακα είναι άχρηστα.

Τώρα, χρησιμοποιώντας δυναμικό πίνακα συντελεστών, μπορείς να έχεις ακριβώς αυτά που χρειάζεσαι.

16-3 Τροποποιώντας την *SList_deleteAll()*, γράψε μια *SList_clear()* που «καθαρίζει» τη λίστα –ανακυκλώνει όλους τους κόμβους αλλά όχι τον φρουρό– ώστε να μπορεί να χρησιμοποιηθεί ξανά.

κεφάλαιο

17

* Εσωτερική Παράσταση Δεδομένων

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις

- πώς παριστάνονται στη μνήμη του ΗΥ οι τιμές διαφόρων τύπων και
- περιορισμούς και προβλήματα που προκύπτουν από τους τρόπους παράστασης.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις πιο αξιόπιστα προγράμματα.

Έννοιες κλειδιά:

- *παράσταση ακεραίων*
- *παράσταση πραγματικών*
- *διαχείριση δυαδικών ψηφίων*
- *ψηφιοπράξεις*
- *σφάλματα παράστασης*
- *σφάλματα πράξεων*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Αριθμητικά Συστήματα:
Ένας από τους πιό γνωστούς τρόπους κωδικοποίησης αριθμητικών πληροφοριών είναι η κωδικοποίηση στο γνωστό **δεκαδικό σύστημα**, στο οποίο χρησιμοποιούμε δέκα διαφορετικά σύμβολα, τα ψηφία: 0,1,2,3,4,5,6,7,8,9.

Για να παραστήσουμε στο σύστημα αυτό έναν ακέραιο αριθμό (θετικό ή αρνητικό), χρησιμοποιούμε ένδεκα σύμβολα: τα δέκα ψηφία και το πρόσημο "**-**" (μείον). Πολλές φορές χρησιμοποιούμε και το πρόσημο "**+**" (συν), για να ξεχωρίζουμε ευκολότερα τους θετικούς αριθμούς από τους αρνητικούς (π.χ. **-**5109, 2048, +2048). Ακόμη, στην παράσταση ενός κλασματικού αριθμού χρειαζόμαστε και την υποδιαστολή (ευρωπαϊκές χώρες: "**,**", ΗΠΑ. Μεγ. Βρεττανία: "**.**"), που διαχωρίζει το ακέραιο μέρος του αριθμού από το κλασματικό (π.χ. 48,25).

Το κοινό δεκαδικό αριθμητικό σύστημα είναι **θεσιακό** (positional), διότι η τιμή που παριστάνει κάθε ψηφίο ενός αριθμού εξαρτάται από τη θέση που έχει στην παράσταση του αριθμού. Για παράδειγμα, το πρώτο ψηφίο 3 στον αριθμό 6343 σημαίνει την τιμή 300 (3×10^2), ενώ το δεύτερο ψηφίο 3 την τιμή 3 (3×10^0).

Εκτός από τα θεσιακά συστήματα αριθμήσεως υπάρχουν και μη θεσιακά συστήματα, όπως είναι το γνωστό –**προσθετικό** (additive)– ρωμαϊκό σύστημα (π.χ. XXXIII = 10 + 10 + 10 + 1 + 1 + 1 = 33) ή το Ελληνικό (ρβ′ = 100 + 2 = 102), όπου τα σύμβολα που χρησιμοποιούμε έχουν την ίδια τιμή ανεξαρτήτως της θέσης τους στον αριθμό που παριστάνουν.

Σε ένα θεσιακό αριθμητικό σύστημα κάθε φυσικός αριθμός N μπορεί να παρασταθεί με ένα πολυώνυμο της εξής μορφής:

$$N = \psi_{L-1}B^{L-1} + \psi_{L-2}B^{L-2} + \ldots + \psi_0 B^0 \qquad (1)$$

Ο αριθμός B λέγεται **βάση** (base, radix) του αριθμητικού συστήματος και υποθέτουμε γενικώς ότι είναι μεγαλύτερος από το 1 (υπάρχουν όμως και συστήματα με αρνητική βάση). Οι συντελεστές ψ_k (k: $0..L-1$) που μπορεί να πάρουν τιμές στο $[0..B)$ ($0 \le \psi_k < B$), αποτελούν τα διαδοχικά ψηφία του αριθμού N ο οποίος έχει L θέσεις (ή ψηφία). Έτσι, η βάση B καθορίζει το πλήθος των διαφορετικών ψηφίων ενός αριθμητικού συστήματος. Στο δεκαδικό αριθμητικό σύστημα έχουμε βάση $B = 10$ και τα δέκα διαφορετικά ψηφία είναι: 0, 1, 2,...,9.

Δηλαδή επιλέγοντας τη βάση, δημιουργούμε ένα αριθμητικό σύστημα. Όταν π.χ. η βάση B = 2 ή 3, 8, 10, 16, τότε το σύστημα λέγεται **δυαδικό** ή αντιστοίχως **τριαδικό, οκταδικό, δεκαδικό, δεκαεξαδικό**. Έτσι λοιπόν ο αριθμός N = 7635, του δεκαδικού θεσιακού συστήματος, μπορεί να γραφτεί με την εξής μορφή:

$$\mathrm{N} = 7\times10^3 + 6\times10^2 + 3\times10^1 + 5\times10^0 = 7635$$

όπου $\psi_3 = 7$, $\psi_2 = 6$, $\psi_1 = 3$ και $\psi_0 = 5$.

Στο οκταδικό σύστημα η βάση B = 8 και τα διάφορα ψηφία αυτού του συστήματος είναι: 0,1,2,...,7. Επομένως ο αριθμός N = 7635, του οκταδικού συστήματος, παριστάνει την τιμή του δεκαδικού αριθμού 5149, επειδή:

$$N = 7635_8 = 7\times8^3 + 6\times8^2 + 3\times8^1 + 5\times8^0 = 5149_{10}$$

Βλέπουμε λοιπόν ότι η ίδια ακολουθία ψηφίων (π.χ. 7635) αντιπροσωπεύει άλλη τιμή στο οκταδικό σύστημα και άλλη στο δεκαδικό. Έτσι, για να μη γίνεται σύγχυση, συχνά γράφουμε σαν κάτω δείκτη, στο δεξιό μέρος της ακολουθίας ψηφίων ενός αριθμού τη βάση του αριθμητικού συστήματος που χρησιμοποιούμε (πάντα στο δεκαδικό συμβολισμό). Στην

παρουσίαση των αριθμών στα θεσιακά συστήματα σημειώνουμε, για λόγους απλοποίησης, μόνο τους όρους ψ_k (σε κατιούσα σειρά) και παραλείπουμε τις δυνάμεις B.

17.1 Παράσταση Φυσικών

Στους ψηφιακούς ΗΥ χρησιμοποιείται το ***δυαδικό*** (binary) *αριθμητικό σύστημα*, για λόγους τεχνολογικής αξιοπιστίας (αλλά και θεωρητικούς). Η Φυσική και η Ηλεκτρονική έχουν να προτείνουν **συστήματα δύο καταστάσεων** (two state systems), στα οποία μπορούμε

- να ανιχνεύουμε την κατάσταση που βρίσκεται το σύστημα και
- να το βάζουμε στην κατάσταση που θέλουμε.

Σε ένα τέτοιο σύστημα, «βαφτίζουμε» τη μια κατάσταση "`0`" και την άλλην "`1`"· παριστάνουμε δηλαδή τα δύο ψηφία του δυαδικού συστήματος ($B = 2$).

Σύμφωνα με αυτά που είπαμε παραπάνω, στο δυαδικό σύστημα κάθε φυσικός αριθμός παριστάνεται ως άθροισμα δυνάμεων του 2. Για παράδειγμα, ο δεκαδικός αριθμός $N = 12$ ($8 + 4 = 2^3 + 2^2$), μπορεί να παρασταθεί ως εξής:

$$N = 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = 1100_2$$

όπου $\psi_3 = 1$, $\psi_2 = 1$, $\psi_1 = 0$ και $\psi_0 = 0$.

Συχνά στα υπολογιστικά συστήματα χρησιμοποιείται και το **δεκαεξαδικό** (hexadecimal) αιθμητικό σύστημα, για το οποίο $B = 16$ και τα ψηφία του είναι:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`

Τα νέα σύμβολα *A*, *B*, *C*, *D*, *E* και *F* (ή *a*, *b*, *c*, *d*, *e*, *f*) αντιστοιχούν στις τιμές των (δεκαδικών) αριθμών: δέκα, ένδεκα, δώδεκα, δεκατρία, δεκατέσσερα και δεκαπέντε. Κατά συνέπεια ο δεκαεξαδικός αριθμός 153 σημαίνει:

$$153_{16} = 1\times16^2 + 5\times16^1 + 3\times16^0 = 256 + 80 + 3 = 339_{10}$$

και ο δεκαεξαδικός αριθμός $1A3F$ σημαίνει:

$$\begin{aligned} 1A3F_{16} &= 1\times16^3 + A\times16^2 + 3\times16^1 + F\times16^0 \\ &= 1\times16^3 + 10\times16^2 + 3\times16^1 + 15\times16^0 \\ &= 6719_{10} \end{aligned}$$

Ο Πίν. 17-1 δείχνει την παράσταση των ακεραίων αριθμών από 1 μέχρι 20 σε διάφορα θεσιακά αριθμητικά συστήματα.

Από τον Πίν. 17-1 φαίνεται ότι όταν μικραίνει η βάση του αριθμητικού συστήματος μεγαλώνει το πλήθος των ψηφίων που χρειάζονται για να παρασταθεί ένας αριθμός. Για παράδειγμα, ο αριθμός 9 χρειάζεται ένα ψηφίο στο 16-δικό και στο 10-δικό σύστημα, δυό στο 8-δικό, τρία στο 3-δικό και τέσσερα στο 2-δικό σύστημα.

Το πλήθος Π των διαφόρων αριθμών που μπορεί να γραφούν σε L θέσεις ενός αριθμητικού συστήματος βάσεως B δίνεται από τον τύπο:

$$\Pi = B^L \tag{2}$$

Αν, ας πούμε, πάρουμε $B = 2$ και $L = 4$, τότε μπορούμε να ξεχωρίσουμε 16 ($=2^4$) διαφορετικούς δυαδικούς αριθμούς (ή συνδυασμούς), δηλ. τους αριθμούς: 0000, 0001, 0010,... 1111.

Αν τώρα γνωρίζουμε το πλήθος των ακεραίων αριθμών Π, που θέλουμε να παραστήσουμε στο αριθμητικό σύστημα με βάση το B, τότε ο απαιτούμενος αριθμός θέσεων L καθορίζεται από τον τύπο:

$$L = \log_B\Pi \tag{3}$$

Έτσι, για να παραστήσουμε τους πρώτους 16 φυσικούς αριθμούς, από 0 μέχρι 15, στο δυαδικό σύστημα χρειαζόμαστε $\log_2 16 = 4$ θέσεις. Ενώ για την παράσταση των πρώτων 9 αριθμών στο τριαδικό σύστημα χρειαζόμαστε $\log_3 9 = 2$ θέσεις. Μπορείς να ελέγξεις την ορθότητα του τύπου (3) και από τον Πίν. 17-1.

Ακόμη, από τον Πίν. 17-1, μπορείς να δεις ότι το οκταδικό και το δεκαεξαδικό είναι «συμπυκνώσεις» του δυαδικού συστήματος. Π.χ. το "20_{10}" στο δυαδικό γράφεται "10100". Αν

10δικό	16δικό	8δικό	3δικό	2δικό
0	00	00	000	00000
1	01	01	001	00001
2	02	02	002	00010
3	03	03	010	00011
4	04	04	011	00100
5	05	05	012	00101
6	06	06	020	00110
7	07	07	021	00111
8	08	10	022	01000
9	09	11	100	01001
10	0A	12	101	01010
11	0B	13	102	01011
12	0C	14	110	01100
13	0D	15	111	01101
14	0E	16	112	01110
15	0F	17	120	01111
16	10	20	121	10000
17	11	21	122	10001
18	12	22	200	10010
19	13	23	201	10011
20	14	24	202	10100

Πίν. 17-1 Παράσταση αριθμών σε διάφορα αριθμητικά συστήματα.

δεις αυτήν την παράσταση ως δυο τριάδες: "010 | 100" και κάθε μια από αυτές ως ψηφίο του οκταδικού συστήματος, παίρνεις τον "24_8". Παρομοίως, αν τη δεις ως δυο τετράδες "0001 | 0100" και τις γράψεις ως ψηφία του δεκαεξαδικού παίρνεις: "14_{16}". Φυσικά, αυτές οι ιδιότητες ξεκινούν από το ότι $8 = 2^3$ και $16 = 2^4$.

Στην §2.5 (Πίν. 2-1) είδαμε ότι η C++, για την παράσταση φυσικών αριθμών, μας δίνει τους ακέραιους τύπους χωρίς πρόσημο: `unsigned char`, `unsigned int`, `unsigned long`. Ο `unsigned char` αποθηκεύει τιμές σε μια ψηφιολέξη[1] των οκτώ δυαδικών ψηφίων· σύμφωνα με τον τύπο (2), μπορεί να παραστήσει $2^8 = 256$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 255. Ο `unsigned short int` δουλεύει με μια λέξη (δυο ψηφιολέξεις) με 16 δυαδικά ψηφία και μπορεί να παραστήσει $2^{16} = 65\,536$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 65 535. Τέλος, ο `unsigned long` μπορεί να παραστήσει $2^{32} = 4\,294\,967\,296$ τιμές, από 0 μέχρι 4 294 967 295. Και ο `unsigned int`; Μπορεί να σαν τον `unsigned long` ή σαν τον `unsigned short int`.

Στη συνέχεια θα δούμε έναν τρόπο για να «σκαλίζουμε» τις εσωτερικές παραστάσεις.

Στις §1.8.1 και §1.8.2 είδαμε ότι η C++ μας δίνει τη δυνατότητα να γράφουμε στο πρόγραμμά μας φυσικούς αριθμούς στο δεκαεξαδικό σύστημα βάζοντας το πρόθεμα "`0x`" ή "`0X`" και στο οκταδικό σύστημα βάζοντας το πρόθεμα "`0`". Π.χ., οι εντολές:

```
k = 255;   k = 0XFF;   k = 0xff;   k = 0377;
```

κάνουν ακριβώς το ίδιο πράγμα. Το "255" είναι μια αριθμητική σταθερά στο δεκαδικό σύστημα. Η ίδια τιμή στο δεκαεξαδικό σύστημα γράφεται "FF_{16}" και στο οκταδικό "377_8". Στη

[1] Οι τιμές των μεγεθών που δίνουμε εδώ δεν είναι υποχρεωτικές, είναι απλώς συνηθισμένες. Θυμίσου ότι όπως λέγαμε στο Κεφ. 2 το υποχρεωτικό είναι ότι:

$$\texttt{sizeof(char)} \le \texttt{sizeof(short)} \le \texttt{sizeof(int)} \le \texttt{sizeof(long)}$$

C++, η δεκαεξαδική τιμή γράφεται "`0XFF`" και η οκταδική "`0377`". Πρόσεξε καλά την οκταδική γραφή:

- ***Όταν μια αριθμητική σταθερά ακέραιου τύπου αρχίζει με "`0`" (μηδέν) είναι οκταδικός και και όχι δεκαδικός αριθμός.***

17.2 Παράσταση Ακεραίων – Αρνητικοί Αριθμοί

Πώς μπορούμε να παραστήσουμε σε μια ψηφιολέξη ακέραιους θετικούς ή αρνητικούς; Αφού το πρόσημο μπορεί να είναι "`+`" ή "`-`", μπορούμε να το παραστήσουμε με ένα δυαδικό ψηφίο: "`0`" για το "`+`" και "`1`" για το "`-`". Έτσι, θα μας μείνουν άλλα 7 δυαδικά ψηφία για την απόλυτη τιμή, που θα μπορεί να είναι από 0 μέχρι $2^7 - 1 = 127$. Να λοιπόν ένας τρόπος για να παραστήσουμε ακέραιες τιμές από –127 μέχρι +127. Αυτός ο τρόπος παράστασης λέγεται «**πρόσημο - απόλυτη τιμή**». Αλλά πρόσεξε: εδώ έχουμε 255 τιμές, ενώ στον τύπο `unsigned char` παριστάνουμε 256 διαφορετικές τιμές από –128 μέχρι 127! Ναί, χάσαμε μια τιμή, διότι το 0 (μηδέν) παριστάνεται με δυο διαφορετικούς τρόπους: ως "`+0`" (`00000000`) και ως "`-0`" (`10000000`).

Συνήθως, στους υπολογιστές μας θα βρούμε την παράσταση αρνητικών με το "**συμπλήρωμα ως προς 2**" (2's complement). Ας δούμε ένα

Παράδειγμα

Όπως είδαμε, το 12 παριστάνεται σε μια ψηφιολέξη, ως:

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

Για να βρούμε το συμπλήρωμα ως προς 2:

Βήμα 1: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1

(αυτό είναι το **συμπλήρωμα ως προς 1**)

Βήμα 2: προσθέτουμε το 1

```
  1 1 1 1 0 0 1 1
+ 0 0 0 0 0 0 0 1
-----------------
  1 1 1 1 0 0 1 0
  7 6 5 4 3 2 1 0
```

Αυτήν την παράσταση χρησιμοποιούμε για να παραστήσουμε το "`-12`".

☜☜☜

Το ότι η τιμή είναι αρνητική φαίνεται –όπως και στην «πρόσημο-απόλυτη τιμή»– από το "`1`" στο δυαδικό ψηφίο 7. Τί κερδίσαμε από όλα αυτά; Ας του προσθέσουμε την παράσταση του 20 (`00010100`) αγνοώντας το ότι το πρώτο ψηφίο είναι πρόσημο:

```
  1 1 1 1 0 1 0 0
+ 0 0 0 1 0 1 0 0
-----------------
1 0 0 0 0 1 0 0 0
```

Αν ξεχάσουμε το πιο σημαντικό ψηφίο, που είναι "`1`", τα υπόλοιπα οκτώ ψηφία παριστάνουν το 8, που είναι σωστό: (-12) + 20 = 8.

Ένας άλλος τρόπος για να πούμε το ίδιο πράγμα είναι ο εξής: Το αποτέλεσμα της πράξης ήταν $2^8+2^3 = 264$. Κρατάμε το υπόλοιπο της διαίρεσής του δια 256 (= 2^8). Γι' αυτό, αυτή η αριθμητική λέγεται **αριθμητική υπολοίπων** (modulo arithmetic).

Δηλαδή:

- ***Όταν παριστάνουμε τους αρνητικούς με συμπλήρωμα ως προς 2 κάνουμε πρόσθεση χωρίς να ενδιαφερόμαστε για τα πρόσημα των προσθεταίων.***

Να λοιπόν πώς δουλεύει ο υπολογιστής στην περίπτωση αυτή:

- οι αρνητικοί παριστάνονται με συμπλήρωμα ως προς 2,
- κατά την πρόσθεση το ψηφίο προσήμου δεν έχει διαφορετική μεταχείριση από τα άλλα,

- η πρόσθεση γίνεται με αριθμητική υπολοίπων ως προς 2^N, όπου N το πλήθος δυαδικών ψηφίων που χρησιμοποιούνται για την παράσταση.

  Ποιος είναι ο μέγιστος αριθμός που μπορούμε να παραστήσουμε; Είναι ο

  7 6 5 4 3 2 1 0
  0 1 1 1 1 1 1 1 = 127_{10}

  Η ελάχιστη τιμή είναι -128 και παριστάνεται ως:

  7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 0 = -128_{10}

Μάλλον θα θέλεις κάποια βοήθεια για να βρεις το "**-128**". Λοιπόν: αφού έχει "**1**" στο ψηφίο 7, είναι αρνητικός. Ας εφαρμόσουμε *αντιστρόφως* αυτά που κάναμε για να υπολογίσουμε το συμπλήρωμα ως προς 2:

Βήμα 1: αφαιρούμε το 1

```
  7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 0
-               1
-----------------
  0 1 1 1 1 1 1 1
  7 6 5 4 3 2 1 0
```

Βήμα 2: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

10000000

που είναι το $2^7 = 128$.

Το μηδέν παριστάνεται με έναν μοναδικό τρόπο: "**00000000**".

Η C++ έχει τρεις τύπους που τους χειρίζεται με τον τρόπο αυτόν:

- (**signed**) **char**, σε 8 ψηφία. Παριστάνει τιμές από **-**128 μέχρι 127 με αριθμητική υπολοίπου ως προς $2^8 = 256$.
- **short int**, σε 16 ψηφία. Παριστάνει τιμές από **-**32 768 μέχρι 32 767 με αριθμητική υπολοίπου ως προς $2^{16} = 65\,536$.
- **int** και **long int**, σε 32 ψηφία. Παριστάνει τιμές από **-**2 147 483 648 μέχρι 2 147 483 647 με αριθμητική υπολοίπου ως προς $2^{32} = 4\,294\,967\,296$.

Να (ξανα)τονίσουμε ότι *τα παραπάνω μεγέθη δεν καθορίζονται από το πρότυπο* της γλώσσας. Ας πούμε ότι είναι συνηθισμένες τιμές.

Στη συνέχεια θα δούμε και μερικούς ακόμη ακέραιους τύπους.

Πού θα μας χρειαστούν όλα αυτά; Το συζητούμε στην συνέχεια.

17.2.1 * Ακέραιοι Τύποι του C99

Στο πρότυπο C99 (ISO/IEC 1999) της C εισάγονται ακέραιοι τύποι με 64 δυαδικά ψηφία: **long long int** και **unsigned long long int**.

- Ελάχιστη τιμή του **long long int**

  LLONG_MIN: **-9223372036854775807** = $-(2^{63} - 1)$

- Μέγιστη τιμή του **long long int**

  LLONG_MAX: **+9223372036854775807** = $2^{63} - 1$

- Μέγιστη τιμή του **unsigned long long int**

  ULLONG_MAX: **18446744073709551615** = $2^{64} - 1$

Οι **long long int** και **unsigned long long int** προβλέπονται και στο C++11.

Πέρα από αυτούς τους ακέραιους 64 δυαδικών ψηφίων, το C99 υποδεικνύει και μερικούς ορισμούς-μεταονομασίες τύπων. Η πρώτη οικογένεια περιλαμβάνει ορισμούς ονομάτων της μορφής **int***N***_t** και **uint***N***_t**. Το *N* υποδηλώνει το πλήθος δυαδικών ψηφίων. Ο gcc (Dev C++), στο **stdint.h**, έχει τους εξής σχετικούς ορισμούς:

```
typedef signed char         int8_t;
typedef unsigned char       uint8_t;
typedef short               int16_t;
typedef unsigned short      uint16_t;
typedef int                 int32_t;
```

```
typedef unsigned           uint32_t;
typedef long long          int64_t;
typedef unsigned long long uint64_t;
```

Τι κερδίζουμε με αυτούς; Όταν θέλουμε μια ακέραιη μεταβλητή με 32 ψηφία δεν χρειάζεται να ψάχνουμε αν θα τη δηλώσουμε υποχρεωτικώς `long int` ή μας κάνει και ο `int`· δηλώνουμε:

```
   int32_t x;
```

και τελειώνουμε.

Μια άλλη οικογένεια περιλαμβάνει ονόματα της μορφής `int_least`*N*`_t` και `uint_least`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Για παράδειγμα, ο τύπος `int_least32_t` είναι ο «μικρότερος« ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Μια τρίτη οικογένεια περιλαμβάνει ονόματα της μορφής `int_fast`*N*`_t` και `uint_fast`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Ο τύπος `int_fast32_t` είναι ο «ταχύτερος» (ό,τι και αν σημαίνει αυτό) ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Τέλος, ο τύπος `intmax_t` είναι αυτός στον οποίον μπορούμε να παραστήσουμε οποιαδήποτε ακέραιη τιμή μπορεί να παρασταθεί στη συγκεκριμένη υλοποίηση. Ο αντίστοιχος για τιμές χωρίς πρόσημο είναι ο `uintmax_t`. Ο gcc, στο `stdint.h`, έχει:

```
typedef long long          intmax_t;
typedef unsigned long long uintmax_t;
```

Όλοι αυτοί οι ορισμοί προβλέπεται να υπάρχουν στο νέο πρότυπο της C++.

17.3 Οι Ακέραιοι στο Πρόγραμμα

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
  int  i( INT_MAX );
  i = 4*i;
  cout << i << endl;
}
```

Αποτέλεσμα:

```
-4
```

Ας δούμε τί έγινε: Η (gcc - Dev) C++ παριστάνει τιμές τύπου `int` σε 32 δυαδικά ψηφία και το *INT_MAX* παριστάνεται ως εξής:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Το αποτέλεσμα του πολλαπλασιασμού υπολογίζεται σε:

```
3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0
2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 = 8589934588
```

Αλλά, στον τύπο `int` δεν μπορούμε να παραστήσουμε αυτήν την τιμή· έχουμε **υπερχείλιση** (overflow). Δουλεύοντας με αριθμητική υπολοίπων και συμπλήρωμα ως προς 2, η συνέχεια είναι η εξής: στη μεταβλητή *i* αποθηκεύονται τελικώς τα ψηφία 0 μέχρι 31 του αποτελέσματος. Όταν έρθει η ώρα να εκτελεσθεί η εντολή `cout << i << endl`, πρώτα ελέγχεται το ψηφίο προσήμου (31)· αφού έχει τιμή 1, ο υπολογιστής καταλαβαίνει ότι πρόκειται για *αρνητική τιμή!* Στη συνέχεια, αφαιρεί 1:

```
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
-                                                           1
---------------------------------------------------------------
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
```

και αλλάζει το 1 σε 0 και τα 0 σε 1:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

που είναι το 4.

Τι δίδαγμα βγαίνει από αυτά;

- ***Στις πράξεις του `int` (και των άλλων ακεραίων τύπων) είναι δυνατόν να έχουμε υπερχείλιση χωρίς καμιά ειδοποίηση από τον υπολογιστή. Αυτό είναι συχνά από τα δυσκολότερα λάθη, τουλάχιστον στην ανίχνευση.***

Πώς αντιμετωπίζεται αυτό το πρόβλημα; Ας δούμε τί μπορούμε να κάνουμε με την πρόσθεση. Έχουμε δυο μεταβλητές x, y, τύπου `int` και θέλουμε να υπολογίσουμε το $x + y$, αν υπολογίζεται. Για τα x, y έχουμε:

$$INT\_MIN \le x \le INT\_MAX$$
$$INT\_MIN \le y \le INT\_MAX$$

και θα αποπειραθούμε την πρόσθεση μόνον αν ξέρουμε από πριν ότι

$$INT\_MIN \le x + y \le INT\_MAX \text{ ή}$$
$$INT\_MIN - y \le x \le INT\_MAX - y$$

Φυσικά, δεν μπορούμε να γράψουμε:

```
if (INT_MIN-y <= x && (x <= INT_MAX-y)
  s = x + y;
else
  Λάθος
```

διότι οι πράξεις `INT_MIN - y` και `INT_MAX - y` *δεν είναι ασφαλείς!* Π.χ. αν η y έχει αρνητική τιμή, η `INT_MAX - y` μας δίνει σίγουρα υπερχείλιση. Ας τα ξαναδούμε πιο προσεκτικά. Κατ' αρχάς, αν οι x, y έχουν ετερόσημες τιμές δεν υπάρχει περίπτωση υπερχείλισης με την πρόσθεση. Αν οι x, y έχουν τιμές ≥ 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $x \le INT\_MAX - y$ και η πράξη δεξιά είναι ασφαλής. Αν οι x, y έχουν τιμές < 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $INT\_MIN - y \le x$ και η πράξη αριστερά είναι ασφαλής. Μπορούμε λοιπόν να γράψουμε την:

```
int addInt( int x, int y )
{
   int fv;

   if ( x >= 0 )
   {
      if ( y < 0 )  fv = x + y;
      else if ( x <= INT_MAX - y )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   else    // x < 0
   {
      if ( y >= 0 )  fv = x + y;
      else if ( INT_MIN - y <= x )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   return fv;
} // addInt
```

«Δηλαδή», σκέφτεσαι με φρίκη, «θα πρέπει αντί για "`c = a + b`" να γράφω "`c = addInt(a, b)`" και αντί για μια πράξη να καλώ μια συνάρτηση και να διαχειρίζομαι εξαιρέσεις;» Όχι! Σε ένα καλοσχεδιασμένο πρόγραμμα οι περισσότερες πράξεις *είναι* συνήθως ασφαλείς· και αυτές που δεν είναι φαίνονται εύκολα.[2] Φυσικά, δεν υπάρχει συνταγή για το πότε βάζουμε έλεγχο και πότε όχι· αλλά, δες δυο παραδείγματα:

```
#include <iostream>
#include <string>
#include <climits>
```

[2] Και ακόμη: για τις άλλες πράξεις τα πράγματα είναι πολύ πιο απλά.

```
using namespace std;

struct IntOvrflXptn
{
   char funcName[100];
   int  errCode;
   int  errVal1;
   int  errVal2;
   IntOvrflXptn( const char* fn, int ec, int ev1=0, int ev2= 0 )
   {
      strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal1 = ev1;  errVal2 = ev2;
   }
}; // IntOvrflXptn

int addInt( int x, int y );

int main()
{
   int x, y, z;

   cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
   cin >> x >> y;
   try
   {
      z = addInt( x, y );
      cout << z << endl;
   // . . .
   }
   catch ( IntOvrflXptn& xp )
   {
      cout << xp.errVal1 << " + " << xp.errVal2
           <<"??? ΣΟΒΑΡΕΨΟΥ!" << endl;
   }
// . . .
   do
   {
      cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
      cin >> x >> y;
   } while (x <= 0 || 100 < x || y <= 0 || 100 < y);
   z = x + y;
// . . .
} // main
```

Το μήνυμα που δίνεις πριν από την εντολή ανάγνωσης δεν σου εξασφαλίζει οτιδήποτε. Στην πρώτη περίπτωση δεν σε ενδιαφέρει να δεις αν ο χρήστης υπάκουσε στην οδηγία σου και προχωράς. Η χρήση της *addInt()* θα σε προφυλάξει από μια τιμή της *z* χωρίς νόημα. Στη δεύτερη περίπτωση, αφού δεν προχωράς παρά μόνο με «σωστές» τιμές των *x*, *y*, μπορείς να κάνεις την πρόσθεση χωρίς άλλον έλεγχο.

17.3.1 Παράμετροι "unsigned"

Στην §6.8 (και στην §13.3.1) δίναμε τον κανόνα:

- ***Μη βάζεις στις συναρτήσεις σου παραμέτρους τιμής τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

  Στο παράδειγμα παραβίασης του κανόνα που δίναμε καλούσαμε

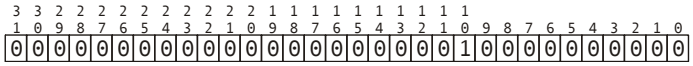
```
   n = -1024;
   cout << n << "  " << intSqrt(n) << endl;
```

τη συνάρτηση με επικεφαλίδα

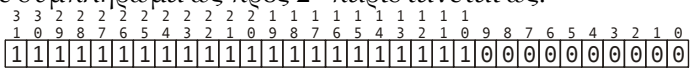
```
unsigned int intSqrt( unsigned int x );
```

Και τι βλέπαμε; Στη συνάρτηση περνούσε στη *x* η τιμή 4294966272 και η συνάρτηση υπολόγιζε την (ακέραιη) τετραγωνική της ρίζα.

Τώρα μπορείς να καταλάβεις τι γίνεται. Σε **int** τεσσάρων ψηφιολέξεων το 1024 (= 2^{10}) παριστάνεται ως:



Το "**-1024**" –σε συμπλήρωμα ως προς 2– παριστάνεται ως:



Αυτή είναι η εσωτερική παράσταση της *n* και αντιγράφεται ως τιμή της *x*. Μέσα στη συνάρτηση αυτή η παράσταση ερμηνεύεται ως **unsigned int**. Έτσι προκύπτει η τιμή 4294966272.

Άσκηση για σένα: η επιβεβαίωση της παράστασης του "**-1024**" και ο υπολογισμός της τιμής 4294966272.

17.4 * Απαριθμητοί Τύποι (ξανά)

Πρωτοείδαμε τους απαριθμητούς τύπους πολύ νωρίς (§4.8) και τους χρησιμοποιούμε όπως θα χρησιμοποιούσαμε τους αντίστοιχους της Pascal –και πολύ καλά κάναμε. Στη συνέχεια, σε ορισμένες περιπτώσεις, θα πρέπει να τους χρησιμοποιήσουμε όπως τους χρησιμοποιεί η C.

Ας ξαναδούμε τον τύπο

```
enum Digit { miden = 48, zero = 48, one, two, three, four,
             five, six, seven, eight, nine };
```

και τη δήλωση:

```
   Digit d1( 49 );
```

Ο μεταγλωττιστής θα την απορρίψει: «invalid conversion from '**int**' to '*Digit*'» (gcc –Dev C++).[3] Αν όμως δώσεις:

```
   Digit d1( static_cast<Digit>(49) );
```

όλα πάνε μια χαρά. Σωστό!

Δοκιμάζουμε όμως και το εξής:

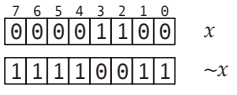
```
   d1 = static_cast<Digit>( 9 );
```

Και αυτό περνάει χωρίς το παραμικρό πρόβλημα, ενώ, με βάση αυτά που ξέρουμε, θα έπρεπε να γίνονται δεκτές τιμές από **static_cast<Digit>(49)** μέχρι και **static_cast <Digit>(57)**.

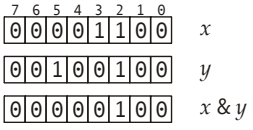
Αυτό που συμβαίνει στην πραγματικότητα είναι το εξής:

- Αν οι σταθερές που έχουμε στην απαρίθμηση είναι μη αρνητικές η C++ θα δεχτεί ως τιμή μιας σταθεράς κάθε ακέραιη τιμή από 0 μέχρι τη μέγιστη τιμή που μπορεί να παρασταθεί στα δυαδικά ψηφία –χωρίς να υπολογίζουμε ψηφίο προσήμου– που απαιτούνται για τη μέγιστη τιμή της απαρίθμησης. Στο παράδειγμά μας μέγιστη τιμή της απαρίθμησης είναι η *nine* που αντιστοιχεί στο $57_{10} = 111001_2$. Η μέγιστη τιμή που μπορεί να παρασταθεί σε έξη δυαδικά ψηφία είναι $111111_2 = 63_{10}$.
- Αν υπάρχουν και αρνητικές τιμές τότε ισχύουν τα ίδια αλλά θα πρέπει να υπολογίζουμε και ψηφίο προσήμου. Για παράδειγμα, αν στην απαρίθμηση του παραδείγματος βάλουμε άλλο ένα στοιχείο **neg = -1**, τότε η περιοχή είναι από -64 μέχρι 63 (παράσταση σε 7 ψηφία). Αν βάλουμε **neg = -100**, τότε η περιοχή είναι από -128 μέχρι 127 (παράσταση σε 8 ψηφία).

[3] Ο δικός σου μεταγλωττιστής, με ή χωρίς διμαρτυρίες (warnings), τη δέχτηκε! Συμβαίνουν και αυτά...



Σχ. 17-1 Η ψηφιοπράξη "**~**" (συμπλήρωμα ως προς 1).



Σχ. 17-2 Ψηφιοπράξη "**&**" (**and**).

Πάντως είναι πολύ πιθανό, ο μεταγλωττιστής σου να δεχτεί χωρίς διαμαρτύρίές ως τιμή μεταβλητής τύπου *Digit* τη `static_cast<Digit>(n)` όπου *n* τυχούσα τιμή τύπου `int`.

17.5 Ψηφιοπράξεις στη C++

Η C++ προσφέρει ορισμένες πράξεις που επιτρέπουν διαχείριση των τιμών όλων των ακέραιων τύπων της (δυαδικό) ψηφίο προς ψηφίο· για τον λόγο αυτόν τις ονομάζουμε **ψηφιοπράξεις** (bitwise operations).

Ας τις δούμε ξεκινώντας από τις πράξεις **ολίσθησης** (shift). Ας πούμε ότι έχουμε:

```
   unsigned char  x, y;
// . . .
   x = 77;
```

Στη μνήμη θα αποθηκευτούν σε μια ψηφιολέξη τα εξής:

7 6 5 4 3 2 1 0
0 1 0 0 1 1 0 1

Αν δώσουμε την εντολή `y = x << 2` (ολίσθησε αριστερά κατά 2 δυαδικά ψηφία), θα συμβούν τα εξής:

- οι τιμές όλων των ψηφίων της *x* θα ολισθήσουν κατά 2 θέσεις προς τα αριστερά,
- τα 2 πρώτα (από αριστερά, 7 και 6) θα χαθούν,
- τα δυο τελευταία θα γίνουν 0.
- Ο τύπος του αποτελέσματος είναι ίδιος με τον τύπο του πρώτου ορίσματος.

Έτσι, στη θέση *y* θα αποθηκευτούν τα εξής:

7 6 5 4 3 2 1 0
0 0 1 1 0 1 0 0

Σχεδόν παρομοίως γίνεται και η ολίσθηση δεξιά. Αν δώσουμε την εντολή `y = x >> 2` (ολίσθησε δεξιά κατά 2 δυαδικά ψηφία), θα πάρουμε στη *y*:

7 6 5 4 3 2 1 0
0 0 0 1 0 0 1 1

Το «σχεδόν» τι αφορά; Το «γέμισμα» με μηδενικά:

- *Αν ο τύπος της x είναι* `unsigned` *τότε οι πρώτες θέσεις που «αδειάζουν» θα «γεμίσουν» με μηδενικά.*
- *Αν ο τύπος της x δεν είναι* `unsigned` *και το ψηφίο προσήμου είναι* "1" *το πώς θα γεμίσουν οι θέσεις που αδειάζουν μπορεί να αλλάζει από τον έναν μεταγλωττιστή στον άλλον.*

Ορίζονται και οι σχετικές συντομογραφίες για την εκχώρηση:

- Αντί για "`x = x << N`" μπορείς να γράφεις "`x <<= N`" και
- αντί για "`x = x >> N`" μπορείς να γράφεις "`x >>= N`".

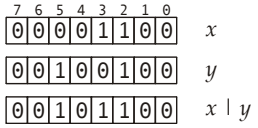
Στα παραδείγματα αυτά με την ολίσθηση χάνονται "`1`". Για να δούμε τι συμβαίνει αν δεν έχουμε τέτοιες απώλειες. Ας πούμε ότι στο *x* έχουμε βάλει την τιμή 12, οπότε, όπως είπαμε, η εσωτερική παράσταση είναι: 7 6 5 4 3 2 1 0 **0 0 0 0 1 1 0 0**.

Μετά τη "`y = x << 2`" η *y* γίνεται: 7 6 5 4 3 2 1 0 **0 0 1 1 0 0 0 0** .που είναι $2^5+2^4 = 48 = 12\times4$, ενώ μετά τη "`y = x >> 2`" η *y* γίνεται: 7 6 5 4 3 2 1 0 **0 0 0 0 0 0 1 1** .που είναι $2^1+2^0 = 3 = 12/4$,

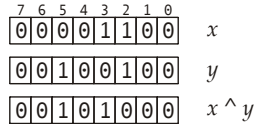
Δηλαδή, μπορούμε να πούμε:

`x << N` σημαίνει $x \times 2^N$ και `x >> N` σημαίνει x / 2^N ;

Ναι,



Σχ. 17-3 Ψηφιοπράξη "`|`" (**or**).



Σχ. 17-4 Ψηφιοπράξη "`^`" (**xor**).

- αν δεν έχουμε υπερχείλιση (για τη "`<<`") και
- αν δεν έχουμε περιπλοκές από τα πρόσημα (`char`, `short`, `int`, `long`).

Αυτές οι δυο πράξεις μας δίνουν τη δυνατότητα να γράφουμε από την C++ δυο εντολές του επεξεργαστή. Είναι λοιπόν πολύ ταχύτερες από τις αντίστοιχες αριθμητικές, αλλά *μην αρχίσεις να σκέφτεσαι να κάνεις έτσι πολλαπλασιασμούς και διαιρέσεις ακεραίων*: χρειάζονται προσοχή στη χρήση και οι πιθανότητες για λάθη είναι πάρα πολλές.

Παρατήρηση: ▶

Ο προσεκτικός αναγνώστης, διαβάζοντας για τους τελεστές ολίσθησης, "`<<`" και "`>>`", θα πρέπει να ανησύχησε: πώς δεν γίνεται μπέρδεμα με τους ίδιους (οπτικώς) τελεστές που χρησιμοποιούμε για γράψιμο και διάβασμα τιμών; Οι τελεστές ολίσθησης περιμένουν δύο ορίσματα που είναι ακέραιοι αριθμοί, ενώ οι τελεστές εισόδου/εξόδου χρειάζονται αριστερά κάποιο ρεύμα. Βεβαίως, σε ορισμένες περιπτώσεις χρειάζεται λίγη προσοχή. Αν γράψεις:

```
int x = 12;
cout << x << 2 << endl;
cout << (x << 2) << endl;
```

θα πάρεις αποτέλεσμα:

```
122
48
```

Το 122 προέρχεται από την πρώτη εντολή εκτύπωσης, που λέει: γράψε την τιμή της x, που είναι 12 και στη συνέχεια γράψε και το 2· έτσι παίρνουμε αυτό το "122". Η δεύτερη εντολή λέει: τύπωσε το αποτέλεσμα της πράξης "`x << 2`", που, όπως είδαμε, είναι 48.◀

Η C++ μας επιτρέπει ακόμη τις πράξεις "`~`", "`&`", "`|`" και "`^`" μεταξύ τιμών ακέραιων τύπων (αντίστοιχες των "`!`", "`&&`", "`||`" και "`!=`" μεταξύ λογικών τιμών). Αν οι x, y είναι τύπου `unsigned char`:

- Η "`~`x" (Σχ. 17-1) προκύπτει από τη x με αλλαγή κάθε ψηφίου "`0`" σε "`1`" και κάθε ψηφίου "`1`" σε "`0`" (συμπλήρωμα ως προς 1).
- Η "x `&` y" προκύπτει από τις x και y με **and** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-2 βλέπεις ότι το μοναδικό "`1`" που προκύπτει είναι στο ψηφίο 2 διότι στη θέση αυτήν έχουν "`1`" και η x και η y.
- Η "x `|` y" προκύπτει από τις x και y με **or** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-3 βλέπεις ότι η x `|` y έχει "`1`" στις θέσεις
  - 2 που έχουν "`1`" και η x και η y,
  - 3 όπου έχει "`1`" η x και
  - 5 όπου έχει "`1`" η y.

  Αν δεν έχουμε πρόσημα και "`1`" στην ίδια θέση τότε το x `|` y είναι το ίδιο με το x `+` y. Π.χ. αν στη y είχαμε "`00100010`" (παράσταση του 34_{10}) τότε το x `|` y είναι "`00101110`" που είναι παράσταση του $46_{10} = 12 + 34$.
- Η "x `^` y" προκύπτει από τις x και y με **xor** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-4 βλέπεις ότι η x `^` y έχει "`1`" στις θέσεις
  - 3 όπου έχει "`1`" μόνον η x και
  - 5 όπου έχει "`1`" μόνον η y.

  Για τους τρεις διμελείς ορίζονται συντομογραφίες εκχώρησης:
- Αντί για "`x = x & y`" μπορείς να γράφεις "`x &= y`",

- αντί για "`x = x | y`" μπορείς να γράφεις "`x |= y`" και
- αντί για "`x = x ^ y`" μπορείς να γράφεις "`x ^= y`".

Εδώ όμως χρειάζεται και πάλι προσοχή: Ας πούμε ότι έχεις:

```
short int m( 3 );
char c( -5 );
```

και θέλεις να υπολογίσεις το: `m |= c`. Η *c* έχει εσωτερική παράσταση: "`111110 11`". Για να γίνει η ψηφιοπράξη "`|`" θα πρέπει η *m* και η *c* να έχουν εσωτερική παράσταση με το ίδιο πλήθος ψηφίων. Η *m* «προωθείται» σε `short int` με εσωτερική παράσταση

"`1111111111111011`" (= -5 σε `short int`)!!!

Μάλλον δεν είναι αυτό που θέλεις. Αν είχες δηλώσει:

```
unsigned char c( 251 );
```

η *c* θα είχε την ίδια εσωτερική παράσταση (11111011) και με την προώθηση θα γίνονταν "`0000000011111011`".

Μετά από όσα είπαμε μπορείς να καταλάβεις τη σύσταση του (CERT 2009):[4]

♦ ***Χρησιμοποίησε τους τελεστές ψηφιοπράξεων μόνο με ορίσματα τύπων*** `unsigned`.[5]

Ας δούμε τώρα ένα παράδειγμα χρήσης των "`<<`" και "`&`".

Παράδειγμα

Θέλουμε να γράψουμε μια:

```
int bitValue( unsigned char b, int pos )
```

που θα μας επιστρέφει την τιμή του ψηφίου στη θέση *pos* της ψηφιολέξης *b*.

Πώς θα σκεφτούμε; Ας πούμε ότι έχουμε δηλώσει: `unsigned char t` και η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`1`". Πώς θα είναι τα ψηφία της `b & t`; Αφού η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* και αφού ξέρουμε ότι *P* `&& false` ≡ `false`, το ίδιο θα ισχύει και για την `b & t`: θα έχει "`0`" σε όλες τις θέσεις εκτός από την *pos*. Στη θέση *pos*, όπου η *t* έχει "`1`":

- Αν η *b* έχει "`0`" τότε και η `b & t` θα πάρει "`0`" και θα έχει τιμή 0 (μηδέν) αφού θα έχει παντού μηδενικά.
- Αν η *b* έχει "`1`" τότε και η `b & t` θα πάρει "`1`" και θα έχει τιμή ≠ 0 αφού θα έχει ένα μη μηδενικό ψηφίο.

Και πώς θα δώσουμε στην *t* την τιμή που θέλουμε; Έτσι:

```
unsigned char t( 1 );
t = t << pos;
```

δηλαδή:

- Δίνοντας στην *t* τιμή 1 βάζουμε 1 στο ψηφίο 0 της *t* και 0 σε όλα τα άλλα και
- με την "`t = t << pos`" μεταφέρουμε το 1, κατά *pos* θέσεις προς τα αριστερά, ενώ σε όλες τις άλλες θέσεις υπάρχουν 0.

Πριν γράψουμε τη συνάρτησή να παρατηρήσουμε ότι ενώ ορίζεται για όλες τις τιμές της *b*, δεν ορίζεται για τιμές της *pos* < 0 ή > 7· δηλαδή δεν είναι ολική.

Να λοιπόν πώς θα είναι η

```
int bitValue( unsigned char b, int pos )
{
   if ( pos < 0 || 7 < pos )   throw pos;

   unsigned char t( 1 );
   t <<= pos;
```

---

[4] Σύσταση *INT*13: "Use bitwise operators only on unsigned operands."

[5] Αν μελετάς σωστά αυτό το κεφάλαιο, θα έχεις ήδη παραβιάσει τη συσταση και θα την παραβιάσεις αρκετές φορές ακόμη για να δεις εσωτερικές παραστάσεις αρνητικών αριθμών. Δεν πειράζει, αφού είναι για εκπαιδευτικούς λόγους!

```
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Αν θέλεις να δεις την εσωτερική παράσταση του (`unsigned char`) "`12`" δηλώνεις

```
   unsigned char x( 12 );
```

ζητάς:

```
   for ( int pos(7); pos >= 0; --pos )
      cout << bitValue( x, pos );
   cout << endl;
```

και παίρνεις:

```
00001100
```

Με χρήση της *bitValue()* μπορούμε να γράψουμε τη:

```
void display( ostream& tout, unsigned char b )
{
   for ( int pos(7); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

που σου είναι χρήσιμη αν μελετάς σοβαρά αυτό το κεφάλαιο.

Το παρακάτω πρόγραμμα μας δίνει τις εσωτερικές παραστάσεις των ακεραίων 12 και 20 χρησιμοποιώντας την *display()*:

```
#include <iostream>
using namespace std;

int bitValue( unsigned char b, int pos );
void display( ostream& tout, unsigned char b );

int main()
{
   unsigned char  x;
   int   k;

   try
   {
      x = 12;
      display( cout, x );  cout << endl;
      x = 20;
      display( cout, x );  cout << endl;
   }
   catch ( int& p )
   {
      cout << " η bitValue κλήθηκε με δεύτερο όρισμα "
           << p << endl;
   }
} // main
```

17.6 Ψηφιοχάρτες και Συνηθισμένες Πράξεις

Παρ' όλο που στα παραδείγματά μας, μέχρι τώρα, το τελικό μας ενδιαφέρον φαίνεται να βρίσκεται στην τιμή της ψηφιολέξης ή, γενικώς, της συνολικής παράστασης, η διαχείριση δυαδικών ψηφίων είναι ενδιαφέρουσα καθ' εαυτή. Με τη διαχείριση δυαδικών ψηφίων, εκτός άλλων εφαρμογών, μπορούμε να υλοποιήσουμε σύνολα, στα οποία μας ενδιαφέρει μόνον η πληροφορία *ανήκει* ("1") ή *δεν ανήκει* ("0"). Σε τέτοιες περιπτώσεις χρησιμοποιούμε πίνακες ακέραιων τιμών, π.χ. τύπου `unsigned long`, που όμως τις βλέπουμε ως **ψηφιοσύνολα** (bitsets) ή **ψηφιοχάρτες** (bitmaps)[6].

[6] Ο όρος *bitmap* χρησιμοποιείται και σε έναν τρόπο παράστασης γραφικών (bitmap graphics).

Στη συνέχεια δίνουμε μερικές συναρτήσεις –για την ακρίβεια: περιγράμματα συναρτησεων– για μερικές πολύ συνηθισμένες περιπτώσεις διαχείρισης ψηφιοπινάκων. Μπορεί να κληθούν με τύπο πρώτης παραμέτρου (*T*) κάποιον από τους `int`, `unsigned int`, `long`, `unsigned long`, `short`, `unsigned short`, `char`, `unsigned char`, `wchar_t`, `long long`, `unsigned long long`.

Μερικές από αυτές ρίχνουν εξαίρεση τύπου:

```
struct BitmapXptn
{
   enum { outOfRange, paramErr };
   char funcName[100];
   int  errorCode;
   int  errVal1, errVal2;
   BitmapXptn( const char* mn, int ec, int v1, int v2 = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = v1;  errVal2 = v2;  }
}; // BitmapXptn
```

Προσοχή! Στα σχόλια τεκμηρίωσης (και μόνο) των περιγραμμάτων που δίνουμε στη συνέχεια θα χρησιμοποιούμε τον συμβολισμό `b[pos]` για το δυαδικό ψηφίο της *b* στη θέση *pos*. *Ο συμβολισμός αυτός δεν είναι δεκτός από τη C++ για τέτοια χρήση.*

17.6.1 Τιμή Δυαδικού Ψηφίου

Ξεκινούμε μετατρέποντας σε περίγραμμα τη *bitValue* που γράψαμε πιο πριν:

```
template < typename T >  int bitValue( T b, int pos )
```

Η βασική διαφορά είναι ότι τώρα το τελευταίο δυαδικό ψηφίο δεν βρίσκεται στη θέση 7, αλλά στη θέση

$$8\texttt{(sizeof}\ b\texttt{)} - 1$$

Έτσι έχουμε:

```
// bitValue -- επιστρέφει τη b[pos]
//             Προϋπόθεση: 0 <= pos < 8*(sizeof b)
template < typename T >
int bitValue( T b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) <= pos )
      throw BitmapXptn( "bitValue", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Μετατρέπουμε σε περίγραμμα και τη *display()* για να μπορείς να κάνεις τις δοκιμές σου:[7]

```
template < typename T >
void display( ostream& tout, T b )
{
   int lastb( 8*(sizeof b) - 1 );

   for ( int pos(lastb); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

[7] Κανονικώς θα πρέπει να ελέγχουμε αν είναι ανοικτό το ρεύμα, αν το γράψιμο έγινε επιτυχώς και να ρίχνουμε τις αντίστοιχες εξαιρέσεις.

17.6.2 Βάλε Τιμή 1 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void setBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "`1`" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Φυσικά θα πρέπει να υπάρχει «*δυαδικό ψηφίο της θέσης pos*», δηλαδή:

$$0 \texttt{ <= } pos \texttt{ < } 8(\texttt{sizeof } b)$$

Πώς θα πετύχουμε το στόχο μας; Όπως μάθαμε, μετά τις

```
   T t( 1 );
   t <<= pos;
```

η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`1`". Η τιμή της παράστασης `b | t` θα έχει:

- "`1`" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "`1`" και ξέρουμε ότι *P* `|| true` ≡ `true`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "`0`" και ξέρουμε ότι *P* `|| false` ≡ *P*.

Να λοιπόν το περίγραμμα της συνάρτησης:

```
// setBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//           "1" στη θέση pos
//           Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//           Απαίτηση:
//           βτελ[pos] == 1 && για κάθε k!=pos: βτελ[k]==βαρχ[k]
template < typename T >
void setBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "setBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   b |= t;
} // setBit
```

Ας δούμε δύο παραδείγματα χρήσης:

Παράδειγμα 1

Σε μια μεταβλητή *si* τύπου `unsigned short int` (16 δυαδικά ψηφία) θέλουμε να έχουμε όλα τα ψηφία "`0`" εκτός από αυτά που βρίσκονται στις θέσεις 10, 3 και 9:

```
      si = 0;
      setBit( si, 10);  setBit( si, 3);  setBit( si, 9);
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
0000011000001000
```

Παράδειγμα 2

Σε μια μεταβλητή *si* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή μιας άλλης μεταβλητής *uc* τύπου `unsigned char`. Θέλουμε ακόμη, το ψηφίο 13 της *si* να έχει τιμή "`1`":

```
      display( cout, uc );  cout << endl;
      si = uc;
      setBit( si, 13 );
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
11111011
0010000011111011
```

🖈🖈🖈

Παρατήρηση: ▶

Ένα εύλογο ερώτημα που μπορεί να έχουν πολλοί είναι το εξής: Γιατί να μην ελέγξουμε αν το συγκεκριμένο ψηφίο είναι ήδη "`1`"; Στην περίπτωση αυτή δεν χρειάζεται να κάνουμε οτιδήποτε. Ας το δούμε· ο έλεγχος θα γίνει όπως είδαμε στη *bitValue()*:

```
   T t( 1 );
   t <<= pos;
   if ( (b & t) == 0 )  b |= t;
```

Δηλαδή:

- Σε κάθε περίπτωση έχουμε υπολογισμό της "`b & t`" και της **if** και
- Όταν το ψηφίο δεν έχει τιμή "`1`" εκτέλεση και της "`b |= t`".

Προφανώς ο τρόπος που επιλέξαμε –υπολογισμός μόνον της "`b |= t`"– είναι σαφώς πιο συμφέρων.◀

17.6.3 Βάλε Τιμή 0 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void clearBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Η λύση στο πρόβλημά μας μπορεί να προκύψει από τη λύση στο προηγούμενο πρόβλημα αν εναλλάξουμε τα "`0`" και "`1`" καθώς και τα **and** και **or**: Πράγματι, ας πούμε ότι η *t* έχει "`1`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`0`". Η τιμή της παράστασης `b & t` θα έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "`0`" και ξέρουμε ότι *P* `&& false` ≡ `false`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "`1`" και ξέρουμε ότι *P* `&& true` ≡ *P*.

Και πώς κάνουμε την t να «*έχει "`1`" σε όλες τις θέσεις εκτός από την pos όπου έχει "`0`"*»; Αυτό είναι απλό:

```
   T t( 1 );
   t <<= pos;  t = ~t;
```

Να λοιπόν η

```
// clearBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//             "0" στη θέση pos
//             Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//             Απαίτηση:
//             βτελ[pos] == 0 && για κάθε k!=pos: βτελ[k]==βαρχ[k]
template < typename T >
void clearBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "clearBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;  t = ~t;
   b &= t;
} // clearBit
```

Παράδειγμα

Σε μια μεταβλητή *m* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή της *si* του Παραδ. 2, της προηγούμενης παραγράφου, με τη διαφορά ότι η *m* θα έχει "`0`" σε όλες τις άρτιες θέσεις:

```
      m = si;
      for ( int k(0); k < 8*sizeof(m); k += 2 )
         clearBit( m, k );
      display( cout, m );  cout << endl;
```

Αποτέλεσμα:

```
0010000010101010
```

17.6.4 Πλήθος "1"

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  size_t count1( T b )
```

που θα επιστρέφει ως τιμή το πλήθος των δυαδικών ψηφίων της που έχουν τιμή "`1`" σε τιμή *b* τύπου *T*.

Θα μπορούσαμε να καλέσουμε 8(`sizeof` *b*) φορές. Αντί για αυτό εξετάζουμε τόσες φορές την τιμή της `b & t` όπου η *t* –τύπου *T*– έχει μόνο ένα "`1`". Κάθε φορά το "`1`" μετατοπίζεται ώστε να περάσει από όλες τις θέσεις της *t*.

```
// count1 -- επιστρέφει το πλήθος των ψηφίων της b με τιμή 1
//           Προϋπόθεση: true
//           Απαίτηση: fv == πληθος ψηφίων της b με τιμή 1
template < typename T >
size_t count1( T b )
{
   const size_t lastb( 8*(sizeof b) - 1 );
   T t( 1 );
   size_t fv( 0 );

   for ( int k(0); k <= lastb; ++k )
   {
      if ( (b & t) != 0 )  ++fv;
      t <<= 1;
   }
   return fv;
} // count1
```

Αν η *b* υλοποιεί κάποιο σύνολο ("`1`": «ανήκει»), η *count*1 μας δίνει τον *πληθάριθμο* του συνόλου.

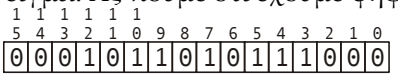
17.6.5 Μέρος Ψηφιοχάρτη

Θέλουμε ένα περίγραμμα συνάρτησης:

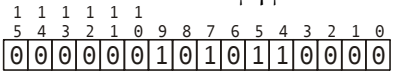
```
template < typename T >  T part( T b, int pos1, int pos2 )
```

που θα επιστρέφει ως τιμή τύπου *T* τα ψηφία της b από ***pos*1** μέχρι και ***pos*2**.

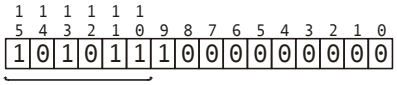
Να το δούμε με ένα παράδειγμα: Ας πούμε ότι έχουμε ψηφιοχάρτη 16 ψηφίων:



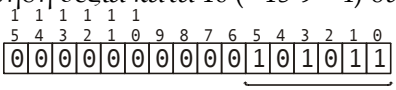
Θέλουμε έναν ψηφιοχάρτη με το ίδιο μέγεθος που θα έχει το υπογραμμισμένο κομμάτι δηλαδή τα ψηφία από 4 μέχρι 9 και όλα τα άλλα ψηφία 0:



Πρέπει δηλαδή να μηδενίσουμε τα ψηφία από 0 μέχρι 3 και από 10 μέχρι 15. Αυτό μπορεί να γίνει με την *clearBit*· μπορεί όμως να γίνει και ταχύτερα με τις πράξεις ολίσθησης. Αν στον αρχικό ψηφιοχάρτη κάνουμε μια ολίσθηση αριστερά και 6 (=15-9) θέσεις θα πάρουμε:



Αν σε αυτό κάνουμε ολίσθηση δεξιά κατά 10 (= 15-9 + 4) θα πάρουμε:



Τέλος, με ολίσθηση αριστερά κατά 4 θέσεις παίρνουμε αυτό που θέλουμε.

Αν πάρουμε υπόψη μας ότι: αντί για 15 έχουμε (γενικώς) το 8(`sizeof` *b*)–1, 4 είναι το *pos*1 και 9 είναι το *pos*2 έχουμε:

```
// part -- επιστρέφει το μέρος της b
//         με τα ψηφία της από pos1 μέχρι και pos2
//         Προϋπόθεση: 0 <= pos1 <= pos2 < 8*(sizeof v)
//         Απαίτηση: για κάθε k: 0..pos1-1 βτελ[k]==0 &&
//                   για κάθε k: pos1..pos2 βτελ[k]==βαρχ[k] &&
//                   για κάθε k: pos2+1..8*(sizeof v)-1 βτελ[k]==0
template < typename T >
T part( T b, int pos1, int pos2 )
{
   const size_t lastb( 8*(sizeof b) - 1 );

   if ( pos2 < pos1 )
      throw BitmapXptn( "part", BitmapXptn::paramErr, pos1, pos2 );
   if ( pos1 < 0 || pos2 < 0 || lastb < pos1 || lastb < pos2 )
      throw BitmapXptn( "part", BitmapXptn::outOfRange, pos1, pos2 );
// 0 <= pos1 <= pos2 <= lastb
   b <<= (lastb - pos2);
   b >>= (lastb - pos2 + pos1);
   b <<= pos1;
   return b;
} // part
```

Παράδειγμα

Αν η *si* είναι τύπου `unsigned short int` οι

```
      display( cout, si );  cout << endl;
      setBit( si, 12);
      unsigned short int sip( part(si, 4, 9) );
      display( cout, sip );  cout << endl;
```

δίνουν:

```
0000011010111000
0000001010110000
```

Και δύο λόγια για την πρώτη εξαίρεση: Για ορισμένες εφαρμογές το *pos*2 < *pos*1 δεν είναι και τόσο παράνομο. Απλώς στην περίπτωση αυτήν η συνάρτηση θα πρέπει να επιστρέφει *T*(0). Διαλέγεις και παίρνεις.

17.7 Τύποι *bitmask*

Πρωτοείδαμε τον τελεστή "`|`" στο Κεφ. 8, όταν συζητούσαμε για άνοιγμα ρεύματος. Ας πούμε ότι είχαμε να γράψουμε εμείς μια συνάρτηση που να ανοίγει ένα ρεύμα προς/από αρχείο. Γυρίζουμε λοιπόν στην §8.12 για να θυμηθούμε τα συστατικά του τρόπου ανοίγματος και καταλαβαίνουμε ότι (αν δεν θέλουμε να βάλουμε 6 ξεχωριστές παραμέτρους) θα πρέπει να βάλουμε μια παράμετρο τύπου:

```
struct OpenFlags
```

```
{
   bool app;
   bool ate;
   bool binary;
   bool in;
   bool out;
   bool trunc;
}; // OpenFlags
```

Επειδή μια τέτοια παράμετρος πιάνει 6 ψηφιολέξεις μπορούμε να κάνουμε οικονομία χρησιμοποιώντας ψηφιοπεδία:

```
struct OpenFlagsBF
{
   bool app:    1;
   bool ate:    1;
   bool binary: 1;
   bool in:     1;
   bool out:    1;
   bool trunc:  1;
}; // OpenFlagsBF
```

Μια τιμή τύπου *OpenFlagsBF* δεν χρειάζεται πάνω από μια ψηφιολέξη.

Σε αυτό το κεφάλαιο μάθαμε ότι μπορούμε να χειριστούμε το κάθε δυαδικό ψηφίο μιας τιμής ακέραιου τύπου. Και αφού οι τιμές που μπορεί να παίρνει είναι "`0`" ή "`1`" μπορούμε να το χειριστούμε ως τιμή τύπου `bool`. Έτσι, αντί για μεταβλητή τύπου *OpenFlagsBF* μας αρκεί μια μεταβλητή τύπου `unsigned char` (τη «μικρότερη» με 6 δυαδικά ψηφία). Πράγματι, ας πούμε ότι σε μια τέτοια τιμή ορίζουμε ότι το δυαδικό ψηφίο 0 ως σημαία για το "`app`", το ψηφίο 1 για το "`ate`", ..., το ψηφίο 5 για το "`trunc`". Για να αποφύγουμε λάθη δηλώνουμε τις σταθερές:

```
const unsigned char appPos = 0, atePos = 1, binaryPos = 2,
                    inPos = 3, outPos = 4, truncPos = 5;
```

Έτσι, για να πούμε ότι θέλουμε να ανοίξουμε ένα ρεύμα *in*, *out* και *binary* σε μια μεταβλητή:

```
   unsigned char om;
```

βάζουμε:

```
   om = 0;
   setBit( om, inPos );  setBit( om, outPos );
   setBit( om, binaryPos );
```

Ελέγχουμε αν, ας πούμε, το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `bitValue(om, binaryPos) == 1`.

Μπορούμε να τα καταφέρουμε χωρίς τη *setBit*; Ναι! Δες μια άλλη, διαφορετική, ομάδα σταθερών:

```
const unsigned char app = 1, ate = (app << 1),
                    binary = (app << 2), in = (app << 3),
                    out = (app << 4), trunc = (app << 5);
```

Αυτές δεν κρατούν τη θέση του ψηφίου που αντιστοιχεί στην κάθε σημαία *αλλά σε εκείνη ακριβώς τη θέση έχουν το μοναδικό* "`1`".[8] Με αυτές δίνουμε στη *om* την τιμή που θέλουμε έτσι:

```
   om = in | out | binary;
```

Αφού οι σταθερές έχουν τα "`1`" σε διαφορετικές θέσεις θα μπορούσαμε να γράψουμε και:

```
   om = in + out + binary;
```

[8] Θα μπορούσαμε να είχαμε γράψει:

```
const unsigned char app = 1, ate = (1 << 1), binary = (1 << 2),
                    in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

αλλά αυτό δεν είναι και πολύ καλή ιδέα!

Πάντως μπορούμε να κάνουμε και αυτό που κάνουμε στη *setBit*:

```
om = 0;
om |= in;  om |= out;  om |= binary;
```

Αν δεν έχουμε τη θέση πώς θα ελέγχουμε αν κάποιο ψηφίο έχει τιμή 1. Για να δούμε, ας πούμε, αν το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `(om & binary) != 0`.

Τα παραπάνω είναι ένας τρόπος υλοποίησης ενός τύπου bitmask.

Ένας **τύπος bitmask** έχει τα εξής χαρακτηριστικά:

- N+1 σταθερές c_0 = `1`, c_1 = `(1 << 1)`, …, c_N`(1 << N)`.
- Αν v_1, v_2 τιμές του τύπου τότε και οι v_1 `&` v_2, v_1 `|` v_2, v_1 `^` v_2, `~`v_1 είναι τιμές του τύπου.
- Εκτός από τις ψηφιοπράξεις "`&`", "`|`", "`^`", "`~`" ορίζονται και οι «συντομογραφίες» εκχώρησης "`&=`", "`|=`", "`^=`".

Μπορούμε να υλοποιήσουμε έναν τέτοιον τύπο με τρεις τρόπους.

1. Ο πρώτος είναι *με κατάλληλο ακέραιο τύπο*, δηλαδή τύπο που οι τιμές του να παριστάνονται σε N+1 δυαδικά ψηφία τουλάχιστον. Παραπάνω, είδαμε παράδειγμα τέτοιας υλοποίησης. Αφού στην περίπτωσή μας N = 5 οποιοσδήποτε ακέραιος τύπος είναι κατάλληλος. Επιλέγουμε τον «μικρότερο»:

```
typedef unsigned char OpenMode;
const OpenMode app = 1, ate = (1 << 1), binary = (1 << 2),
               in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

   Όλες οι ψηφιοπράξεις που μας ενδιαφέρουν είναι ήδη ορισμένες.

2. Ο δεύτερος τρόπος υλοποίησης είναι με κάποιον *απαριθμητό τύπο*. Για το παράδειγμά μας ορίζουμε:

```
enum OpenMode
{  app = 1, ate = (1 << 1), binary = (1 << 2), in = (1 << 3),
   out = (1 << 4), trunc = (1 << 5)  };
```

   Αν δεν διάβασες την §17.4 να πούμε ότι θα πρέπει να ξεχάσεις αυτά που μάθαμε για τους απαριθμητούς τύπους: σε μια μεταβλητή τύπου *OpenMode* μπορούμε (με ή χωρίς διαμαρτυρίες από τον μεταγλωττιστή) να βάλουμε τιμές που δεν υπάρχουν στην παραπάνω απαρίθμηση.

   Αν και ορισμένοι μεταγλωττιστές θα δεχτούν –με διαμαρτυρίες (warnings)– να κάνουν ψηφιοπράξεις με αυτές τις σταθερές, το σωστό είναι να επιφορτώσεις τους τελεστές που είδαμε παραπάνω. Δίνουμε για το παράδειγμά μας την επιφόρτωση των "`&`" και "`&=`":

```
OpenMode operator&( OpenMode x, OpenMode y )
{
   return static_cast<OpenMode>( static_cast<unsigned char>(x) &
                                 static_cast<unsigned char>(y) );
} // operator&( OpenMode

OpenMode& operator&=( OpenMode& x , OpenMode y )
{
   x = x & y;
   return x;
} // operator&=( OpenMode
```

   Όπως βλέπεις, ενώ με τη χρήση της απαρίθμησης δεν χρειάστηκε να επιλέξουμε ακέραιο τύπο, χρειάζεται τώρα για την επιφόρτωση των τελεστών.

3. Ο τρίτος τρόπος υλοποίησης είναι *με χρήση του περιγράμματος bitset* της STL. Θα τον δούμε αργότερα.

Οι τύποι bitmask χρησιμοποιούνται πολύ συχνά σε προγράμματα και βιβλιοθήκες C++ (και C).

17.8 Αριθμητικές Πράξεις και Ψηφιοπράξεις

Σε υποσημείωση της §8.6 λέγαμε για την "`ios_base::in|ios_base::out`" «*αυτό μπορεί να το δεις και ως:* "`ios_base::in+ios_base::out`".» Στην προηγούμενη παράγραφο είδαμε γιατί μπορεί να γραφεί κάτι τέτοιο, τουλάχιστον για τον πρώτο τρόπο υλοποίησης ενός τύπου *bitmask*. Αλλά,

- για τον τρίτο τρόπο υλοποίησης, η πρόσθεση δεν είναι δεκτή ενώ
- για τον δεύτερο τρόπο θα πρέπει να βάλεις τυποθεωρήσεις ώστε να περνάει από όλους τους μεταγλωττιστές.

Φυσικά, μπορεί να το δεις γραμμένο κάπου αλλού αλλά εσύ *δεν θα πρέπει να το γράφεις* με βάση το εξής σκεπτικό: Αφού αυτό που κάνουμε είναι διαχείριση δυαδικών ψηφίων και όχι αριθμητική πράξη γιατί να βάλουμε το "+"; Μόνο και μόνο επειδή δουλεύει;

Στις προηγούμενες παραγράφους επισημάναμε ότι είναι επικίνδυνο και το αντίστροφο: να προσπαθήσεις να πάρεις αριθμητικά αποτελέσματα με ψηφιοπράξεις. Τα αριθμητικά αποτελέσματα θα τα παίρνεις με αριθμητικές πράξεις.

Καταλήγουμε λοιπόν στη σύσταση του (CERT 2009):[9]

♦ ***Απόφυγε να κάνεις ψηφιοπράξεις και αριθμητικές πράξεις στα ίδια δεδομένα.***

17.9 Παράσταση και Πράξεις στον Τύπο "`float`"

Όπως ο `int` σε σχέση με το σύνολο $\mathbb{Z}$ των ακεραίων, έτσι και ο τύπος `float`, σε σύγκριση με το σύνολο $\mathbb{R}$ των πραγματικών, έχει δυο σοβαρούς περιορισμούς:

- Υπάρχει μέγιστος και ελάχιστος αριθμός τύπου `float`.
- Κάθε αριθμός τύπου `float` παριστάνεται με πεπερασμένο πλήθος ψηφίων.

Συνήθως, έναν πραγματικό αριθμό x τον γράφουμε ως:

$\sigma\ \psi_n \psi_{n-1} \ldots \psi_0 . \psi_{-1} \psi_{-2} \ldots$

και με αυτό εννοούμε ότι:

$$x = \sigma(\psi_n 10^n + \psi_{n-1} 10^{n-1} \ldots + \psi_0 10^0 + \psi_{-1} 10^{-1} + \psi_{-2} 10^{-2} \ldots)$$

όπου σ: πρόσημο (+ ή −) και ψ_k: δεκαδικό ψηφίο.

Φυσικά, ο ΗΥ δεν μπορεί να αποθηκεύσει στην μνήμη του τα άπειρα ψηφία ενός άρρητου αριθμού. Κάθε αριθμός παριστάνεται με πεπερασμένο αριθμό ψηφίων. Έχουμε λοιπόν **απώλεια σημαντικών ψηφίων** (loss of significant digits). Αυτό το σφάλμα, που εισάγεται με την παράσταση των πραγματικών αριθμών, μεγαλώνει με την εκτέλεση των πράξεων μεταξύ τους.

Ας υποθέσουμε ότι δουλεύουμε σε έναν **δεκαδικό** υπολογιστή. Κάθε τιμή τύπου `float`, για να αποθηκευτεί, μετατρέπεται στη μορφή:

$$M \times 10^e \qquad (1)$$

και αποθηκεύονται η **μαντίσα** (mantissa) M και ο **εκθέτης** (exponent) e. Η μαντίσα έχει τη μορφή:

$$\sigma\ 0 . \psi_1 \psi_2 \psi_3 \psi_4 \psi_5$$

όπου σ το πρόσημο και ψ_k, $k = 1..5$ τα πέντε πιο σημαντικά ψηφία του αριθμού (με στρογγύλευση). Ο εκθέτης είναι διψήφιος ακέραιος και έχει τη μορφή:

$$\sigma\ \varepsilon_1 \varepsilon_2$$

Η παράσταση στη μορφή (1) γίνεται μονοσήμαντη αν κάνουμε τη σύμβαση ότι *το ψ_1 δεν μπορεί να είναι μηδέν*. Στην περίπτωση αυτήν λέμε ότι η παράσταση είναι **κανονικοποιημένη** (normalized). Θα παριστάνουμε την αποθηκευμένη πληροφορία, στη μορφή:

$$\sigma\psi_1 \psi_2 \psi_3 \psi_4 \psi_5 : \sigma\varepsilon_1 \varepsilon_2$$

---

[9] Σύσταση *INT*14: "*Avoid performing bitwise and arithmetic operations on the same data.*"

Αυτό είναι ένα παράδειγμα αποθήκευσης σε μορφή **κινητής υποδιαστολής** (floating point).

Το γνωστό μας $\pi = 3.1415926535...$, θα αποθηκευτεί ως:

+31416 : +01

(Θα γράφουμε π_f = +31416 : +01 ή π_f = 3.1416).

Η μέγιστη θετική τιμή που μπορεί να αποθηκευτεί είναι η:

+99999 : +99 (= 0.99999×10^{99})

και η ελάχιστη θετική τιμή:

+10000 : -99 (= 0.1×10^{-99})

Ένα χαρακτηριστικό της υλοποίησης του τύπου `float` είναι ο ελάχιστος θετικός ε που αν προστεθεί στο 1 μας δίνει τιμή μεγαλύτερη από 1:

$$\varepsilon = \min_{u>0}\{u \mid (1+u)_f \neq 1_f\}$$

Στον υπολογιστή μας, το 1 παριστάνεται ως:

+10000 : +01

και προφανώς η ελάχιστη αλλαγή που μπορούμε να του κάνουμε, είναι στο τελευταίο σημαντικό ψηφίο, κατά 1:

+10001 : +01 (= $0.10001 \times 10^{1} = 1.0001$)

Έχουμε λοιπόν ότι: $\varepsilon = 0.0001$.

Προσοχή όμως! Αυτό δεν σημαίνει ότι για κάθε $x \in$ `float` θα έχουμε και $(x + \varepsilon)_f \neq X_f$. Για παράδειγμα: $(10 + \varepsilon)_f = 10_f$ (Άσκ. 14-1). Πάντως το ε μας δείχνει πόσα σημαντικά ψηφία μπορούμε να παραστήσουμε: εφ' όσον μπορούμε να παραστήσουμε το $1+\varepsilon = 1.0001$ μπορούμε να παραστήσουμε 5 σημαντικά ψηφία. Αν αυτό φαίνεται τετριμμένο τώρα που δουλεύουμε στο δεκαδικό σύστημα, δεν είναι τετριμμένο όταν δουλεύουμε στο δυαδικό ή στα παράγωγά του, που δεν μας είναι και τόσο οικεία.

Μπορούμε λοιπόν να πούμε ότι ο τύπος `float` είναι υποσύνολο του συνόλου των πραγματικών:

- `float` $\subset \mathbb{R}$ ***(ακριβέστερα:*** `float` $\subset \mathbb{Q}$ ***(ρητοί))***

Η αποθήκευση πραγματικών τιμών στον υπολογιστή, είναι μια απεικόνιση από το σύνολο $\mathbb{R}$ στο υποσύνολό του `float`. Μερικές ιδιότητες αυτής της απεικόνισης θα δούμε στη συνέχεια.

Όπως φαίνεται από το παράδειγμά μας με το π, η αποθήκευση δεν γίνεται με ακρίβεια. Πάντως, για κάθε υπολογιστή, υπάρχει η εγγύηση ότι δυο τουλάχιστον πραγματικοί αριθμοί παριστάνονται με ακρίβεια: το 0 (μηδέν) και το 1 (ένα)!

Το 0 παριστάνεται ως:

+00000 : +00

και το 1 ως:

+10000 : +01 (= $0.1 \times 10^{1} = 1$)

- $\mathbf{0_f == 0}$ ***και*** $\mathbf{1_f == 1}$

Ο δείκτης $_f$ υποδηλώνει την παράσταση στο σύνολο `float`.

Αν μια πραγματική τιμή α παριστάνεται στο σύνολο `float` με την α_f, τότε και η $-\alpha$ παριστάνεται στο `float` και μάλιστα με την $-\alpha_f$:

$(-\alpha)_f == -\alpha_f$

Όπως είδαμε παραπάνω, η τιμή 3.1415926535..., θα αποθηκευτεί ως:

+31416 : +01

Αλλά με τον ίδιο τρόπο θα αποθηκευτεί και η τιμή 3.14156 και η 3.1416 κλπ. Γενικά:

- ***Αν οι τιμές*** α_1, α_2 ***παριστάνονται στον υπολογιστή με την ίδια τιμή*** $\tau \in$ `float`, ***τότε με την ίδια τιμή παριστάνονται όλες οι τιμές του διαστήματος*** $[\alpha_1, \alpha_2]$.

Από την ιδιότητα αυτή βγαίνουν τα εξής:

αν $\alpha > \beta$ τότε $\alpha_f \geq \beta_f$

αν $\alpha == \beta$ τότε $\alpha_f == \beta_f$

αν $\alpha < \beta$ τότε $\alpha_f \leq \beta_f$

Ας δούμε τώρα τι γίνεται με τις πράξεις.

Το πρώτο που θα δούμε είναι η **υπερχείλιση** και η **υποχείλιση**: είναι δυνατόν το αποτέλεσμα μιας πράξης να μην παριστάνεται γιατί είναι πολύ μεγάλο ή πολύ μικρό. Οι τιμές:

+60000:+99 (= 0.60000×10^{99})

και

+70000:+99 (= 0.70000×10^{99})

παριστάνονται στον υπολογιστή μας χωρίς πρόβλημα. Το άθροισμά τους όμως, $1.30000\times10^{99} = 0.13000\times10^{100}$ *δεν μπορεί να παρασταθεί* διότι είναι πολύ μεγάλο για τον υπολογιστή μας! Έχουμε δηλαδή **υπερχείλιση** (overflow). Τι θα κάνει ο υπολογιστής σε μια τέτοια περίπτωση; Σίγουρα θα σου δώσει μήνυμα για την κατάσταση που δημιουργήθηκε· στις περισσότερες περιπτώσεις θα σταματήσει και την εκτέλεση του προγράμματος.

Όπως καταλαβαίνεις, τέτοια αποτελέσματα μπορεί να βγουν και από τις τέσσερις πράξεις της αριθμητικής, όταν τις εκτελεί ο υπολογιστής. Δηλαδή:

- ***Το σύνολο `float` δεν είναι κλειστό ως προς τις πράξεις +, –, * και / .***

Καταλαβαίνεις ακόμη, ότι η υπερχείλιση στον τύπο `float` είναι λιγότερο επικίνδυνη από αυτήν του `int`, αφού αποφεύγεις την περίπτωση να συνεχιστεί η εκτέλεση του προγράμματος με τιμές που δεν έχουν νόημα. Έχει νόημα να γράψουμε κάτι σαν *addFloat*, για ασφαλή πρόσθεση τιμών `float`; Ναι, διότι συχνά ξέρεις τί πρέπει να κάνεις σε περίπτωση υπερχείλισης και οπωσδήποτε είναι ενοχλητικό να βλέπεις το πρόγραμμα να σταματάει, έστω και αν σου γνωστοποιεί τί έγινε.

Αν από τον:

+11000:-99 (= 0.11×10^{-99})

αφαιρέσουμε τον:

+10000:-99 (= 0.1×10^{-99})

το αποτέλεσμα $0.01\times10^{-99} = 0.1\times10^{-100}$ δεν μπορεί να παρασταθεί στον υπολογιστή μας, γιατί είναι πολύ μικρό! Στην περίπτωση αυτή λέμε ότι έχουμε **υποχείλιση** (underflow). Συνήθως, ο υπολογιστής σε μια τέτοια περίπτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!

Ας δούμε τώρα ένα άλλο επακόλουθο της πεπερασμένης παράστασης. Έστω ότι θέλουμε να προσθέσουμε με τον υπολογιστή μας τους αριθμούς 14.563 και 0.16773. Η αποθήκευσή τους θα γίνει με ακρίβεια:

+14563:+02 (= $0.14563\times10^{2} = 14.563$)

και

+16773:+00 (= $0.16773\times10^{0} = 0.16773$)

Για να τους προσθέσει ο υπολογιστής θα πρέπει πρώτα να τους μετασχηματίσει ώστε να έχουν τον ίδιο εκθέτη. Για την ακρίβεια μετασχηματίζει την τιμή με το μικρότερο εκθέτη (στρογγυλεύοντας σε πέντε θέσεις)[10]:

---

[10] Συνήθως, η Αριθμητική Μονάδα του υπολογιστή θα κάνει τις πράξεις με μεγαλύτερη ακρίβεια, αλλά η τιμή που θα μας επιστρέψει τελικά είναι σύμφωνη με τη μορφή που έχουμε στην αποθήκευση. Στην περίπτωσή μας, η δεύτερη τιμή θα γίνει:

+0016773:+02

Στη συνέχεια θα γίνει η πρόσθεση:

+1473073:+02

Αλλά, το αποτέλεσμα που θα είναι διαθέσιμο στο πρόγραμμά μας θα είναι:

+14731:+02

+00168 : +02

Στη συνέχεια κάνει την πρόσθεση:

+14731 : +02

Βλέπουμε λοιπόν, ότι και οι πράξεις εισάγουν **σφάλματα στρογγύλευσης** (roundoff errors). Πρόσεξε ότι, αν προσπαθούσαμε να προσθέσουμε στο 14.563 τον 0.0001, το αποτέλεσμα θα ήταν 14.563!

Θα πρέπει να έχει γίνει πια φανερό, ότι όπως ξεχωρίσαμε τις ακέραιες τιμές από τις παραστάσεις τους στον τύπο `int`, *θα πρέπει να ξεχωρίσουμε και τις πράξεις των πραγματικών από αυτές του υπολογιστή για τον τύπο* `float`. Θα παριστάνουμε λοιπόν με:

$+_f \quad -_f \quad *_f \quad /_f$

τις πράξεις:

$+ \quad - \quad \times \quad /$

όπως εκτελούνται στον υπολογιστή.

Στη συνέχεια θα δούμε μερικές «αναμενόμενες» ιδιότητες των πράξεων αλλά και μερικές «περίεργες». Το σύμβολο της ισότητας θα έχει πιο γενικό νόημα από ότι συνήθως: π_1 `==` π_2, όπου π_1, π_2 αριθμητικές παραστάσεις, θα σημαίνει ότι: αν μεν οι πράξεις γίνουν χωρίς πρόβλημα (υπερχείλιση ή υποχείλιση) οι τιμές των δύο παραστάσεων θα είναι ίσες· θα έχουμε υπερχείλιση ή υποχείλιση αριστερά αν και μόνον αν έχουμε το ίδιο πράγμα και δεξιά.

Η αντιμεταθετικότητα της πρόσθεσης και του πολλαπλασιασμού ισχύει στον τύπο `float`:

- ***Αν*** $a, b \in$ `float` ***τότε*** $a +_f b = b +_f a$ ***και*** $a *_f b = b *_f a$

Αυτές οι ισότητες ισχύουν με το γενικευμένο νόημα που δώσαμε πιο πάνω.

Η προσεταιριστικότητα της πρόσθεσης δεν ισχύει! Ας δούμε ένα

Παράδειγμα

Οι αριθμοί $a = 0.10000\times10^{94}$, $b = 0.55000\times10^{99}$, $c = -0.50000\times10^{99}$ παριστάνονται στον υπολογιστή μας με ακρίβεια. Το άθροισμα:

$$\begin{aligned}(a +_f b) +_f c &= (0.000001 +_f 0.55000)\times10^{99} +_f (-0.50000\times10^{99})\\ &= 0.55000\times10^{99} +_f (-0.50000\times10^{99})\\ &= 0.50000\times10^{98}\end{aligned}$$

δεν είναι ίσο με το:

$$\begin{aligned}a +_f (b +_f c) &= 0.10000\times10^{94} +_f (0.55000\times10^{99} +_f 0.50000\times10^{99})\\ &= 0.10000\times10^{94} +_f 0.05000\times10^{99}\\ &= 0.50001\times10^{98}\end{aligned}$$

Παρ' όλα αυτά, αν: $a, b \in$ `float` και $a \geq b \geq 0$, ισχύει η γνωστή μας ιδιότητα:

$(a -_f b) +_f b$ `==` a

δηλαδή: η πρόσθεση ακυρώνει την αφαίρεση.

Προβλήματα έχουμε και με την προσεταιριστικότητα του πολλαπλασιασμού: Αν πάρουμε: $a = 0.10000\times10^{60}$, $b = 0.10000\times10^{60}$, $c = 0.10000\times10^{-60}$ τότε το γινόμενο:

$$\begin{aligned}(a *_f b) *_f c &= (0.10000\times10^{60} *_f 0.10000\times10^{60}) *_f 0.10000\times10^{-60}\\ &= 0.10000\times10^{119} *_f 0.10000\times10^{-60} \quad \text{(υπερχείλιση!!)}\end{aligned}$$

δεν είναι ίσο με το:

$$\begin{aligned}a *_f (b *_f c) &= 0.10000\times10^{60} *_f (0.10000\times10^{60} *_f 0.10000\times10^{-60})\\ &= 0.10000\times10^{60} *_f 0.10000\times10^{-1}\\ &= 0.10000\times10^{58}\end{aligned}$$

Οι παραπάνω ενδεικτικές επισημάνσεις δεν εξαντλούν πλήρως τα προβλήματα του τύπου `float`, που ξεκινούν από την πεπερασμένη παράσταση. Αλλά δείχνουν μερικά χαρακτηριστικά σημεία που πρέπει να προσέχεις όταν γράφεις αριθμητικά προγράμματα.

17.9.1 Υπολογισμός Περιοδικής Συνάρτησης

Ας ξαναδούμε τώρα κάτι που μάθαμε παλιά υπό το φως αυτών που είδαμε παραπάνω.

Στην §7.8 μάθαμε να κάνουμε αναγωγή της τιμής του ορίσματος μιας περιοδικής συνάρτησης στο διάστημα ορισμού με τις εντολές:

```
    x0 = x;
    while ( x0 >= b )  x0 = x0 - T;
// (x0 < b) && (f(x0) == f(x))
    while ( x0 < a )   x0 = x0 + T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Τι θα γίνει αν η απόλυτη τιμή του x είναι πολύ μεγαλύτερη από την τιμή της περιόδου T; Στην περίπτωση αυτήν η εντολή `x0 = x0 - T` (ή `x0 = x0 + T`) *δεν αλλάζει την τιμή της* $x0$ και *η εκτέλεση της αντίστοιχης* ***while*** *δεν τελειώνει*. Θα γράφαμε πιο σωστά:

```
    x0 = x;
    if (x0 >= b)
    {
       if (x0 - T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 >= b)  x0 = x0 - T;
       // (x0 < b) && (f(x0) == f(x))
       }
    }
    else if (x0 < a)
    {
       if (x0 + T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 < a)   x0 = x0 + T;
       // (a ≤ x0 < b) && (f(x0) == f(x))
       }
    }
```

Θα πεις: καλύτερα τότε να δουλεύουμε με τον γρήγορο τρόπο αναγωγής. Ναι, αλλά πρόσεξε: είναι καλύτερο να γράψουμε:

```
    x0 = x - T*floor((x-a)/T);
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Όμως και στην περίπτωση αυτή, αν το $x - a$ είναι πολύ μεγαλύτερο από την περίοδο T τότε η $x0$ θα πάρει τιμή που δεν έχει και πολύ νόημα, πιθανότατα 0. Ακόμη, αν η περίοδος είναι μικρότερη από 1 είναι δυνατόν να έχουμε υπερχείλιση στη διαίρεση $(x - a)/T$.

17.10 Άλλοι Τύποι Κινητής Υποδιαστολής

Για να αντιμετωπισθούν τα προβλήματα του τύπου `float` η C++ –και άλλες γλώσσες προγραμματισμού– δίνουν στον προγραμματιστή άλλους αριθμητικούς τύπους με μερικές βελτιώσεις.

Η πιο συνηθισμένη περίπτωση είναι ο «διπλός `float`», που η C++ ονομάζει `double`. Οι τιμές αυτού του τύπου αποθηκεύονται σε χώρο (μνήμης) *διπλό* απ' όσον πιάνουν οι τιμές του τύπου `float`. Η αποθήκευση γίνεται συνήθως με κάποιον από τους παρακάτω τρόπους:

- Ο αριθμός θεωρείται σαν άθροισμα δύο τιμών τύπου `float`.
  - Η πρώτη έχει τα περισσότερο σημαντικά ψηφία του αριθμού και
  - η δεύτερη τα λιγότερο σημαντικά ψηφία.

  Για παράδειγμα, η τιμή 12.34567891234 θα γραφεί ως:

  $$12.345678912 = 12.345 + 0.00067891$$
  $$= 0.12345\times10^{2} + 0.67891\times10^{-3}$$

  και στις δυο θέσεις θα αποθηκευτούν τα:

  +12345:+02 και +67891:-03
- Ο δεύτερος τρόπος διαφέρει ως προς το περιεχόμενο της δεύτερης θέσης: εκεί απόθηκεύονται μόνο (τα λιγότερο) σημαντικά ψηφία. Δηλαδή, η δεύτερη θέση δεν είναι οργανωμένη όπως οι θέσεις τύπου `float` (δεν αποθηκεύεται εκθέτης). Για το παραπάνω παράδειγμα θα αποθηκευτούν τα εξής:

  +12345:+02 και 6789123
- Μια παραλλαγή του δεύτερου τρόπου, που επικρατεί στους νέους υπολογιστές, επιτρέπει και την αύξηση των θέσεων του εκθέτη. Στην περίπτωση αυτήν η περιοχή τιμών που μπορεί να παρασταθεί είναι ευρύτερη από αυτήν που παριστάνεται στον τύπο `float`. Σε μια τέτοια περίπτωση ο αριθμός μας θα αποθηκεύονταν ως:

  +1234:+002 και 5678912

  Ανακεφαλαιώνοντας, μπορούμε να πούμε για τον "διπλό `float`":
- Επιτρέπει την αποθήκευση περίπου διπλού πλήθους σημαντικών ψηφίων απ' ότι ο τύπος `float`.
- Μπορεί να επιτρέπει την παράσταση ευρύτερης περιοχής τιμών απ' ότι ο τύπος `float`, αλλά αυτό δεν είναι αναγκαίο.
- Κάθε τιμή του καταλαμβάνει διπλό χώρο μνήμης απ' ότι ο τύπος `float`.

  Φυσικά, τα προβλήματα του τύπου `float` δεν λύνονται· απλώς μετατοπίζονται.

17.11 Ο Τύπος "`float`" στο Δυαδικό Σύστημα

Μέχρι τώρα χρησιμοποιήσαμε το δεκαδικό σύστημα για να δούμε μερικά από αυτά που συμβαίνουν στους τύπους `float`. Αλλά οι ψηφιακοί υπολογιστές δουλεύουν συνήθως με το **δυαδικό** (binary) σύστημα και τα παράγωγά του **οκταδικό** (octal) και **δεκαεξαδικό** (hexadecimal). Τώρα θα δούμε μερικά από τα παραπάνω στο δυαδικό σύστημα.

Στο δυαδικό σύστημα η παράσταση μιας τιμής τύπου `float` γίνεται ως εξής:

$$M \times 2^{e}$$

Η **μαντίσα** M και ο **εκθέτης** e που αποθηκεύονται, είναι δυαδικοί αριθμοί. Ο εκθέτης είναι ακέραιος και έχει τη μορφή:

$$s\ \varepsilon_1\ \varepsilon_2 \ldots\ \varepsilon_L$$

όπου s πρόσημο και ε_k, $k = 1..L$ τα L δυαδικά ψηφία.

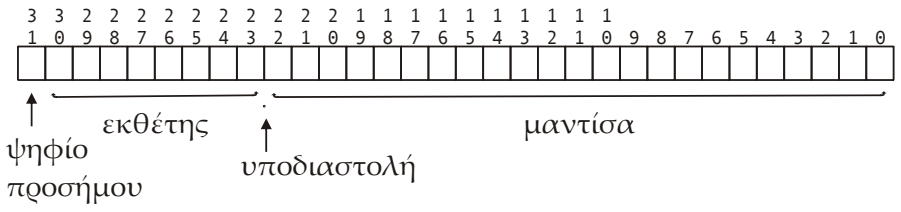
Η μαντίσα μπορεί να έχει τη μορφή:

$$s\ 0.\psi_1\psi_2\ldots\psi_N$$

ή

$$s\ \psi_1\psi_2\ldots\psi_N.0$$

Συνήθως η παράσταση κανονικοποιείται, δηλαδή *το ψ_1 δεν μπορεί να είναι μηδέν*. Στο δυαδικό σύστημα αυτό σημαίνει ότι $\psi_1 = 1$.



Σχ. 17-5 Παράσταση κινητής υποδιαστολής.

Μια τέτοια παράσταση βλέπουμε στο Σχ. 17-5.

Στο ψηφίο 31 αποθηκεύεται το πρόσημο του αριθμού –"**0**" για το "**+**", "**1**" για το "**-**". Στα ψηφία 30 - 23 αποθηκεύεται ο εκθέτης: Στο ψηφίο 30 το πρόσημό του (όπως παραπάνω) και στα ψηφία 29 - 23 η απόλυτη τιμή του. Στα ψηφία 22 - 0 αποθηκεύεται η μαντίσα. Το πρώτο της ψηφίο (22) είναι πάντοτε 1, εκτός αν ο αριθμός είναι 0 (μηδέν). Η υποδιαστολή νοείται ακριβώς πριν από το ψηφίο αυτό (22). Δηλαδή:

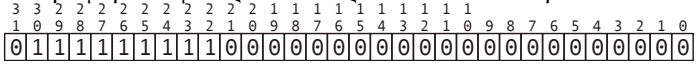
$$e=\begin{cases}\varepsilon_{29}\varepsilon_{28}\ldots\varepsilon_{23}, & \text{αν } \varepsilon_{30}=0\\ -\varepsilon_{29}\varepsilon_{28}\ldots\varepsilon_{23}, & \text{αν } \varepsilon_{30}=1\end{cases}$$

$$M=\begin{cases}0.\psi_{22}\psi_{21}\ldots\psi_{0}, & \text{αν } s_{31}=0\\ -0.\psi_{22}\psi_{21}\ldots\psi_{0}, & \text{αν } s_{31}=1\end{cases}$$

Ο εκθέτης μπορεί να παίρνει (ακέραιες) τιμές, από –127 μέχρι +127. Η ελάχιστη μη μηδενική τιμή της μαντίσας μπορεί να είναι $0.1_2 = 0.5_{10}$. Η μέγιστη σχηματίζεται όταν όλα τα ψηφία (22 - 0) είναι 1:

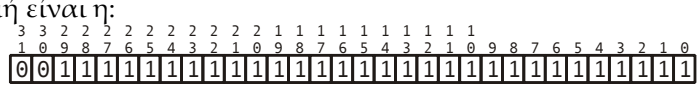
$2^{-1} + 2^{-2} + \ldots + 2^{-23} = 1 - 2^{-23} \approx 0.99999988 \approx 1$

Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί είναι η:



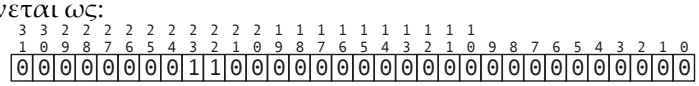
Δηλαδή, ελάχιστη μη-μηδενική μαντίσα ($0.1_2 = 0.5_{10}$) και ελάχιστος εκθέτης (-127). Προφανώς πρόκειται για το $2^{-128} \approx 0.29387e\text{-}38$.

Η μέγιστη τιμή είναι η:



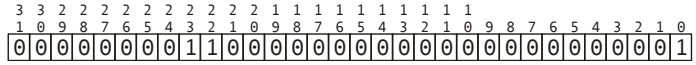
που σημαίνει: μέγιστη μαντίσα (≈ 1) και μέγιστος εκθέτης (+127). Η τιμή είναι: $2^{127} \approx 0.17014e{+}39$.

Το 1 παριστάνεται ως:



δηλαδή: $0.1_2 \times 2^1$.

Ο πλησιέστερος μεγαλύτερος αριθμός είναι αυτός που προκύπτει αν αλλάξουμε σε 1 το ψηφίο της θέσης 0.



Η διαφορά του από το 1 είναι $2^{-23} \times 2^1 \approx 0.23842e{-}06$. Αυτό είναι το ε για την παράσταση αυτή.

17.11.1 Πόλωση

Η μορφή πρόσημο - απόλυτη τιμή δεν είναι η καλύτερη δυνατή για τον εκθέτη:

- οι πράξεις είναι πολύπλοκες και
- περιορίζεται λιγάκι η περιοχή τιμών (έχουμε δυο παραστάσεις για το 0: +0, –0).

Ένα απλό τέχνασμα μας απαλλάσσει από τα παραπάνω: όταν αποθηκεύουμε μια τιμή, ο εκθέτης δεν αποθηκεύεται όπως είναι, αλλά αφού πρώτα του προσθέσουμε 127. Μέ τον τρόπο αυτόν ο εκθέτης είναι πάντοτε μη αρνητικός. Λέμε ότι ο εκθέτης είναι **πολωμένος** (biased) κατά 127.

Πρόσεξε ότι όταν κάνουμε πράξεις μεταξύ τιμών τύπου `float` δεν χρειάζεται να κάνουμε κάτι ιδιαίτερο λόγω της πόλωσης του εκθέτη. Τώρα οι πράξεις είναι πιο απλές (δεν υπάρχει πρόσημο) και οι τιμές του εκθέτη μπορούν να είναι από 0 μέχρι 255 ή πραγματικές τιμές –127 μέχρι 128. Έχουμε δηλαδή 256 διαφορετικές τιμές αντί για 255 που είχαμε χωρίς πόλωση.

Όταν όμως θέλουμε να τυπώσουμε τον αριθμό, θα πρέπει να αφαιρούμε από τον εκθέτη το 127. Στην παράσταση:

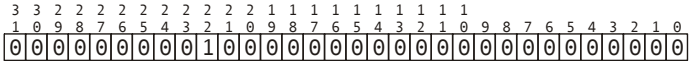
$$M\times 2^{e}$$

το νόημα του e αλλάζει:

$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

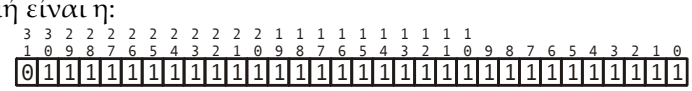
Όταν έχουμε πόλωση,

- Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί έχει τη μορφή:



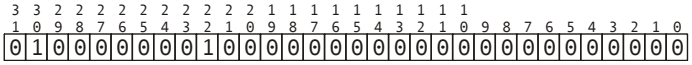
  ελάχιστος εκθέτης: 0 –στην πραγματικότητα 0 – 127 = –127– και φυσικά ελάχιστη μημηδενική μαντίσα ($0.1_2 = 0.5_{10}$). Προφανώς πρόκειται για την τιμή που υπολογίσαμε και πριν: $2^{-128} \approx 0.29387e{-}38$.

- Η μέγιστη τιμή είναι η:



  μέγιστος εκθέτης: 255 –με πραγματική τιμή 255 – 127 = 128– και μέγιστη μαντίσα (≈ 1). Η τιμή είναι: $2^{128} \approx 0.34028e{+}39$, δηλαδή διπλή από αυτήν που είχαμε χωρίς πόλωση.

- Το 1 παριστάνεται ως:



  δηλαδή: $0.1_2\times 2^{128-127}$.

  Είναι φανερό ότι το e δεν αλλάζει.

17.11.2 Άλλες Περιπλοκές – Πρότυπο IEEE

Μια άλλη παρατήρηση που μπορούμε να κάνουμε είναι η εξής: Αφού η τιμή του πρώτου ψηφίου της μαντίσας είναι πάντοτε 1, δεν υπάρχει λόγος να αποθηκεύεται· αυτό και γίνεται σε ορισμένες παραστάσεις. Βεβαίως, όπως είπαμε, είναι πάντοτε 1, εκτός από μια περίπτωση: όταν αποθηκεύουμε το 0. Θα πρέπει λοιπόν να κάνουμε ορισμένες συμβάσεις. Θα δούμε μερικές τέτοιες συμβάσεις που γίνονται στο πρότυπο της IEEE για αριθμητική κινητής υποδιαστολής, μια και το πρότυπο αυτό το συναντούμε σε πολλούς σύγχρονους υπολογιστές.

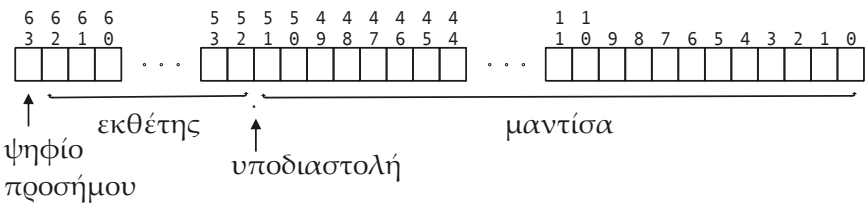
Χρησιμοποιούμε και πάλι 32 δυαδικά ψηφία (31 - 0), στο ψηφίο 31 έχουμε το πρόσημο και στα 30 - 23 έχουμε τον εκθέτη, πολωμένο κατά 127.

Για τη μαντίσα όμως τα πράγματα αλλάζουν: ακριβώς πριν από το ψηφίο 22 νοείται ότι υπάρχει το "`.1`". Έτσι, το νόημα της μαντίσας είναι:

$$M = \begin{cases} 0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu s_{31}=0 \\ -0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu s_{31}=1 \end{cases}$$

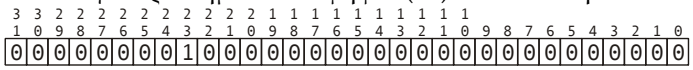
$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

Το μηδέν παριστάνεται με 32 δυαδικά ψηφία ίσα με 0. Αυτή είναι μια *σύμβαση*, αφού σύμφωνα με όσα είπαμε ο αριθμός αυτός θα έπρεπε να είναι ίσος με: $1\times 2^{-127} = 5.877\times 10^{-39}$. Γενικώς, υπάρχει μια σύμβαση (IEEE) για τις παραστάσεις που έχουν όλα τα ψηφία του εκθέτη 0. Πρόκειται για **μη-κανονικοποιημένες** (denormalized ή subnormal) παραστάσεις (πολύ μικρών) αριθμών.



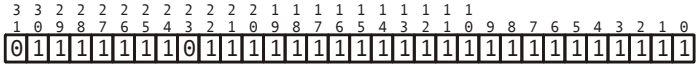
Σχ. 17-6 Τύπος **double**.

- Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει 1 το λιγότερο σημαντικό ψηφίο (23) του εκθέτη και όλα τα άλλα 0:



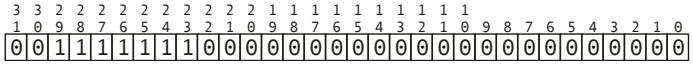
  δηλαδή: $1\times2^{1-127} = 2^{-126} \approx 0.11755e\text{-}37$.

- Η παράσταση του μέγιστου θετικού είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (23), ίσα με 1 και όλα τα ψηφία της μαντίσας 1:

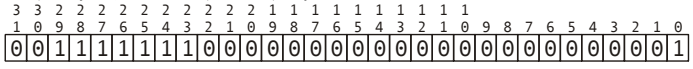


  Η τιμή αυτή είναι περίπου: $2\times2^{254-127} = 2^{128} \approx 0.34028e\text{+}39$.

- Το 1 παριστάνεται ως:



  ενώ ο πλησιέστερος μεγαλύτερος αριθμός είναι ο:



  Το ε είναι $2^{-23} \approx 1.1921e\text{-}07$. Μπορούμε δηλαδή να παραστήσουμε 24 σημαντικά δυαδικά ψηφία, που ισοδυναμούν με 7 δεκαδικά ψηφία περίπου.

- Το **άπειρο** (**INF**) παριστάνεται ως εξής: όλα τα ψηφία του εκθέτη 1 και όλα τα ψηφία της μαντίσας 0.

- Αν όλα τα ψηφία του εκθέτη είναι 1 και ένα τουλάχιστον ψηφίο της μαντίσας δεν είναι 0, *η παράσταση θεωρείται ότι δεν παριστάνει αριθμό* (**N**ot **a** **N**umber, **NaN**).

17.11.3 Οι Τύποι "double" και "long double"

Ο τύπος **double** της C++ είναι «διπλός **float**» (§17.10). Θα δώσουμε τώρα μια παράσταση σύμφωνη με το πρότυπο της IEEE.

Όπως προαναφέραμε, ο χώρος που καταλαμβάνεται από μια θέση τύπου **double** καταλαμβάνει στη μνήμη διπλό χώρο απ' ότι μια θέση τύπου **float**. Αν λοιπόν αποθηκεύουμε μια τιμή τύπου **float** σε 32 δυαδικά ψηφία, μια τιμή τύπου **double** θα αποθηκεύεται σε 64 δυαδικά ψηφία, που τα αριθμούμε από 0 μέχρι 63 (Σχ. 17-6).

Ο εκθέτης αποθηκεύεται σε 11 δυαδικά ψηφία (62 - 52) πολωμένος κατά 1023, ενώ η μαντίσα αποθηκεύεται σε 52 ψηφία (51 - 0).

Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει το λιγότερο σημαντικό ψηφίο (52) του εκθέτη ίσο με 1 και όλα τα άλλα 0: δηλαδή: $1\times2^{1-1023} = 2^{-1022} \approx 0.22251e\text{-}307$. Με μη κανονικοποιημένες μορφές μπορεί να παρασταθούν και μικρότεροι αριθμοί, μέχρι $5.0e\text{-}324$.

Η παράσταση του μέγιστου ακέραιου είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (52), ίσα με 1 και όλα τα ψηφία της μαντίσας 1: Η τιμή αυτή είναι περίπου: $2\times2^{2046-1023} = 2^{1024} \approx 0.17977e\text{+}309$

Το ε είναι $2^{-52} \approx 2.220446049250e\text{-}16$. Μπορούμε δηλαδή να παραστήσουμε 53 σημαντικά ψηφία στο δυαδικό σύστημα ή περίπου 16 σημαντικά ψηφία στο δεκαδικό.

Η C++ δίνει και τον τύπο **long double**, οι τιμές του οποίου αποθηκεύονται σε 10 ψηφιολέξεις, δηλ. 80 δυαδικά ψηφία, με 15ψήφιο εκθέτη και 64ψήφια μαντίσα. Δίνει ακρίβεια 20 (δεκαδικών) ψηφίων με τιμές από 1.9*e*-4951 μέχρι 1.1*e*4932.

Οι παραπάνω τύποι, **float**, **double**, **long double** δεν είναι αποκλειστικότητα της C++· θα τους βρεις και σε άλλες γλώσσες προγραμματισμού.

17.11.4 Μερικά Χρήσιμα Εργαλεία από τη C

Η C μας δίνει μερικές (μακρο)συναρτήσεις για να καταλαβαίνουμε τι συμβαίνει με παραστασεις τιμών κινητής υποδιαστολής. Οι ορισμοί τους υπάρχουν στο **cmath**.

int fpclassify(*τύπος κινητής υποδιαστολής* **x);**

όπου:

τύπος κινητής υποδιαστολής **= "float" | "double" | "long double" ;**

Η *fpclassify()* επιστρέφει ως τιμή μια από τις *αμοιβαίως αποκλειόμενες* τιμές **FP_INFINITE**, **FP_NAN**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO** που ορίζονται (με "**#define**") επίσης στο **cmath**.

Μας δίνει ακόμη τα εξής κατηγορήματα:

int isinf(*τύπος κινητής υποδιαστολής* **x);**
int isnan(*τύπος κινητής υποδιαστολής* **x);**
int isnormal(*τύπος κινητής υποδιαστολής* **x);**
int isfinite(*τύπος κινητής υποδιαστολής* **x);**

Σημείωση: ▶

Για να τις χρησιμοποιήσεις θα πρέπει να τις σκέφτεσαι ως

bool isinf(*τύπος κινητής υποδιαστολής* **x);**
bool isnan(*τύπος κινητής υποδιαστολής* **x);**
bool isnormal(*τύπος κινητής υποδιαστολής* **x);**
bool isfinite(*τύπος κινητής υποδιαστολής* **x);**

Έτσι, μπορείς να γράφεις "**if (isnan(sqrt(x)) …**"◀

Το *isnormal()* επιστρέφει "**1**" ("**true**") αν η *x* έχει κανονικοποιημένη παράσταση (ούτε "**0**", ούτε μη-κανονικοποιημένη, ούτε άπειρο, ούτε NaN).

Το *isfinite()* επιστρέφει "**1**" ("**true**") αν η *x* δεν είναι άπειρο ούτε NaN.

Για την (πεπερασμένη!) παράσταση του απείρου ορίζονται οι σταθερές **HUGE_VALF**, **HUGE_VAL**, **HUGE_VALL** για τους **float**, **double**, **long double** αντιστοίχως.[11]

Το παρακάτω προγραμματάκι χρησιμοποιεί τα παραπάνω εργαλεία:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   float x( 0 );
   double dn( 1e-100 ), dx( 1e100 );

   if ( fpclassify(x) == FP_ZERO )
      cout << "x is ZERO" << endl;
   cout << dn << "  " << isinf(dn) << "  " << isnan(dn)
        << "  " << isnormal(dn) << "  " << isfinite(dn) << endl;
   cout << dx << "  " << isinf(dx) << "  " << isnan(dx)
        << "  " << isnormal(dx) << "  " << isfinite(dx) << endl;
   x = dn;
   cout << x << "  " << isinf(x) << "  " << isnan(x)
        << "  " << isnormal(x) << "  " << isfinite(x) << endl;
```

[11] Στο πρότυπο C99 υπάρχει πρόβλεψη και για άλλο (πιο «πραγματικό») άπειρο, το **INFINITY**, για υλοποιήσεις που μπορούν να το υποστηρίξουν.

```
   x = dx;
   cout << x << "  " << isinf(x) << "  " << isnan(x)
        << "  " << isnormal(x) << "  " << isfinite(x) << endl;
   if ( x == HUGE_VALF )
      cout << "It IS HUGE!!!" << endl;
   cout << 1/INFINITY << endl;
}
```

και μας δίνει:

```
x is ZERO
1e-100  0  0  1  1
1e+100  0  0  1  1
0  0  0  0  1
1.#INF  1  0  0  0
It IS HUGE!!!
0
```

17.12 Σφάλμα από Μετατροπή Τύπου

Ξεκινάμε με ένα μικρό προγραμματάκι:

```
int main()
{
   long   i1, i2;
   float  r1, r2;

   i1 = 123456787;
   i2 = 123456786;
   r1 = i1;  r2 = i2;
   if ( r1 == r2)  cout << "E, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!" << endl;
}
```

Μεταγλωττίζουμε το πρόγραμμα αυτό με τη Borland C++ όπου οι τιμές τύπων `long` και `float` παριστάνονται με 32 δυαδικά ψηφία, όπως είδαμε παραπάνω. Ακόμα, χειρίζεται την παράσταση κινητής υποδιαστολής σύμφωνα με το πρότυπο της IEEE. Το αποτέλεσμα; Νάτο:

```
E, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!
```

Ας προσπαθήσουμε να βγάλουμε άκρη: Οι μεταβλητές *i*1 και *i*2 παίρνουν τις τιμές τους χωρίς κανένα πρόβλημα, μια και βρίσκονται μέσα στα όρια των τιμών που μπορούν να πάρουν (§4.3). Στην εντολή "`r1 = i1`" γίνονται τα εξής: πριν αποθηκευτεί η τιμή του *i*1 στην θέση της μνήμης *r*1 μετατρέπεται σε παράσταση κινητής υποδιαστολής. Αλλά, όπως είδαμε στην §4.3, μπορούμε να κρατήσουμε το πολύ 7 σημαντικά ψηφία. Έτσι, η «λεπτομέρεια» των δυο τελευταίων ψηφίων χάνεται. Το ίδιο συμβαίνει και με την εκτέλεση της "`r2 = i2`". Στην επόμενη εντολή –"`if (r1 == r2) cout <<...`"– όταν γίνεται η σύγκριση των *r*1 και *r*2, αυτές βρίσκονται ίσες αφού η σύγκριση γίνεται με τα πρώτα έξη ψηφία!

Φτάνουμε λοιπόν στο συμπέρασμα: *Κατά τη μετατροπή τύπου από ακέραιο τύπο σε τύπο κινητής υποδιαστολής μπορεί να έχουμε απώλεια σημαντικών ψηφίων.*

17.13 Τα Σφάλματα και πώς Μεταδίδονται

Αφού είδαμε τι συμβαίνει με την παράσταση στον τύπο `float` στο δεκαδικό σύστημα και στο δυαδικό, θα δούμε τώρα την παράσταση πιο γενικά· θα βγάλουμε ένα άνω φράγμα για το σχετικό σφάλμα από την παράσταση.

Στη συνέχεια θα δούμε πως μεταδίδονται τα σφάλματα από τους αριθμούς στις πράξεις μεταξύ τους.

17.13.1 Το Σφάλμα Παράστασης

Ας υποθέσουμε ότι έχουμε έναν υπολογιστή που δουλεύει σε σύστημα με βάση b, μπορεί να παραστήσει N ψηφία στο σύστημα αυτό. Το b είναι συνήθως 2, 8, 10, 16. Ας υποθέσουμε ότι η παράσταση ενός αριθμού x γίνεται στη μορφή:

$$x_f = M \times b^e$$

όπου:

$$M = \pm(\psi_{-1}b^{-1} + \psi_{-2}b^{-2} \ldots \psi_{-N}b^{-N})$$
$$0 \le \psi_k \le b - 1, \quad k = 1 \,..\, N$$
$$E_E \le e \le E_M$$

Όταν παριστάνουμε έναν αριθμό κρατάμε τα N πιο σημαντικά ψηφία του. Μπορούμε να πάρουμε μια τέτοια προσέγγιση είτε με **στρογγύλευση** (rounding) είτε με **αποκοπή** (truncation, choping). Να δούμε τι σφάλμα κάνουμε στις δυο περιπτώσεις.

Στρογγύλευση: Αν το ψηφίο (N-1) τάξης είναι $\ge b/2$ τότε το ψ_N αυξάνεται κατά μια μονάδα. Επομένως το (απόλυτο) σφάλμα που κάνουμε, $|x_f - x|$, είναι το πολύ:

$$|x_f - x| \le (b/2) \times b^{-(N+1)} \times b^e = 0.5 \times b^{e-N}$$

Από τη σχέση αυτή μπορούμε να υπολογίσουμε το μέγιστο **σχετικό σφάλμα** (relative error) λόγω της στρογγύλευσης:

$$\frac{|x_f - x|}{|x|} \le 0.5\frac{b^{e-N}}{|x|} = 0.5\frac{b^{e-N}}{(0.\psi_1\psi_2\ldots)b^e} = 0.5\frac{b^{-N}}{0.\psi_1\psi_2\ldots}$$

Αλλά, $0.\psi_1\psi_2\ldots_b \ge 0.100\ldots_b$ ($= b^{-1}$) και έτσι:

$$\Sigma\chi\varepsilon\tau(x_f) = \frac{|x_f - x|}{|x|} \le 0.5b^{-N+1}$$

Ας εφαρμόσουμε αυτή τη σχέση στις παραστάσεις που είδαμε στις προηγούμενες παραγράφους. Πρώτα στον δεκαδικό υπολογιστή: εδώ έχουμε $b = 10$, $N = 5$, άρα, για κάθε x:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.5\times10^{-4} \quad \text{ή} \quad \Sigma\chi\varepsilon\tau(x_f) \le 0.00005$$

Στη δυαδική παράσταση της §14.7: $b = 2$, $N = 23$, άρα:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.5\times2^{-23}$$

Αλλά, $0.5\times2^{-23} = 2^{-24} \approx 10^{-7.22} \le 0.6\times10^{-7}$ και μπορούμε να γράψουμε:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.00000006$$

Αποκοπή: Στην περίπτωση αυτή αποκόπτονται όλα τα ψηφία μετά το N-οστό. Όλο το τμήμα που αποκόπτεται είναι μικρότερο από μια μονάδα της τάξης του N-οστού ψηφίου. Δηλαδή:

$$|x_f - x| < 1\times b^{e-N}$$

Έτσι, το φράγμα του σχετικού σφάλματος είναι διπλάσιο αυτού που είχαμε στην περίπτωση της στρογγύλευσης:

$$\Sigma\chi\varepsilon\tau(x_f) = \frac{|x_f - x|}{|x|} < b^{-N+1}$$

17.13.2 Μετάδοση Σφαλμάτων

Έχουμε δυο τιμές x, y και τις παριστάνουμε στον υπολογιστή μας ως x_f, y_f σε παράσταση κινητής υποδιαστολής. Θα έχουμε:

$$x = x_f + \delta x \qquad y = y_f + \delta y$$

Ας συμβολίσουμε με π μια από τις πράξεις +, −, ×, / και με π_f την αντίστοιχη πράξη του υπολογιστή μας, μια από τις: $+_f$, $-_f$, $*_f$, $/_f$. Για το σφάλμα του αποτελέσματος θα έχουμε:

$$(x \pi y) - (x_f \pi_f y_f) = (x \pi y) - (x_f \pi_f y_f) + (x_f \pi y_f) - (x_f \pi y_f)$$
$$= (x_f \pi y_f - x_f \pi_f y_f) + (x \pi y - x_f \pi y_f)$$

Ο πρώτος όρος αναφέρεται μόνο σε τιμές που παριστάνονται ήδη στον υπολογιστή. Μας δίνει το σφάλμα που γίνεται από την *πράξη* π_f. Αν γυρίσεις στην §17.8 μπορείς να δεις πώς κάναμε στον δεκαδικό υπολογιστή την πρόσθεση των 14.563 (= x_f) και 0.16773 (= y_f). Ο πρώτος όρος για την περίπτωση αυτή είναι:

$$(x_f + y_f) - (x_f +_f y_f) = (14.563 + 0.16773) - (14.563 +_f 0.16773)$$
$$= 14.73073 - 14.731$$
$$= -0.0008$$

Αν υποθέσουμε ότι, όπως λέγαμε στην §14.8, η Αριθμητική Μονάδα κάνει τις πράξεις με μεγαλύτερη ακρίβεια και το αποτέλεσμα στρογγυλεύεται (ή αποκόπτεται) στη συνέχεια, μπορούμε να πούμε ότι:

$$(x_f \pi_f y_f) = (x_f \pi y_f)_f$$

Οπότε, σύμφωνα με αυτά που είπαμε για το σφάλμα στρογγύλευσης, στην προηγούμενη παράγραφο, έχουμε:

$$|x_f \pi y_f - x_f \pi_f y_f| \le 0.5 |x_f \pi y_f| b^{1-N}$$

Ας έρθουμε τώρα στο δεύτερο όρο:

$$(x \pi y - x_f \pi y_f)$$

Εδώ *οι πράξεις είναι ακριβείς*, αλλά συγκρίνουμε το αποτέλεσμα που παίρνουμε από τις αληθείς τιμές με αυτό που παίρνουμε από τις (προσεγγιστικές) τιμές σε μορφή κινητής υποδιαστολής. Αυτό λέγεται **μεταδιδόμενο σφάλμα** (propagated error). Θα δούμε στη συνέχεια το μεταδιδόμενο σφάλμα για τις τέσσερις πράξεις.

Πολλαπλασιασμός:

$$xy - x_f y_f = xy - (x - \delta x)(y - \delta y)$$
$$= x\delta y + y\delta x - \delta x \delta y$$

και

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f) + \Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f)$$

Αν $\Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)$$

Διαίρεση:

$$\Sigma\chi\varepsilon\tau\left(\frac{x_f}{y_f}\right) = \frac{|x/y - x_R/y_R|}{|x/y|} = \frac{|y_R/y - x_R/x|}{|y_R/y|}$$
$$= \frac{|\delta x/x - \delta y/y|}{|1 - \delta y/y|} \le \frac{\Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)}{|1 - \Sigma\chi\varepsilon\tau(y_f)|}$$

Και εδώ, αν $\Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)$$

Όπως βλέπεις, στον πολλαπλασιασμό και τη διαίρεση τα σφάλματα μεταδίδονται αργά. Είναι, θα μπορούσαμε να πούμε, αρκετά «σίγουρες» πράξεις. Δεν συμβαίνει όμως το ίδιο με τις άλλες δυο πράξεις που έχουμε αφήσει τελευταίες.

Πρόσθεση - Αφαίρεση: Αν χρησιμοποιήσουμε το π για το + ή το -, το απόλυτο σφάλμα (που αντιστοιχεί στο δεύτερο όρο) είναι:

$$x \pi y - x_f \pi y_f = x \pi y - (x - \delta x) \pi (y - \delta y)$$
$$= \delta x \pi \delta y$$
$$\le |\delta x| + |\delta y|$$

Ας δούμε τι σημαίνει αυτό, με ένα παράδειγμα: έχουμε τους αριθμούς 0.123445 και 0.123454 και χρησιμοποιούμε το δεκαδικό μας υπολογιστή για να υπολογίσουμε τη διαφορά

τους. Ο πρώτος παριστάνεται ως: 0.12345 (σφάλμα -0.000005) και ο δεύτερος: 0.12345 (σφάλμα 0.000004) και η διαφορά τους υπολογίζεται: 0. Το ακριβές αποτέλεσμα είναι 0.000009 και το σχετικό σφάλμα είναι 100%! Για να δούμε τι έγινε εδώ: Έχουμε δυο αριθμούς με έξη ψηφία και παραπλήσιες απόλυτες τιμές –η διαφορά τους βρίσκεται στα δυο τελευταία ψηφία. Για να παρασταθούν οι αριθμοί στον υπολογιστή, που δέχεται μόνο 5 ψηφία, γίνεται το καλύτερο δυνατό: στρογγύλευση σε 5 ψηφία. Τώρα, τα τέσσερα πρώτα –και σωστά– ψηφία είναι ίδια ενώ τα 5α ψηφία είναι λάθος, λόγω της στρογγύλευσης. Το αποτέλεσμα υπολογίστηκε ακριβώς με βάση τα 5α ψηφία.

Μήπως αυτό ήταν ένα «παρατραβηγμένο» παράδειγμα; Πόσο συχνά συμβαίνει κάτι τέτοιο; Σχεδόν κάθε φορά που έχουμε να αφαιρέσουμε δυο αριθμούς με παραπλήσιες απόλυτες τιμές. Οι αριθμοί αυτοί συνήθως προκύπτουν από (μη ακριβείς) πράξεις και έχουν κάποιο σφάλμα. Το σφάλμα αυτό βρίσκεται στα τελευταία (λιγότερο σημαντικά) ψηφία. Έτσι, με μια αφαίρεση, παίρνουμε αποτέλεσμα που μπορεί να είναι τελείως λάθος, όπως στο παράδειγμά μας.

17.14 Ισότητα στους Τύπους Κινητής Υποδιαστολής

Τα παραπάνω γεννούν και ένα άλλο ερώτημα: αν έχουμε δηλώσει:

```
float A, B;
```

πόσο νόημα έχει η σύγκρισή τους για ισότητα:

```
if ( A == B ) ...
```

Ας πούμε α, β τις ακριβείς τιμές των ποσοτήτων που φιλοδοξούμε να υπολογίσουμε στο πρόγραμμά μας με τα A, B αντίστοιχα. Οι πράξεις που θα κάνουμε στον υπολογιστή μας θα εισαγάγουν σφάλματα και τελικώς:

$$|A - \alpha| \le \delta A \quad \text{και} \quad |B - \beta| \le \delta B$$

ή

$$\alpha - \delta A \le A \le \alpha + \delta A \quad \text{και} \quad \beta - \delta B \le B \le \beta + \delta B$$

Από αυτές παίρνουμε:

$$(\alpha - \beta) - (\delta A + \delta B) \le A - B \le (\alpha - \beta) + (\delta A + \delta B)$$

Αυτό που μας ενδιαφέρει βεβαίως, είναι να συγκρίνουμε για ισότητα τα α και β. Αν $\alpha = \beta$ τότε για τα A και B έχουμε:

$$-(\delta A + \delta B) \le A - B \le (\delta A + \delta B)$$

Όπως φαίνεται λοιπόν, το σωστό είναι να ρωτήσουμε:

```
if ( fabs(A - B) <= eps ) ...
```

Το *eps*, κατ' αρχήν, εξαρτάται από

- το συγκεκριμένο πρόβλημα, αφού εξαρτάται από τα σφάλματα παράστασης και πράξεων που συσσωρεύτηκαν κατά τους υπολογισμούς των A και B.
- την ακρίβεια που απαιτούμε στον υπολογισμό του A ή του B.

Αλλά, το *eps* έχει περιορισμούς από τον τύπο `float` που δουλεύουμε: *δεν μπορεί να είναι μικρότερο* από το

$$\delta A = min\{\, u: \mathbb{R} \mid u > 0 \wedge (|A|+u)_R \neq |A|_R \bullet u \,\}$$

Αποδεικνύεται ότι για το δA έχουμε:

$$|A|\varepsilon/b < \delta A \le |A|\varepsilon$$

όπου b η βάση του αριθμητικού συστήματος του υπολογιστή σου και ε το *έψιλον* του τύπου `float`. Αν έχεις $b = 2$, τότε $|A|\varepsilon/2 < \delta A \le |A|\varepsilon$.

Από τον ορισμό έχεις ότι: αν $0 \le x < \delta A$ τότε αν $B = A + x$ θα πρέπει να δεχτείς ότι A `==` B αφού ο υπολογιστής σου δεν μπορεί να σου δώσει κάτι ακριβέστερο· όλοι οι B, με τα παραπάνω χαρακτηριστικά, παριστάνονται με τον ίδιο τρόπο με τον A. Δηλαδή, αν ήδη δεν μπορείς να ξεχωρίσεις δυο αριθμούς που διαφέρουν λιγότερο από δA, λόγω του τρόπου

που παριστάνονται στον τύπο **float**, δεν έχει νόημα να προσπαθείς να ανιχνεύσεις διαφορά παράστασης μικρότερη από δ*A*. Το *eps* λοιπόν δεν έχει νόημα αν είναι μικρότερο από δ*A* και, επειδή αυτό που έχουμε εύκολα είναι το πάνω φράγμα του, θα πρέπει $eps \geq |A|\varepsilon$.

«Δεν έχει νόημα» ή θα έχουμε πρόβλημα; Μπορεί να έχεις και πρόβλημα! Για παράδειγμα, στις επαναπροσεγγιστικές (iterative) μεθόδους συχνά δουλεύουμε ως εξής: βρίσκουμε ένα διάστημα [*a*, *b*] μέσα στο οποίο βρίσκεται η τιμή ξ που θέλουμε να προσεγγίσουμε και μικραίνουμε το διάστημα έτσι ώστε να γίνει τελικά μηδενικού μήκους, οπότε θα έχουμε *a*=ξ=*b*. Σύμφωνα μεαυτά που είπαμε, θα κάνουμε έλεγχο ως εξής:

```
while ( fabs(b-a) > eps )...
```

αλλά, θα έλεγε κανείς, εδώ πέρα το *eps* ας είναι ό,τι θέλει, ακόμη και μηδέν. Το πολύ-πολύ να γίνουν τα *a* και *b* ίσα, πράγμα όχι άσχημο. Λοιπόν: *δεν είναι έτσι*. Μπορεί (αυτό εξαρτάται από τη μέθοδο) το *b* να γίνει *a*+δ*A*. Από κει και πέρα οι τιμές των *a* και *b* (επιλεγόμενες από το [*a*,*b*]) μπορεί να μην αλλάζουν, ενώ *fabs*(*b*-*a*) = δ*A* > *eps*. Μια τέτοια μέθοδος είναι αυτή της διχοτόμησης και το πρόβλημα φαίνεται στο παρακάτω

Παράδειγμα

Από την §14.3 αντιγράφουμε, με μικρές αλλαγές, το παρακάτω πρόγραμμα, που δοκιμάζει μια συνάρτηση για επίλυση αλγεβρικών εξισώσεων με τη μέθοδο της διχοτόμησης.

```
#include <iostream>
#include <cmath>

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main
```

Δεν έχουμε αλλάξει τίποτε στη *bisection*· αλλά, ενώ στήν §14.3 την είχαμε καλέσει με *epsilon* = 1*e*-5 –και μας έδωσε προσέγγιση της ρίζας 0.1585983– τώρα την καλούμε με *epsilon* = 1e-20. Ο τύπος **double**, στην C++ που δουλεύουμε, είναι αυτός που είδαμε στην §14.7.3, με $\varepsilon = 2^{-23} \approx 2.220446049250e\text{-}16$.

Το πρόγραμμα, κατά την εκτέλεση, πέφτει σε αέναη ανακύκλωση. Ζητώντας εκτύπωση ενδιαμέσων στοιχείων βλέπουμε τα εξής:

```
#Επαν       a           b              |b-a|/2              m
   :        :           :                 :                :
   54  0.15859434  0.15859434  5.5511151231257827e-17  0.15859434
   55  0.15859434  0.15859434  2.7755575615628914e-17  0.15859434
   56  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   57  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   58  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   59  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
    :        :           :                 :                :
```

Δηλαδή, το $m = (a+b)/2$ δεν μπορεί να γίνει καλύτερο και το *fabs*(*b*-*a*)/2 παραμένει μεγαλύτερο από το *epsilon*. Το πρόβλημα ξεκινάει από την τιμή του *epsilon*, που είναι πολύ μικρή για τον τύπο κινητής υποδιαστολής (**double**) που δουλεύουμε. Πράγματι,

$$|a|\varepsilon = 0.15859434 \times 2.220446049250e\text{-}16 \approx 3.521501756864e\text{-}17$$

και $\delta A \in [1.76e\text{-}17, 3.52e\text{-}17]$. Το *epsilon* δεν θα έπρεπε να είναι μικρότερο από δA.[12]

Και τί κάνουμε τώρα; Πώς διορθώνουμε το πρόγραμμα; Μια διόρθωση θα ήταν η εξής: Να βάλουμε ένα

$$trEps = max(|m|\varepsilon/b, epsilon)$$

και να αλλάξουμε τη συνθήκη της while σε `fabs(b-a)/2 >= trEps`. Στην περίπτωση αυτήν όμως η διαδικασία μας εξαρτάται από χαρακτηριστικά του τύπου `float` και χάνει τη δυνατότητα μεταφοράς. Καταφεύγοντας στην καλύτερη μελέτη της μεθόδου, βλέπουμε ότι $m_{νέο} = m_{παλιό} \pm |b - a|/2$. Αν λοιπόν το $|b - a|/2$ γίνει πολύ μικρό –*πρακτικώς μηδέν*– ως προς το m, δεν έχει νόημα να συνεχίζουμε. Πώς ελέγχεται αυτό σε C++; Με τον εξής «περίεργο» τρόπο:

```
m == m + fabs(b-a)/2
```

Εκτός από αυτό, η διαδικασία χρειάζεται και μια άλλη βελτίωση: το πρόγραμμα που την καλεί θα πρέπει να μπορεί να καθορίσει μέγιστο πλήθος επαναλήψεων που θα εκτελεσθούν, άσχετα με το αν πετύχαμε την ακρίβεια που θέλουμε:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   double m, d;

   if ( f(a)*f(b) > 0 )
      errCode = 3;
   else
   {
      int n( 0 );
      do {
         ++n;
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                             else  a = m;
         d = fabs(b - a)/2;
      } while ( (d >= epsilon) && (m != m + d) && (n != nMax) );
      root = m;
      if (d <= epsilon)  errCode = 0;
      else if (m == (m + d))  errCode = 1;
      else if (n == nMax)  errCode = 2;
   } // if
} // bisection
```

Τώρα, που η επαναλήψεις σταματούν για τρεις διαφορετικούς λόγους, θα πρέπει να ελέγξουμε το λόγο τερματισμού και να τον γνωστοποιήσουμε στο πρόγραμμα που κάλεσε τη διαδικασία. Αυτό γίνεται με τη *errCode*:

errCode == 1 σημαίνει: Δεν μπορεί να γίνει άλλη βελτίωση.

errCode == 2 σημαίνει: Ξεπεράσαμε τις *nMax* επαναλήψεις χωρίς να επιτευχθεί η ακρίβεια που ζητήθηκε.

errCode == 3 σημαίνει: Λάθος αρχικό διάστημα.

Και να μια δοκιμή:

```
   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
```

αποτέλεσμα:

[12] Όπως καταλαβαίνεις, τα τελευταία ψηφία των *a*, *b* δεν έχουν νόημα.

```
Λάθος 1
Ρίζα =  0.158594
Λάθος 2
Ρίζα =  0.184375
Λάθος 0
Ρίζα =  0.158594
```

Με την ευκαιρία, πρόσεξε και κάτι άλλο: στον έλεγχο της *n* δεν βάλαμε *n* < *nMax* αλλά *n* != *nMax*· το αποτέλεσμα; Αν η συνάρτηση κληθεί με *nMax* ≤ 0 ο αριθμός των επαναλήψεων δεν ελέγχεται.

Με τον ίδιο τρόπο που γράψαμε το «αν η βελτίωση είναι 0 ως προς το *m*» θα πρέπει να χειρίζεσαι πολλές φορές την περίπτωση: «αν το *A* είναι 0 τότε...». Αντί να γράφεις:

```
if ( fabs(A) <= eps ) ...
```

συχνά είναι προτιμότερο να γράφεις

```
if ( x + A = x ) ...
```

δηλαδή, αν το *A* είναι 0 ως προς το *x*. Φυσικά, το *x* εξαρτάται από το πρόβλημα που έχεις να λύσεις και πρέπει να επιλεγεί προσεκτικά.

17.15 Πρακτικές Συμβουλές

Οι πρώτες δυο συμβουλές για κάποιον που γράφει προγράμματα για επεξεργασία αριθμητικών στοιχείων είναι μάλλον τετριμμένες, αλλά πρέπει να τις πούμε:

- ***Μη χρησιμοποιείς χωρίς λόγο τους τύπους κινητής υποδιαστολής. Προτίμησε ακέραιους τύπους όπου μπορείς. Πρόσεχε, όμως τους πολύ μεγάλους ακέραιους!***
- ***Βελτίωσε τον αλγόριθμό σου ώστε να ελαττώσεις τις πράξεις που κάνεις στο ελάχιστο δυνατό!***

Καλά, θα πεί ο πεπειραμένος προγραμματιστής, για νέο μας το λες; Αυτό ισχύει σε κάθε περίπτωση. Σωστά! Αλλά, αν είναι στόχος να ελαττώνουμε τον χρόνο επεξεργασίας στο ελάχιστο για κάθε πρόγραμμα, για τα αριθμητικά προγράμματα έχουμε ένα λόγο ακόμη: *λιγότερες πράξεις σημαίνει και πιο ακριβές αποτέλεσμα*.

- ***Πρόσεξε τις προσθέσεις και τις αφαιρέσεις! Αν νομίζεις ότι μπορεί να σου δημιουργήσουν προβλήματα προσπάθησε να τις αποφύγεις!***

Να τις αποφύγω; Αν πρέπει να τις κάνω, πως να τις αποφύγω; Δυο παραδείγματα σου δίνουν μια ιδέα. Και τα δυο προγράμματα εκτελέστηκαν σε τύπο `float` 32 ψηφίων.

Παράδειγμα 1

Θέλουμε να λύσουμε την εξίσωση: $ax^2 + bx + c = 0$ όπου $a = 1.0$, $b = 1000000000.0 = 10^9$, $c = 4.0$ και φυσικά χρησιμοποιούμε τους γνωστούς μας τύπους:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ο τύπος `float` δουλεύει όπως περιγράψαμε στην §14.6.2. Όταν υπολογίζουμε το υπόριζο $b^2 = 10^{18}$, ενώ $4ac = 16$. Η αφαίρεση στον υπολογιστή μας θα δώσει αποτέλεσμα 10^{18} και τελικά θα έχουμε $x_+ =$ -1000000000.0 και $x_- = 0$. Η πρώτη ρίζα προσεγγίστηκε θαυμάσια. Στη δεύτερη, η προσέγγιση $\sqrt{b^2 - 4ac} \approx b$ και η πράξη $-b + b$ (υποχείλιση) που ακολούθησε, μας έδωσε ένα τελείως λάθεμένο αποτέλεσμα. Τον υπολογισμό του υπορίζου δεν μπορούμε να τον βελτιώσουμε· ας τον χρησιμοποιήσουμε λοιπόν μόνο για την πρώτη ρίζα και ας αποφύγουμε την αφαίρεση. Θα στηριχτούμε στην ιδιότητα: $x_+x_- = c/a$, από όπου παίρνουμε: $x_- = c/(ax_+)$.

Το παρακάτω πρόγραμμα υπολογίζει το x_+(x1) και στη συνέχεια το x_- (x2) με τον γνωστό τύπο. Στη συνέχεια υπολογίζει τη δεύτερη ρίζα ως x3 χωρίς να κάνει την αφαίρεση.

```
int main()
{
   float a, b, c, x1, x2, x3;

   a = 1.0f;  b = 1000000000.0f;  c = 4.0f;
   x1 = (-b - sqrt(b*b - 4*a*c))/(2*a);
   x2 = (-b + sqrt(b*b - 4*a*c))/(2*a);
   x3 = c/(a*x1);
   cout << x1 << "  " << x2 << "  " << x3 << endl;
}
```

Να το αποτέλεσμα:

```
-1e+009  0  -4e-009
```

Παρατηρήσεις ▶

Για πρώτη φορά αναφέραμε ότι «*η αφαίρεση είναι μια «κακή» πράξη για τους τύπους κινητής υποδιαστολής διότι μπορεί να προκαλέσει απώλεια σημαντικών ψηφίων*» στην παρατήρηση 3 του Παραδ. 2 της §5.5. Μπορείς να κάνεις αυτά που λέμε παραπάνω με τα προγράμματα που είδαμε εκεί αφού αλλάξεις τον `double` σε `float`.◀

Παράδειγμα 2

Θέλουμε να υπολογίσουμε την τιμή της παράστασης 1 - *συνt* για πολύ μικρά *t*. Αν δεν προσέξουμε θα πάρουμε 0 (υποχείλιση). Ας αποφύγουμε την αφαίρεση:

$$1-\sigma\upsilon\nu t = (1-\sigma\upsilon\nu t)\frac{1+\sigma\upsilon\nu t}{1+\sigma\upsilon\nu t} = \frac{\eta\mu^2 t}{1+\sigma\upsilon\nu t}$$

Και δοκιμάζουμε τους δυο τρόπους:

```
   t = 0.0000000001;
   x1 = 1 - cos(t);
   cout << x1 << endl;
   x2 = sin(t)*sin(t)/(1+cos(t));
   cout << x2 << endl;
```

αποτέλεσμα:

```
0
5e-021
```

Δηλαδή, με την ασφαλέστερη "1+*συνt*" αποφύγαμε την «καταστροφική» "1-*συνt*".

- ***Απόφυγε την υπερχείλιση χωρίς λόγο.***

Όπως είδαμε πιο πριν, για τους τύπους κινητής υποδιαστολής, δεν ισχύουν η προσεταιριστικότητα της πρόσθεσης και του πολλαπλασιασμού, όπως και η επιμεριστικότητα των δυο πράξεων. Είναι πολύ πιθανό, με μια αναδιάταξη πράξεων σε μια αριθμητική παράσταση, να αποφύγεις μια υπερχείλιση από ενδιάμεσες πράξεις.

Ας δούμε δυο παραδείγματα:

Παράδειγμα 3

Από τα πρώτα πράγματα που μαθαίνει κανείς στον προγραμματισμό είναι να μην διαιρεί δια 0. Έτσι, η εντολή:

```
z = x/y;
```

αντικαθίσταται πολλές φορές από κάτι σαν:

```
if ( y != 0 )  z = x/y;
         else  ...
```

Όπως είπαμε πιο πάνω, κάτι τέτοιο δεν έχει και πολύ νόημα. Πιο σωστό θα ήταν κάτι σαν:

```
if ( fabs(y) > eps ) ...
```

Πράγματι, αν η τιμή του *y* έχει προκύψει μετά από πολλές πράξεις στον τύπο `float`, είναι απίθανο να είναι 0 όταν την περιμένουμε. Στην περίπτωση αυτήν ένα «απρόσεκτο» πρόγραμμα μπορεί να υποστεί τις συνέπειες μιας «κρυφής υπερχείλισης». Δηλαδή, το *y* δεν είναι ακριβώς μηδέν, αλλά έχει μια πολύ μικρή τιμή χωρίς κανένα νόημα. Το αποτέλεσμα θα είναι, πολλές φορές, να μην έχουμε υπερχείλιση αλλά το *z* να πάρει μια απίθανη τιμή που αλλοιώνει όλους τους υπολογισμούς στη συνέχεια.

Παράδειγμα 4

Παρόμοιες προφυλάξεις μπορούμε να πάρουμε και στην πρόσθεση. Ας υποθέσουμε ότι οι *x* και *y* μπορούν να πάρουν μεγάλες θετικές τιμές και η εντολή:

```
z = x + y;
```

θα μπορούσε να οδηγήσει σε υπερχείλιση. Θα μπορούσαμε και εδώ να προλάβουμε μια τέτοια πρόσθεση. Όπως ξέρουμε η C++ μας δίνει για τον τύπο `float` τη σταθερά *FLT_MAX* που είναι η μέγιστη τιμή που μπορεί να παρασταθεί στον τύπο `float`[13]. Μπορούμε λοιπόν, όπου χρειάζεται, να αντικαταστήσουμε την εντολή εκχώρησης με κλήση κάποιας διαδικασίας *addFloat*, που γράφεται όπως η *addInt* που γράψαμε για τον τύπο `int`.

Φυσικά, παρόμοιες προφυλάξεις μπορούμε να πάρουμε για όλες τις πράξεις και να έχουμε το κεφάλι μας ήσυχο. Ε, όχι κι έτσι! Ένα τέτοιο πρόγραμμα θα ήταν *απαράδεκτο*. Οι περισσότεροι από τους ελέγχους, που είναι *χρονοβόροι*, θα ήταν άχρηστοι. Ο *αμυντικός προγραμματισμός* σε σχέση με τις αριθμητικές πράξεις μπορεί να χρησιμοποιείται, αλλά μόνον όπου χρειάζεται.

Η τελευταία μας συμβουλή δεν μπορεί να είναι άλλη:

- ***Διάβασε Αριθμητική Ανάλυση.***

Με όσα είπαμε στις προηγούμενες παραγράφους, δεν φιλοδοξούμε να καλύψουμε τα θέματα που παρουσιάσαμε, αλλά περισσότερο να σου δείξουμε τα προβλήματα: *Τα αριθμητικά προγράμματα είναι δύσκολα. Η σύνθεσή τους απαιτεί* ***γνώσεις*** *και πολλή προσοχή.* Τις περισσότερες από τις γνώσεις που θα σου χρειαστούν θα σου τις δώσει η Αριθμητική Ανάλυση.

Ασκήσεις

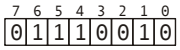
Α Ομάδα

17-1 (*γενίκευση της count*1) Γράψε περίγραμμα συνάρτησης *xCount*1 που θα τροφοδοτείται με τιμή *x*, ακέραιου τύπου και δύο φυσικούς $k_1 \le k_2$ και θα υπολογίζει και θα επιστρέφει το πλήθος των δυαδικών ψηφίων που έχουν τιμή 1 στις θέσεις από k_1 μέχρι k_2 της *x*.

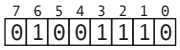
17-2 (*κυκλική ολίσθηση*) Γράψε συνάρτηση, με το όνομα *rShiftRight*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με μια τιμή *x*, τύπου `unsigned char`, και μια μη αρνητική ακέραιη τιμή *p* και θα υπολογίζει και θα επιστρέφει την *κυκλική* ολίσθηση δεξιά της *x* κατά *p* θέσεις (τα *p* δυαδικά ψηφία που χάνονται προς τα δεξιά εμφανίζονται από τα αριστερά, αντί μηδενικών, με την ίδια σειρά), όπως φαίνεται στο παρακάτω παράδειγμα. Αν η συνάρτησή μας κληθεί με το *x* που φαίνεται και *p* = 3 παίρνουμε το αποτέλεσμα που βλέπεις:

---

[13] Παρομοίως υπάρχουν και οι *DBL_MAX*, *LDBL_MAX* για τους τύπους `double` και `long double` αντίστοιχως.



x



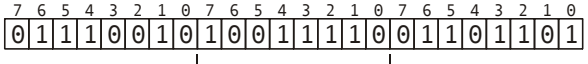
αποτέλεσμα

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *rShiftLeft*, για κυκλική ολίσθηση αριστερά.

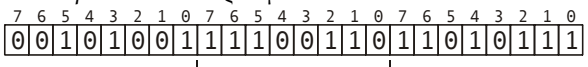
Β Ομάδα

17-3 (*ολίσθηση σε πίνακα*) Γράψε συνάρτηση, με το όνομα *shiftLeft*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου `unsigned char`, το πλήθος των στοιχείων του *n* και μια μη αρνητική ακέραιη τιμή *p* και θα κάνει ολίσθηση αριστερά κατά *p* θέσεις των δυαδικών ψηφίων *ολόκληρου του πίνακα*, όπως φαίνεται στο παρακάτω παράδειγμα. Έχουμε αρχικώς (*n* = 3):



a[2] a[1] a[0]

και ζητείται ολίσθηση κατά *p* = 4. Θα πάρουμε:



a[2] a[1] a[0]

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *shiftRight*, για ολίσθηση πίνακα δεξιά.

17-4 Γενίκευσε τις *bitValue()*, *setBit()*, *clearBit()*, *count1()* και *part()* για πίνακα ακέραιου τύπου *T*. Αν κάθε τιμή τύπου *T* αποθηκεύεται σε s_T ψηφιολέξεις τότε η παράμετρος *p* των τριών πρώτων θα μπορεί να παίρνει τιμές από 0 μέχρι $n \times s_T - 1$, όπου *n* το πλήθος των στοιχείων του πίνακα.

17-5 Ακολουθώντας το παράδειγμα που δώσαμε στην §17.3 για την πρόσθεση, γράψε διαδικασίες για ασφαλείς πράξεις: αφαίρεση (*subtrInt*), πολλαπλασιασμό (*multInt*), διαίρεση (*divInt*) στον τύπο `int`.

17-6 Γράψε διαδικασίες για ασφαλείς πράξεις για τον τύπο `float`.

Γ Ομάδα

17-7 Αν *x* τιμή τύπου `float`, ορίζουμε:

$$\varepsilon_x = min\{\, u: \mathbb{R} \mid u > 0 \wedge (|x|+u)_f \neq |x|_f \bullet u \,\}$$

Μπορείς να βρεις σχέση του ε_x με το ε; Πόσο σωστό είναι το $\delta x = x \cdot \varepsilon$;

[Απάντηση: $x \cdot \varepsilon$ /βάση $< \varepsilon_x \leq x \cdot \varepsilon$]

κεφάλαιο

18

Προετοιμάζοντας Βιβλιοθήκες

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να κάνεις (στατικές) βιβλιοθήκες συναρτήσεων

- είτε διότι το ζήτησε κάποιος πελάτης (ή εργοδότης)
- είτε για να κεφαλαιοποιήσεις την προγραμματιστική δουλειά που κάνεις, ώστε να μη χρειάζεται να ανακαλύπτεις τον τροχό κάθε τόσο.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράψεις και να χρησιμοποιήσεις στατικές βιβλιοθήκες συναρτήσεων και περιγραμμάτων.

Έννοιες κλειδιά:

- *οδηγία* `define`
- *οδηγία* `ifdef`
- *βιβλιοθήκη περιγραμμάτων συναρτήσεων*
- *στατική βιβλιοθήκη*
- *χωριστή μεταγλώττιση*
- `namespace`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Αρχίζοντας, να τονίσουμε ότι οι προγραμματιστικές βιβλιοθήκες με τις οποίες θα ασχοληθούμε είναι *βιβλιοθήκες συναρτήσεων* και όχι εκτελέσιμων προγραμμάτων (υπάρχουν και τέτοιες). Οι πρώτες εμφανίστηκαν σχετικώς νωρίς με πιο γνωστές –τα πρώτα χρόνια– τις βιβλιοθήκες της FORTRAN.

Οι συναρτήσεις των βιβλιοθηκών *συνδέονται* στα προγράμματα που αναπτύσσουν οι προγραμματιστές απαλλάσσοντάς τους έτσι από το να «ξαναεφευρίσκουν τον τροχό»!

Οι προγραμματιστές αναπτύσσουν βιβλιοθήκες συναρτήσεων «κατά παραγγελία». Αλλά πολλές βιβλιοθήκες προκύπτουν και ως παραπροϊόντα της ανάπτυξης προγραμμάτων: ο προγραμματιστής, κρίνοντας από την εμπειρία του, ότι κάποιες συναρτήσεις έχουν γενι-

κότερη χρησιμότητα και όχι μόνο για το πρόγραμμα που γράφηκαν τις εντάσσει σε βιβλιοθήκες ώστε να μπορεί να τις ξαναχρησιμοποιήσει.

Μια βιβλιοθήκη, σε σχέση με τον τρόπο που χειρίζεται τις συνιστώσες της ο **συνδέτης** (linker), μπορεί να είναι **στατική** (static) ή **δυναμικής σύνδεσης** (dynamic linking):

- *Στατικές Βιβλιοθήκες*: Μετά τη μεταγλώττιση του (αρχικού) προγράμματος ο συνδέτης εντάσσει σε αυτό τις συναρτήσεις που καλούνται από τη βιβλιοθήκη δημιουργώντας το τελικό εκτελέσιμο πρόγραμμα που είναι αυθύπαρκτο, μπορεί δηλαδή να εκτελεσθεί χωρίς την παρουσία της βιβλιοθήκης.
- *Βιβλιοθήκες Δυναμικής Σύνδεσης*: Τέτοιες είναι οι *DLL* (Dynamic Link Library) των Windows και οι *DSO* (Dynamic Shared Object) των Unix, Linux. Στη φάση της δημιουργίας του εκτελέσιμου προγράμματος ο συνδέτης δεν αντιγράφει σε αυτό τις συναρτήσεις που καλούνται· απλώς καταγράφει τι καλείται και σε ποιο σημείο. *Η τελική φάση της σύνδεσης γίνεται κάθε φορά που αρχίζει η εκτέλεση του προγράμματος*: ο συνδέτης αναζητεί τις δυναμικές βιβλιοθήκες, τις φορτώνει στη μνήμη και κάνει την τελική σύνδεση. *Αν κάποια βιβλιοθήκη είναι ήδη φορτωμένη, επειδή τη ζήτησε άλλο πρόγραμμα, δεν ξαναφορτώνεται*. Η δυναμική βιβλιοθήκη σβήνεται από τη μνήμη όταν δεν εκτελείται πρόγραμμα που να έχει συνδεθεί με αυτήν. Ένα (εκτελέσιμο) πρόγραμμα που χρησιμοποιεί βιβλιοθήκες δυναμικής σύνδεσης *δεν είναι αυθύπαρκτο*. Όταν το μεταφέρεις από τη μια εγκατάσταση στην άλλη θα πρέπει να φροντίσεις να υπάρχουν –στη νέα εγκατάσταση– και οι απαιτούμενες βιβλιοθήκες.

Πέρα από τις βιβλιοθήκες συναρτήσεων υπάρχουν βιβλιοθήκες περιγραμμάτων (συναρτήσεων ή κλάσεων), κλάσεων κλπ.

Εδώ θα ασχοληθούμε με

- στατικές βιβλιοθήκες συναρτήσεων και
- βιβλιοθήκες περιγραμμάτων συναρτήσεων.

Πριν ξεκινήσουμε όμως θα πρέπει να δούμε μερικές «οδηγίες προς τον προεπεξεργαστή» που θα μας είναι απαραίτητες από εδώ και πέρα: τις **define** και **undef** καθώς και τις **ifdef** και **ifndef**.

> *ΠΡΟΣΟΧΗ! Σε αυτό το κεφάλαιο, για να δείξουμε ορισμένα πράγματα, θα πρέπει να χρησιμοποιήσουμε διαδικασίες που εξαρτώνται από το περιβάλλον ανάπτυξης και το ΛΣ. Εδώ θα στηριχτούμε στον Dev-C++, σε περιβάλλον Windows. Εσύ θα πρέπει, πιθανότατα, να ψάξεις τα εγχειρίδια (ή το Help) της C++ που χρησιμοποιείς.*

18.1 Οι Οδηγίες "`define`", "`ifdef`" κλπ

Στην §1.5 είδαμε την οδηγία προς τον προεπεξεργαστή "`include`" και από τότε τη χρησιμοποιούμε σε όλα τα προγράμματά μας.

Στην επόμενη παράγαρφο –αλλά και στα επόμενα κεφάλαια– θα χρειαστούμε μερικές οδηγίες ακόμη· τις βλέπουμε στη παρούσα παράγραφο.

Η οδηγία "`define`" (όρισε) έχει συντακτικό:

"`#`", "`define`", *όνομα*, *λίστα λεξικών οντοτήτων*

και ορίζει μια **μακροεντολή** ή **μακροσυνάρτηση** (macro). Το *όνομα* είναι το **όνομα της μακροεντολής** (macro identifier), ενώ η *λίστα λεξικών οντοτήτων* είναι το **σώμα της μακροεντολής** (macro body).

Αποτέλεσμα της οδηγίας είναι να αντικατασταθεί, στις γραμμές που ακολουθούν, το *όνομα* της μακροεντολής από αυτά που ορίζει η *λίστα λεξικών οντοτήτων*. Αυτή η διαδικασία λέγεται **μακροανάπτυξη** (macro expansion).

Η οδηγία "`define`" αναιρείται από μια οδηγία "`undef`"(ine):

"`#`", "`undef`", *όνομα*
με το ίδιο όνομα μακροεντολής.

Μετά από οδηγία "`undef`" το όνομα της μακροεντολής είναι **αόριστο** (undefined), όπως και πριν από την οδηγία "`define`". Μετά την define είναι **ορισμένο** (defined).

Παράδειγμα

Η οδηγία:

```
#define NMAX 50
```

ζητάει να αντικατασταθεί η λέξη *NMAX* με το "`50`" οπουδήποτε βρεθεί στη συνέχεια μέχρι να βρεθεί οδηγία "`#undef NMAX`".

Στο παρακάτω κομμάτι προγράμματος:

```
#define NMAX 50

int main()
{
   int a[NMAX];
      :
   for (int k(0); k <= NMAX-1; ++k)
   {
#undef NMAX
       swap(a[k], a[NMAX-k]);
   } // for
      :
```

η αντικατάσταση της *NMAX* από την τιμή 50 γίνεται μέχρι και τη γραμμή με την εντολή **for** ενώ δεν γίνεται στη γραμμή με την εντολή `swap(a[k], a[NMAX-k])` διότι προηγουμένως μεσολαβεί η οδηγία "`#undef NMAX`".Έτσι, ο μεταγλωττιστής θα μας βγάλει το μήνυμα "`Undefined symbol 'NMAX'`" για τη γραμμή αυτή.

Το *NMAX* είναι *ορισμένο* στις γραμμές του προγράμματος μεταξύ των οδηγιών "`#define NMAX 50`" και "`#undef NMAX`" ενώ είναι *αόριστο* πριν από την πρώτη και μετά τη δεύτερη.

Οι αντικαταστάσεις μπορεί να είναι και πιο πολύπλοκες:

Παράδειγμα

Αντιγράφουμε από το `stdlib.h`:

```
#define max(a,b)    (((a) > (b)) ? (a) : (b))
```

Αυτή η οδηγία θα έχει ως αποτέλεσμα μια εντολή σαν την:

```
      y = x + max(x+1, y-1);
```

να γίνει:

```
      y = x + (((x+1) > (y-1)) ? (x+1) : (y-1));
```

Η μακροανάπτυξη *δεν γίνεται* όταν το όνομα της μακροεντολής ευρεθεί:

- μέσα σε ορμαθό χαρακτήρων ή σε σχόλιο,
- μέσα στην ίδια τη μακροεντολή (π.χ. η "`#define A A`" δεν θα έχει ως αποτέλεσμα την επ' άπειρο εκτέλεσή της)

Φυσικά, μακροανάπτυξη *δεν γίνεται* και μέσα σε μια οδηγία "`undef`". Δηλαδή, για παράδειγμα, η

```
#define NMAX 50
```

δεν θα κάνει την

```
#undef NMAX
```

που ακολουθεί "`#undef 50`".

Και τι κερδίσαμε; Απλές συναρτήσεις, που καλούνται πολύ συχνά δεν «φορτώνουν» τον χρόνο εκτέλεσης του προγράμματος με το κόστος κλήσης συνάρτησης. Αυτό το πρόβλημα όμως το αντιμετωπίσαμε ήδη στην §14.1 με τις συναρτήσεις "`inline`". Έτσι, όσα είπαμε μέχρι εδώ για τη "`define`" σκοπό έχουν να την αναγνωρίζεις και να καταλαβαίνεις τη χρήση της σε άλλα προγράμματα. Στη C++

- η δήλωση σταθερών με "`const`" και
- οι συναρτήσεις "`inline`"

λύνουν τα αντίστοιχα προβλήματα με πολύ καλύτερο τρόπο και αυτά τα εργαλεία θα χρησιμοποιούμε.

Τώρα θα δούμε μια «περίεργη» μορφή της "`define`" την οποία και θα χρησιμοποιήσουμε στη συνέχεια.

Αν δεν υπάρχει σώμα της μακροεντολής τότε ο προεπεξεργαστής θα διαγράψει το όνομά της όπου το βρει. Π.χ. το:

```
#define NMAX
int main()
{
   int a[NMAX];

   for (int k(0); k <= NMAX-1; ++k)
   {
      swap(a[k], a[NMAX-k]);
   }
```

θα γίνει:

```
#define NMAX
int main()
{
   int a[];

   for (int k(0); k <= -1; ++k)
   {
      swap(a[k], a[-k]);
   }
```

Μια τέτοια χρήση της "`define`" κάνουμε όταν θέλουμε να έχουμε *ορισμένο* κάποιο όνομα σε μια περιοχή του προγράμματός μας, όπως θα δούμε στη συνέχεια.

Πολύ συχνά, θέλουμε να μεταγλωττίσουμε (ή να μη μεταγλωττίσουμε) ένα κομμάτι προγράμματος με κριτήριο το αν έχει ορισθεί ή δεν έχει ορισθεί κάποιο σύμβολο χωρίς να μας ενδιαφέρει η τιμή που μπορεί να του δόθηκε.

Ας πούμε ότι, σε κάποιο πρόγραμμα, αν έχει ορισθεί το σύμβολο *N* θέλουμε να έχουμε ως συνάρτηση *f* την

```
int f( int x )
{  ++x;  return x;  }
```

αλλιώς, αν δεν έχει ορισθεί το *N*, θέλουμε ως *f* την

```
int f( int x )
{  return x;  }
```

Αυτό γράφεται ως εξής:[1]

```
#ifdef N

int f( int x )
{  ++x;  return x;  }

#endif
```

[1] Υπάρχει και οδηγία "`#else`" και θα μπορούσαμε να το γράψουμε διαφορετικά αλλά ας την αφήσουμε...

```
#ifndef N

int f( int x )
{  return x;  }

#endif
```

Η "`#ifdef N`" είναι ο τρόπος που μας δίνει η C++ για να διατυπώσουμε την οδηγία: «`#if έχει ορισθεί το N`».

Για να διατυπώσουμε την "`#if δεν έχει ορισθεί το N`" θα πρέπει να γράψουμε: "`#ifndef N`".

Η "`#endif`" σημειώνει το τέλος αυτών που ισχύουν για την αντίστοιχη **if** ("`#ifdef N`" ή "`#ifndef N`").

Παράδειγμα

Αντιγράφουμε από το `iostream` (gcc – Dev-C++):

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

// . . . όλο το περιεχόμενο του αρχείου

#endif /* _GLIBCXX_IOSTREAM */
```

Ας δούμε πρώτα το πρόβλημα και μετά το νόημα των παραπάνω. Ας πούμε ότι έχεις ένα αρχείο, το `myFuncs.h`, με κάποιες συναρτήσεις. Μέσα στο αρχείο έχεις βάλει τη οδηγία:

```
#include <iostream>
```

Στο αρχείο `myprog.cpp`, που περιέχει ένα πρόγραμμα που γράφεις, βάζεις στην αρχή:

```
#include <iostream>
#include "myFuncs.h"
```

Τι θα μπορούσε να συμβεί στην περίπτωση αυτή; Οι δηλώσεις και οι ορισμοί του `iostream` θα υπάρχουν στο πρόγραμμά σου δύο φορές και αυτό δεν θα γίνει δεκτό από τον μεταγλωττιστή.

Το `iostream` αμύνεται, με τις οδηγίες που είδαμε πιο πάνω, ως εξής:

- Όταν εκτελεσθεί η "`#include <iostream>`" του `myprog.cpp` το σύμβολο "`_GLIBCXX_IOSTREAM`" δεν είναι ορισμένο. Έτσι, λόγω της
  `#ifndef _GLIBCXX_IOSTREAM`
  περικλείεται στο πρόγραμμά μας ολόκληρο το περιεχόμενο του `iostream`. Πρώτη και καλύτερη περικλείεται και εκτελείται η οδηγία
  `#define _GLIBCXX_IOSTREAM 1`
  Τώρα πια το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο.[2]
- Στη συνέχεια, όταν περιληφθεί το `myFuncs.h`, λόγω της οδηγίας
  `#include <iostream>`
  που υπάρχει εκεί, περιλαμβάνεται για δεύτερη φορά το `iostream`. Τώρα όμως, όταν εκτελείται η
  `#ifndef _GLIBCXX_IOSTREAM`
  το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο και δεν περιλαμβάνεται οτιδήποτε υπάρχει μέχρι και την
  `#endif /* _GLIBCXX_IOSTREAM */`

Αυτό θα πρέπει να κάνουμε και εμείς στα αρχεία με ορισμούς και δηλώσεις που θέλουμε να χρησιμοποιούμε συχνά. Για παράδειγμα, στο `myFuncs.h` θα πρέπει να περιλάβουμε όλο το περιεχόμενο μεταξύ των οδηγιών:

```
#ifndef _MYFUNCS_H
#define _MYFUNCS_H
```

[2] Εκείνο το "1" στη **define** δεν ειναι απαραίτητο, κατ' αρχήν.

```
// . . . όλο το περιεχόμενο του αρχείου

#endif  /* _MYFUNCS_H */
```

18.2 Μια Βιβλιοθήκη Περιγραμμάτων Συναρτήσεων

Το πιο απλό είδος βιβλιοθήκης είναι μια βιβλιοθήκη περιγραμμάτων συναρτήσεων. Θα κάνουμε λοιπόν και εμείς μια μικρή βιβλιοθήκη –τη *MyTmpltLib*– που θα περιέχει περιγράμματα που ήδη έχουμε χρησιμοποιήσει:

- Το περίγραμμα συνάρτησης *linSearch()*, που είδαμε στην §16.13.1, παρ' όλο που είπαμε ότι στις βιβλιοθήκες της C++ υπάρχει η *find()*.
- Το *renew()*, (§16.12), που μας είναι χρήσιμο μέχρι να μάθουμε τα περιγράμματα περιεχόντων (containers) της C++.
- Τα *new2d()* και *delete2d()* που είδαμε στην §16.9.

Σε ένα αρχείο, ας το πούμε `MyTmpltLib.h`, αντιγράφουμε τα τέσσερα περιγράμματα και πριν από όλα την κλάση εξαιρέσεων *MyTmpltLibXptn* (§16.9):

```
#ifndef _MYTMPLLIB_H
#define _MYTMPLLIB_H

#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// ΟΠΩΣ ΣΤΗΝ §16.13.1

template< typename T >
void renew( T*& p, int ni, int nf )
// ΟΠΩΣ ΣΤΗΝ §16.12

template< typename T >
T** new2d( int nR, int nC )
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
void delete2d( T**& a, int nR )
// ΟΠΩΣ ΣΤΗΝ §16.9

#endif // _MYTMPLLIB_H
```

Τα πάντα περιέχονται ανάμεσα στις `#ifndef _MYTMPLLIB_H` και `#endif`. Πριν από όλα η `#define _MYTMPLLIB_H`. Με αυτές αμυνόμαστε για περίπτωση πολλαπλής `#include "MyTmpltLib.h"`.

Ακόμη:

- Η `#include <string>` μας είναι απαραίτητη για τη *strncpy()*.
- Η `#include <new>` μας είναι απαράιτητη για τη *bad_alloc*.
- Η using `namespace std` μας είναι απαραίτητη για να μη γράφουμε `std::strncpy`, `std::bad_alloc`.

Να δούμε τώρα πώς τη χρησιμοποιούμε. Το πρόγραμμα της §16.12 γίνεται:

```
#include <iostream>
```

```
#include <new>
#include "MyTmpltLib.h"

using namespace std;

int main()
{
   int*    ip;
   double* dp;
// . . .
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
// . . .
} // main
```

Δηλαδή: υπάρχει η `#include "MyTmpltLib.h"` και δεν υπάρχει το περίγραμμα *renew()*. Από ολόκληρο το `MyTmpltLib.h` χρησιμοποιείται μόνο το *renew* από το οποίο δημιουργούνται δύο περιπτώσεις: μια για `int` (λογω της `renew(ip, 3, 7)`) και μια για `double` (λογω της `renew(dp, 3, 7)`).

Στο πρόγραμμα του Παράδ. 2 της §16.13 θα έχουμε:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>
#include "MyTmpltLib.h"

using namespace std;

struct ApplicXptn
// . . .
```

Και εδώ βάλαμε την `#include "MyTmpltLib.h"` και αφαιρέσαμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *renew()* και *linSearch()*. Από το *renew()* θα πάρουμε ένα στιγμιότυπο για τον τύπο *GrElmn* και από τη *linSearch()* ένα στιγμιότυπο για τον ίδιο τύπο.

Στο πρόγραμμα πολλαπλασιαμού πινάκων (§16.9) θα έχουμε:

```
#include <iostream>
#include <fstream>
#include "MyTmpltLib.h"

using namespace std;

void input2DAr( istream& tin, int** a, int nRow, int nCol );
void output2DAr( ostream& tout, int** a, int nRow, int nCol );

int main()
// . . .
```

Και πάλι βάζουμε την `#include "MyTmpltLib.h"` και αφαιρούμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *new2d* και *delete2d*. Παίρνουμε μια περίπτωση για το κάθε ένα για τον τύπο `int`.

Ως προς τη σύνδεση: τι είδους βιβλιοθήκη κάναμε; Στατική ή δυναμικής σύνδεσης; Παρ' όλο που οι όροι αυτοί αναφέρονται σε βιβλιοθήκες με μεταγλωττισμένο περιεχόμενο, μπορούμε να πούμε ότι έχουμε μια βιβλιοθήκη *στατική*· ο συνδέτης καλείται μόνο μια φορά να δώσει ένα αυθύπαρκτο εκτελέσιμο πρόγραμμα.

Όπως βλέπεις, τα πράγματα είναι πολύ απλά. Βέβαια, αν δεις τα πράγματα από εμπορική άποψη υπάρχει ένα προβληματάκι: Αν θέλεις να πουλήσεις προγράμματα σε κάποιον πελάτη του δίνεις υποχρεωτικώς το αρχικό πρόγραμμα, σε C++. Αλλά το αρχικό πρόγραμμα, συνήθως, τιμολογείται πολύ ακριβά. Είναι δυνατόν να του δίνεις μεταγλωττισμένες συναρτήσεις μόνο; Αυτό βλέπουμε στη συνέχεια.

18.3 Χωριστή Μεταγλώττιση

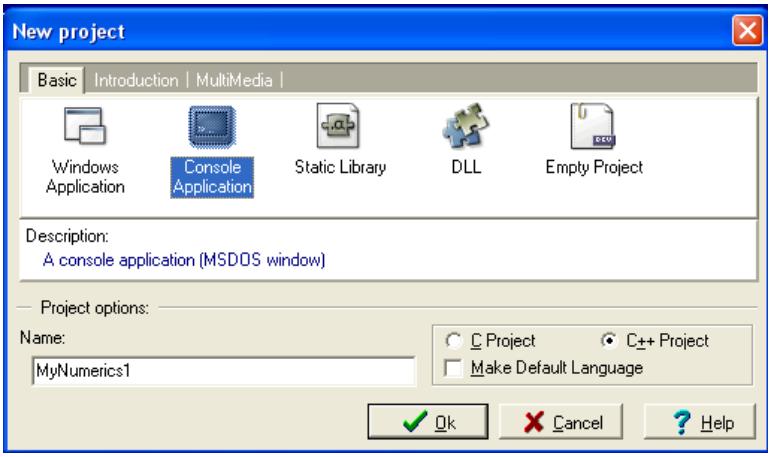
Ο επόμενος στόχος μας είναι να δημιουργήσουμε μια ακόμη πιο μικρή βιβλιοθήκη με τους δύο αριθμητικούς αλγόριθμους που έχουμε μεταφράσει σε πρόγραμμα: τον *αλγόριθμο του Horner* (§9.4, Παράδ. 1) και τον αλγόριθμο της διχοτόμησης (bisection, §17.14).

Ας ξεκινήσουμε ως εξής: Βάζουμε τις δύο συναρτήσεις σε ένα αρχείο με όνομα `MyNumerics.cpp`. Στο αρχείο `MyNumerics.h` βάζουμε τα εξής:

```
#ifndef _MYNUMERICS_H
#define _MYNUMERICS_H

#include <string>

using namespace std;

struct MyNumericsXptn
{
   enum {  domainError, noArray  };

   char   funcName[100];
   int    errCode;
   double errDblVal;
   MyNumericsXptn( const char* fn, int ec, double ev = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errDblVal = ev;  }
}; // MyNumericsXptn

// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x );

// bisection -- προσεγγίζει λύση (root) της εξίσωσης f(x) = 0
//              στο διάστημα [α,β] με τη μέθοδο της διχοτόμησης
//              με nMax επαναλήψεις το πολύ.
//              Στο [a,b] η f πρέπει να είναι συνεχής και
//              f(a)*f(b) <= 0
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );

#endif // _MYNUMERICS_H
```

Στο `MyNumerics.cpp` βάζουμε:

```
#include <string>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNumericsXptn( "ph", MyNumericsXptn::noArray );
   if ( m < 0 )
      throw MyNumericsXptn( "ph",
                            MyNumericsXptn::domainError, m );
// τα υπόλοιπα όπως ήταν

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNumericsXptn( "bisection",
```



Σχ. 18-1 Ο πίνακας επιλογής "New Project" του περιβάλλοντος ανάπτυξης Dev C++.

```
                                MyNumericsXptn::domainError, epsilon );
// τα υπόλοιπα όπως ήταν
```

Παρατηρήσεις: ▶

1. Όπως βλέπεις, τώρα οι συναρτήσεις μας ρίχνουν και εξαιρέσεις. Όταν έχεις να μετατρέψεις μια συνάρτηση που την έχεις αναπτύξει για μια συγκεκριμένη εφαρμογή, όπου η χρήση της είναι –πιθανότατα– ελεγχόμενη, σε συνάρτηση «γενικής χρήσης» θα πρέπει να κάνεις κάτι τέτοιες αλλαγές.
2. Η *bisection()* ρίχνει εξαίρεση αν κληθεί με *epsilon* < 0. Τα υπόλοιπα προβλήματα αντιμετωπίζονται με επιστρεφόμενο κωδικό λάθους.◀

Φυλάγουμε αυτά τα αρχεία στο ευρετήριο **MyNumerics1**.

Στο περιβάλλον ανάπτυξης της Dev-C++ επιλέγουμε:

File|New|Project

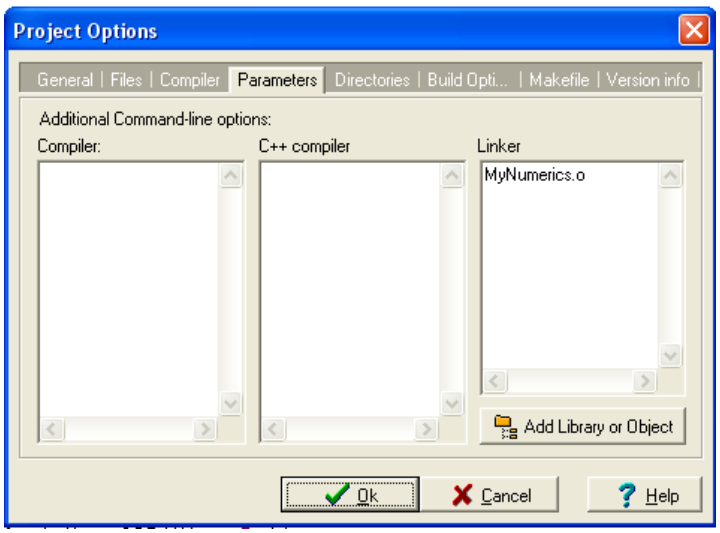
και βλέπουμε τον πίνακα επιλογής που βλέπεις στο Σχ. 18-1. Στο "**Name:**" γράφουμε **MyNumerics1** και επιλέγουμε ως είδος project "**Console Application**". Δίνοντας "**Ok**" μας ζητείται να επιλέξουμε πού θα αποθηκευτεί. Επιλέγουμε το ευρετήριο **MyNumerics1**. Στο **MyNumerics1**, δημιουργείται ένα αρχείο με όνομα **MyNumerics1.dev**. Αυτό είναι το αρχείο του project.

Στο περιβάλλον ανάπτυξης (επιλογή "**Project**" στο αριστερό παράθυρο) βλέπεις ότι έχεις ανοιγμένο το project **MyNumerics1**, που έχει ως περιεχόμενο το αρχείο **main.cpp**· το βλέπεις στο δεξιό παράθυρο:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Τώρα, επιλέγοντας **Project|Add to Project** ζητούμε να προστεθεί στο project το αρχείο **MyNumerics.cpp**. Βλέπουμε το όνομά του να εμφανίζεται στο αριστερό παράθυρο. Τέλος, επιλέγουμε **Execute|Rebuild all** και βλέπουμε να εμφανίζονται στο **MyNumerics1** τα εξής αρχεία:

```
main.o
Makefile.win
MyNumerics.o
MyNumerics1.exe
```



Σχ. 18-2 Ο πίνακας επιλογής "Project Options" του περιβάλλοντος ανάπτυξης Dev C++.

Μας ενδιαφέρει το **MyNumerics.o** που είναι το «μεταγλωττισμένο **MyNumerics.cpp**».[3] Δες τώρα πώς θα το χρησιμοποιήσουμε.

Απλουστεύουμε το πρόγραμμα της §17.14 τόσο:

```
#include <iostream>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double q( double x );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
}

double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και το φυλάγουμε στο **testBisection.cpp** στο ευρετήριο **testBisection**.

Όπως βλέπεις, το μόνο που υπάρχει από τη *bisection()* είναι η επικεφαλίδα της που υπάρχει στο **MyNumerics.h**. Παρ' όλα αυτά την καλούμε τρεις φορές.

Αντιγράφουμε στο **testBisection** τα **MyNumerics.h** και **MyNumerics.o**.

Και τώρα:

- Δημιουργούμε ένα project είδους **Console Application**, με όνομα **testBisection** και το φυλάγουμε στο ευρετήριο **testBisection**.

[3] Αλλού, μπορεί να το δεις ως **MyNumerics.obj**.

- Προσθέτουμε στο project το αρχείο **testBisection.cpp**.
- Κλείνουμε, *χωρίς να φυλάξουμε*, το **main.cpp**. Με τον τρόπο αυτόν το αρχείο διαγράφεται από το project όπου μένει μόνο το **testBisection.cpp**.
- Επιλέγουμε **Project|Project options** και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "**Parameters**".
- Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2) με το "**Add Library or Object**" επιλέγουμε το **MyNumerics.o**. Με αυτόν τον τρόπο ζητούμε από τον συνδέτη, να συνδεσει και το περιεχόμενο αυτού του αρχείου στο εκτελέσιμο που θα δημιουργήσει.
- Για να πάρουμε το εκτελέσιμο, επιλέγουμε **Execute|Rebuild all**.

Στο **testBisection** θα δημιουργηθούν τα εξής αρχεία:

```
testBisection.exe
testBisection.o
```

Το **testBisection.exe** είναι το εκτελέσιμο. Το «**testBisection.o** είναι το μεταγλωττισμένο **testBisection.cpp**».

Στο παράδειγμα αυτό βλέπουμε το εξής: *Το πρόγραμμα αποτελείται από δύο αρχεία που μεταγλωττίζονται ξεχωριστά*. Αν ξαναγυρίσουμε σε αυτά που λέγαμε –για το τι θα δώσουμε στον πελάτη χωρίς να του αποκαλύψουμε τα μυστικά μας– τώρα έχουμε μιαν απάντηση: θα του δώσουμε τα **MyNumerics.h** και **MyNumerics.o**.

Παρατηρήσεις: ▶

1. Χρησιμοποιήσαμε το όνομα "**testBisection**" α) για το project, β) για το ευρετήριο που βάλαμε τα αρχεία και γ) για το αρχείο **cpp** στο οποίο υπάρχει η **main**. Θα μπορούσαμε να χρησιμοποιήσουμε τρία διαφορετικά ονόματα. *Το όνομα του εκτελέσιμου θα είναι ίδιο με αυτό του project*.
2. Στο πρόγραμμά μας εντάσσεται και η *ph* αφού υπάρχει στο **MyNumerics.cpp** επομένως και στο **MyNumerics.o**, παρ' όλο που δεν καλείται από το πρόγραμμά μας. Θα το «ξεφορτωθούμε» στη συνέχεια.
3. Θα μπορούσαμε να είχαμε βάλει ("Add to project") το **MyNumerics.cpp** στο project **testBisection** και να είχαμε και πάλι ξεχωριστή μεταγλώττιση, με την έννοια ότι θα είχαμε δύο ξεχωριστά αρχεία **testBisection.o** και **MyNumerics.o**. Αλλά, σε μια τέτοια περίπτωση μπορεί να σου έμενε η υποψία ότι η **main** «έβλεπε» τη *bisection()*. Τις ξεχωρίσαμε λοιπόν τελείως για να σε πείσουμε.
4. Δες έναν άλλον τρόπο που μπορείς να γράψεις το πρόγραμμά σου:

```
#include <iostream>
#include <cmath>

using namespace std;

extern "C++"
{
   struct MyNumericsXptn;
   void bisection( double (*f)(double),
                   double a, double b, double epsilon,
                   int nMax,
                   double& root, int& errCode );
}

double q( double x );

int main()
// τα υπόλοιπα όπως ήταν
```

Όπως βλέπεις, δεν βάλαμε την **#include "MyNumerics.h"** αλλά βάλαμε την **extern** η οποία δηλώνει ότι σε κάποιο άλλο αρχείο, που θα συνδεθεί στο πρόγραμμα, υπάρχει ένας τύπος δομής με το όνομα *MyNumericsXptn* και μια συνάρτηση με το όνομα *bisection* και την επικεφαλίδα που φαίνεται.

Αν το αρχείο που συνδέουμε έχει βγεί από μεταγλωττιστή C θα έπρεπε να δηλώσουμε "`extern "C"`". Τα διάφορα περιβάλλοντα ανάπτυξης σου επιτρέπουν να συνδέσεις αρχεία που έχουν βγει από μεταγλωττιστές άλλων γλωσσών. Φυσικά, σου επιτρέπουν να κάνεις και την κατάλληλη δήλωση **extern**, π.χ. "`extern "Ada"`", "`extern "FORTRAN"`" κλπ.

5. Δεν είπαμε οτιδήποτε για το `Makefile.win`. Δεν θα πούμε τίποτε περισσότερο από το ότι σε αυτό υπάρχουν οι εντολές για το κτίσιμο του εκτελέσιμου.

6. Αυτά που είπαμε για το περιβάλλον ανάπτυξης Dev C++ ισχύουν περίπου και για το περιβάλλον ανάπτυξης Code::Blocks. Για την Borland C++ v.5.5 σε ένα («μαύρο») παράθυρο `cmd.exe` δώσε:

```
C:\Borland\bc55\bin>bcc32 main.cpp MyNumerics.cpp
```

όπου "`main.cpp`" είναι αυτό που είδαμε πιο πάνω. Έτσι, θα πάρεις το `MyNumerics.obj`. Στη συνέχεια δίνεις:

```
C:\Borland\bc55\bin>bcc32 testBisection.cpp MyNumerics.obj
```

και παίρνεις το "`testBisection.exe`".◀

18.4 Μια Στατική Βιβλιοθήκη

Και τώρα θα κάνουμε μια *στατική βιβλιοθήκη* που θα περιέχει τις δύο συναρτήσεις *ph()* και *bisection()*.

Στο ευρετήριο `MyNumerics` αντιγράφουμε και πάλι τα `MyNumerics.cpp` και `MyNumerics.h` και –όπως περιγράψαμε στην προηγούμενη παράγραφο– δημιουργούμε ένα project με το όνομα `MyNumerics` αλλά αυτή τη φορά διαφορετικού είδους: "`Static Library`". Προσθέτουμε (Add to project) το `MyNumerics.cpp` και δίνουμε `Execute|Rebuild all`. Στο `MyNumerics`, εκτός από τα `MyNumerics.dev` και `MyNumerics.o`, θα δεις και το `MyNumerics.a`. Αυτή είναι η στατική βιβλιοθήκη[4] που θα χρησιμοποιήσουμε για να ξαναχτισουμε το πρόγραμμά μας.

Στο ευρετήριο `testBisectionL` αντιγράφουμε τα

```
testBisection.cpp
MyNumerics.a
MyNumerics.h
```

Στη συνέχεια:

- Δημιουργούμε ένα project "`Console Application`"με το όνομα `testBisectionL` και
- προσθέτουμε το `testBisection.cpp` (κλείνουμε χωρίς να φυλάξουμε το `main.cpp`).
- Επιλέγουμε `Project|Project options` και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "`Parameters`". Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2), με το "`Add Library or Object`", επιλέγουμε τη βιβλιοθήκη `MyNumerics.a`.
- Τέλος, επιλέγουμε `Execute|Rebuild all` για να πάρουμε το εκτελέσιμο που θα έχει το όνομα `testBisectionL.exe`,

Αυτό το εκτελέσιμο διαφέρει από το προηγούμενο ως προς το ότι δεν έχει και τη συνάρτηση *ph()*, αλλά μόνον τη *bisection()*.

Όπως καταλαβαίνεις, στον πελάτη, για τον οποίον συζητούσαμε, θα δώσουμε το `MyNumerics.a` και το `MyNumerics.h`.

---

[4] Μια άλλη συνηθισμένη κατάληξη (π.χ. από τη Borland C++) για στατικές βιβλιοθήκες είναι η "`.lib`".

18.5 "namespace": Το Πρόβλημα και η Λύση

Ας πούμε τώρα ότι γράψαμε τη βιβλιοθήκη περιγραμμάτων, γράψαμε τη στατική βιβλιοθήκη μας και θέλουμε να τις χρησιμοποιήσουμε σε ένα πρόγραμμα που γράφουμε μαζί με δυο βιβλιοθήκες που «κατεβάσαμε» από το Internet. Και ξαφνικά, ο μεταγλωττιστής διαμαρτύρεται: βρίσκει δύο συναρτήσεις με το όνομα *renew()* και δύο συναρτήσεις με το όνομα *ph()*. Και τώρα τι γίνεται; Καλά, άντε και αλλάζουμε αυτά τα δύο ονόματα. Είναι σχεδόν σίγουρο ότι μετά από λίγο θα έχουμε άλλο παρόμοιο πρόβλημα. Μηπως να βάλουμε κάποια «εξωφρενικά» ονόματα στις συναρτήσεις μας. Ε, όχι και έτσι...

Η C++ προσφέρει μια λύση για το πρόβλημα: τον **ονοματοχώρο** (namespace). Δες ένα παράδειγμα. Σε κάποιο πρόγραμμα έχουμε:

```
namespace test
{
   const double x = 1.5;
}
char x = 'A';
int main()
{
   int x = 4;
      :
   cout << x << "  " << ::x << "  " << test::x << endl;
```

Εδώ βλέπουμε:

- Έναν ονοματοχώρο, με όνομα *test*, όπου δηλώνεται μια σταθερά τύπου `double` με όνομα *x* και τιμή 1.5.
- Μια δήλωση της καθολικής μεταβλητής *x* τύπου `char` με αρχική τιμή `'A'`.
- Μια δήλωση μεταβλητής τύπου `int`, τοπικής στην `main`, με *x* όνομα αρχική τιμή 4.

Από τη `main` μπορούμε να έχουμε πρόσβαση και στα τρία παραπάνω αντικείμενα, όπως φαίνεται και από την εντολή εξόδου που δίνει:

```
4  A  1.5
```

Όπως ήδη ξέρουμε, με τα `x` και `::x` παίρνουμε την τοπική και την καθολική μεταβλητή αντιστοίχως. Με το `test::x` εννοούμε: το αντικείμενο με το όνομα *x* που δηλώνεται στον ονοματοχώρο *test*. Όπως βλέπεις, ο τελεστής "`::`" είναι ο τελεστής **επίλυσης** των προβλημάτων **εμβέλειας** (scope resolution) στο πρόγραμμά μας

Έστω λοιπόν ότι μας δίνεται μια βιβλιοθήκη προγραμμάτων. Αν ξέρεις ότι μέσα στη βιβλιοθήκη υπάρχει δηλωμένος κάποιος ονοματοχώρος δεν έχεις πρόβλημα. Αν δεν υπάρχει ονοματοχώρος –αν π.χ. είναι προγράμματα C– μπορείς να κάνεις το εξής: αν, ας πούμε, οι δηλώσεις της βιβλιοθήκης υπάρχουν στο αρχείο `bib.h` δηλώνεις:

```
namespace xbib
{
#include "bib.h"
}
```

Ύστερα από αυτό μέσα στο πρόγραμμά σου χρησιμοποιείς οτιδήποτε δηλώνεται στο bib.h με το πρόθεμα "`xbib::` ".

Φυσικά, η λύση που δίνεται με το **namespace** μπορεί να δημιουργήσει το αντίθετο πρόβλημα: να θέλεις να χρησιμοποιήσεις μια βιβλιοθήκη, να μην υπάρχει πρόβλημα με τα ονόματα και παρ' όλα αυτά να έχεις την υποχρέωση να γράφεις το όνομα του ονοματοχώρου πριν από όλα τα αντικείμενα που χρησιμοποιείς. Υπάρχει λύση και γι' αυτό. Ας πούμε ότι χρησιμοποιείς μια βιβλιοθήκη, που οι δηλώσεις της υπάρχουν στο `prlib.h` στον ονοματοχώρο *prlib*. Αν βάλεις στο πρόγραμμά σου:

```
#include "prlib.h"
using namespace prlib;
```

μπορείς να χρησιμοποιείς οτιδήποτε υπάρχει στον ονοματοχώρο *prlib* χωρίς το πρόθεμα "`prlib::`".

Αντί για την καθολική οδηγία "`using namespace`" που κάνει ορατά τα πάντα παντού, υπάρχει και η δήλωση **`using`** με την οποία μπορείς να κάνεις πιο επιλεκτική δουλειά. Ας πούμε ότι εσύ θέλεις στη συνάρτηση *f1()* να χρησιμοποιήσεις τον τύπο *cl1* της *prlib* ενώ στη συνάρτηση *f2()* θέλεις να χρησιμοποιήσεις τη συνάρτηση *fl0()* της *prlib*. Μπορείς να γράψεις:

```
double f1(...)
{
   using prlib::cl1;
   cl1 a, b;
      :
} // f1

void f2(...)
{
   using prlib::fl0;
      :
   q = fl0(1, u) + w;
} // f2
```

Η δήλωση "`using prlib::cl1`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα **`cl1`** μέσα στη συνάρτηση *f1()* μόνον. Παρομοίως, η δήλωση "`using prlib::fl0`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα "`fl0`" μέσα στη συνάρτηση *f2()* μόνον. Για τη δήλωση **using** ισχύουν οι κανόνες εμβέλειας που ξέρουμε.

Τώρα μπορείς να καταλάβεις και τις δηλώσεις "`using namespace std`" που βάζουμε στα προγράμματά μας. Η C++ έχει τον δικό της πάγιο ονοματοχώρο με το όνομα "`std`" (**st**and**d**). Μέσα σε αυτόν είναι δηλωμένα τα πάντα. Αν δεν το βάζαμε θα έπρεπε να γράφουμε **`std::strncpy`**, **`std::cout`**, **`std::endl`**, **`std::cin`**, **`std::bad_alloc`**, κλπ.

Ας έλθουμε τώρα στα παραδείγματά μας. Στο αρχείο **`MyTmpltLib.h`** κάνουμε την εξής αλλαγή:

```
namespace MyTmplt
{
struct MyTmpltLibXptn
// . . .
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// . . .
template< typename T >
void renew( T*& p, int ni, int nf )
// . . .
template< typename T >
T** new2d( int nR, int nC )
// . . .
template< typename T >
void delete2d( T**& a, int nR )
// . . .
} // namespace MyTmplt
```

Έτσι, τα ονόματα: *MyTmpltLibXptn*, *linSearch*, *renew*, *new2d* και *delete2d* εισάγονται στον ονοματοχώρο *MyTmplt*.

Εδώ όμως πρόσεξε το εξής:

- ***Ο ορισμός δομής (κλάσης) ορίζει και έναν ονοματοχώρο με το όνομα της δομής.***

Έχουμε λοιπον τον ονοματοχώρο *MyTmpltLibXptn* φωλιασμένο μέσα στον ονοματοχώρο *MyTmplt*. Έτσι, στη διαχείριση εξαιρέσεων στο πρόγραμμα του Παράδ. 2 της §16.13 οι κωδικοί σφάλματος θα έχουν «διπλό πρόθεμα», για παράδειγμα,

"`MyTmplt::MyTmpltLibXptn::domainError`"

```
// . . .
   catch( MyTmplt::MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
```

```
         case MyTmplt::MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
// . . .
```

Ακόμη, στην *elmntInTable* θα έχουμε:

```
// . . .
   int lPos( MyTmplt::linSearch( grElmnTbl, nElmn,
                                 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         MyTmplt::renew( grElmnTbl, nElmn, nReserved+incr );
// . . .
```

Ας δούμε τώρα πώς θα δηλώσουμε ονοματοχώρο στη στατική βιβλιοθήκη μας. Στο αρχείο **MyNumerics.h** κάνουμε την εξής αλλαγή:

```
namespace MyNmr
{
struct MyNumericsXptn
// . . .
double ph( const double a[], int m, double x );
// . . .
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );
} // namespace MyNmr
```

ενώ στο **MyNumerics.cpp** θα πρέπει να βάλουμε:

```
// . . .
double MyNmr::ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNmr::MyNumericsXptn( "ph",
                                   MyNmr::MyNumericsXptn::noArray );
// . . .
void MyNmr::bisection( double (*f)(double),
                       double a, double b, double epsilon,
                       int nMax,
                       double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNmr::MyNumericsXptn( "linSearch",
                     MyNmr::MyNumericsXptn::domainError, epsilon );
// . . .
```

Δεν θα μπορούσαμε να βάλουμε στο **MyNumerics.cpp** μια

```
using namespace MyNmr;
```

και να γλυτώσουμε από όλα αυτά τα "`MyNmr`"; Όχι! Οι ορισμοί των δύο συναρτήσεων είναι πλήρεις και ο μεταγλωττιστής θα τις μεταγλωττίσει ως *ph()* και *bisection()* και όχι ως *MyNmr*::*ph()* και *MyNmr*::*bisection()*.

Στο `testBisection.cpp` θα πρέπει να βάλουμε:

```
MyNmr::bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
```

Αν είχαμε και διαχείριση εξαιρέσεων θα βάζαμε:

```
catch( MyNmr::MyNumericsXptn& x )
{
   if ( x.errCode == MyNmr::MyNumericsXptn::domainError )
      cout << x.funcName << " called with negative ( "
           << x.errDblVal << " ) epsilon" << endl;
   else
      cout << "unexpected MyNumericsXptn from "
           << x.funcName << endl;
}
```

Αν θέλεις μπορείς να χρησιμοποιήσεις τον ίδιο ονοματοχώρο για πολλές βιβλιοθήκες. Για παράδειγμα, αντί για τα δύο ονόματα, *MyTmplt* και *MyNmr*, θα μπορούσαμε να χρησιμοποιήσουμε ένα μόνο όνομα, ας πούμε *MyNmspc*.

Γενικώς, να θυμάσαι ότι:

♦ ***Μια καλή βιβλιοθήκη έχει και τον ονοματοχώρο της.***

Αυτό μπορεί να σε γλυτώσει από προβλήματα χωρίς να σου δημιουργεί νέα αφού αρκεί μια "`using namespace …`" για να σου επιτρέψει να τον αγνοήσεις όπου δεν τον χρειάζεσαι.

18.6 Ανακεφαλαίωση

Οι βιβλιοθήκες συναρτήσεων είναι πολύτιμα εργαλεία για τον προγραμματιστή. Στο διαδίκτυο υπάρχουν πολλές και αρκετές από αυτές είναι πολύ καλής ποιότητας. Μπορείς να κτίσεις και δικές σου.

Οι *βιβλιοθήκες περιγραμμάτων συναρτήσεων* είναι οι πιο απλές αφού το περιεχόμενό τους είναι γραμμένο σε C++. Περιέχουν περιγράμματα συναρτήσεων, εξειδικεύσεις τους και πιθανότατα κάποια κλάση εξαιρέσεων. Καλό είναι να δηλώνονται μέσα σε έναν *ονοματοχώρο*.

Πριν δούμε τις βιβλιοθήκες μεταγλωττισμένων συναρτήσεων είδαμε πώς μπορούμε να μεταγλωττίζουμε χωριστά διάφορα κομμάτια του προγράμματος και με τη σύνδεση να παίρνουμε το εκτελέσιμο.

Οι βιβλιοθήκες μεταγλωττισμένων συναρτήσεων χωρίζονται σε δύο μεγάλες κατηγορίες: τις *στατικές* και τις *δυναμικής σύνδεσης*.

- Στατικές: Οι συναρτήσεις τους, που καλούνται από κάποιο πρόγραμμα, ενσωματώνονται (αντιγράφονται) στο εκτελέσιμο που μπορεί να εκτελείται χωρίς να είναι παρούσα η βιβλιοθήκη.
- Δυναμικής Σύνδεσης: Οι συναρτήσεις τους δεν ενσωματώνονται στο εκτελέσιμο αλλά φορτώνονται στη μνήμη όταν αυτό εκτελείται.

Για τη χωριστή μεταγλώτιση και τη δημιουργία και τη χρήση βιβλιοθηκών τηρούνται διαδικασίες που εξαρτώνται σε μεγαλύτερο ή μικρότερο βαθμό από το περιβάλλον ανάπτυξης ή/και το ΛΣ.

Όταν έχεις χωριστή μεταγλώττιση ή χρησιμοποιείς βιβλιοθήκες συναρτήσεων είναι απαραίτητα τα αρχεία επικεφαλίδων (`.h`).

Ασκήσεις

Α Ομάδα

18-1 Στις συναρτήσεις *ph()* και *bisection()* αντιστοιχίσαμε στο $\mathbb{R}$ τον `double`. Θα μπορούσαμε να είχαμε αντιστοιχίσει τον `float` ή τον `long double`. Αν κάνουμε τις συναρτήσεις περιγράμματα δίνουμε στον προγραμματιστή τη δυνατότητα να επιλέξει τον τύπο για την περίπτωση που τον ενδιαφέρει.

Αφού μετατρέψεις τις δύο συναρτήσεις σε περιγράμματα, να τις ενταξεις σε μια βιβλιοθήκη περιγραμμάτων `NumTmplt`. Γράψε προγράμματα για να τη δοκιμάσεις.

Β Ομάδα

18-2 Στο Κεφ. 15 είδαμε τον τύπο

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

και τις συναρτήσεις

```
GrElmn GrElmn_copyFromElmn( const Elmn& a );
void GrElmn_save( const GrElmn& a, ostream& bout );
void GrElmn_load( GrElmn& a, istream& bin );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_display( const GrElmn& a, ostream& tout );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_writeToTable( const GrElmn& a, ostream& tout );
```

για τη διαχείριση των στοιχείων του.

Κάνε μια στατική βιβλιοθήκη `GrElmnLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις. Δοκίμασέ την στα προγράμματα των §15.14.1 και 15.14.2.

Προσοχή! Δυο συναρτήσεις ρίχνουν εξαιρέσεις *ApplicXptn*. Αυτό πρέπει να αλλάξει: θα ρίχνουν εξαιρέσεις `GrElmnXptn`. Φυσικά, αυτόν τον τύπο θα τον ορίσεις εσύ!

18-3 Στο Κεφ. 15 είδαμε και τον τύπο

```
struct Date
{
   enum { saveSize = sizeof(int) + 2*sizeof(char) };
   unsigned int  year;
   unsigned char month;
   unsigned char day;
   Date( int yp=1, int mp=1, int dp=1 )
   {  year = yp;  month = mp;  day = dp;  }
}; // Date
```

του οποίου τα στοιχεία διαχειριστήκαμε με τις συναρτήσεις

```
void Date_save( const Date& a, ostream& bout );
```

```
void Date_load( Date& a, istream& bin );
```

και τους τελεστές

```
bool operator==( const Date& lhs, const Date& rhs );
bool operator<( const Date& lhs, const Date& rhs );
ostream& operator<<( ostream& tout, const Date& rhs );
```

Κάνε μια στατική βιβλιοθήκη `DateLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις και τους τελεστές. Δοκίμασέ την σε αυτά που λέγαμε για τον τύπο *Date*.

Ξαναλύσε την Άσκ. 15-1 με χρήση της βιβλιοθήκης.

Προσοχή! Οι συναρτήσεις αυτές δεν ρίχνουν εξαιρέσεις αλλά θα έπρεπε. Δες την προηγούμενη άσκηση και τις (αντίστοιχες) συναρτήσεις της.

Μέρος Γ
Αντικειμενοστρεφής Προγραμματισμός

κεφάλαιο

19

Από τις Δομές στις Κλάσεις

Habeas Corpus

Ο στόχος μας σε αυτό το κεφάλαιο:

- Θα θυμηθούμε τις δομές.
- Θα δούμε προβλήματα που υπάρχουν και πώς λύνονται με τις κλάσεις.

Προσδοκώμενα αποτελέσματα:

Όταν θα έχεις μελετήσει αυτό το κεφάλαιο θα μπορείς να σχεδιάσεις και να υλοποιήσεις μια απλή κλάση.

Έννοιες κλειδιά:

- *δομή*
- *κλάση*
- *εσωτερικά* (`private`) *μέλη*
- *ανοικτά* (`public`) *μέλη*
- *είδος των μεθόδων*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Ας ξεκινήσουμε από το "*Habeas Corpus*". Κυριολεκτικώς σημαίνει «να έχεις το σώμα σου» (να είσαι κύριος του σώματός σου). Αυτή είναι μια αρχή δικαίου που σήμερα σημαίνει ότι κάθε πολίτης έχει δικαίωμα να δικάζεται από τον «φυσικό δικαστή του». Η διατύπωσή της όμως έχει σχέση με τα κρατούντα την εποχή της φεουδαρχίας· ο φεουδάρχης ήταν απόλυτος κύριος του κάθε υπηκόου του (και του σώματός του) αφού, σε οποιαδήποτε διένεξη, δικαστής ήταν ο φεουδάρχης! Το "*Habeas Corpus*" κατακτήθηκε με την επανάσταση του Cromwell το 1679.

Και τι σχέση έχει με τον προγραμματισμό; Οι μεταβλητές (αντικείμενα) που ο τύπος τους είναι απλή δομή είναι στο «έλεος» του προγράμματος που τις χρησιμοποιεί (**εφαρμογή-πελάτης**, client application). Έτσι:

- Όπως λέγαμε στην §15.3 «*Ο δημιουργός δεν μας απαγορεύει να δηλώσουμε:*

```
Date d1( 2004, 14, 37 );
```

Θα πρέπει να τον εφοδιάσουμε με μερικούς ελέγχους ώστε να απαγορεύει τέτοια λάθη.»
- Σε μια μεταβλητή τύπου *Employee* μπορεί να βάλουμε την ημερομηνία γέννησης (*birthDate*) μεταγενέστερη της ημερομηνίας πρόσληψης (*emplDate*) ή αριθμό παιδιών (*numberOfChld*) αρνητικό.
- Σε μια μεταβλητή τύπου *GrElmn* μπορεί να βάλουμε αρνητικό ατομικό αριθμό (*geANumber*) ή αρνητικό ατομικό βάρος (*geAWeight*).

Εδώ θα μάθουμε ότι η C++, όπως και άλλες **αντικειμενοστρεφείς** γλώσσες προγραμματισμού, μας δίνουν τη δυνατότητα να δημιουργούμε αντικείμενα που έχουν πραγματική «κυριότητα του εαυτού τους» και δεν επιτρέπουν την «κακομεταχείρισή» τους από το περιβάλλον στο οποίο «ζουν».

19.1 Κλάσεις

Θα ξεκινήσουμε με τη διόρθωση των προβλημάτων της *Date*.[1] Πριν από οτιδήποτε άλλο θα πρέπει καταγράψουμε με ακρίβεια τη «συνθήκη νομιμότητας» για τη *Date*, δηλαδή τι είναι δεκτό και τι όχι:

$$(dYear > 0) \wedge (0 < dMonth \leq 12) \wedge (0 < dDay \leq lastDay(dYear, dMonth))$$

Αυτή είναι η **αναλλοίωτη της κλάσης** (class invariant) με την έννοια: ισχύει σε όλη τη διάρκεια της ζωής κάθε τιμής (αντικειμένου) τύπου *Date*.

Η *lastDay* είναι είναι μια συνάρτηση που, όταν την καλέσουμε όπως εδώ, μας δίνει την τελευταία ημέρα του μήνα *dMonth* του έτους *dYear*. Πώς θα είναι; Έτσι:

```
unsigned int lastDay( int y, int m )
{
   unsigned int fv;

   if ( m == 1 || m == 3 || m == 5 || m == 7 || m == 8 ||
        m == 10 || m == 12 )
      fv = 31;
   else if ( m == 4 || m == 6 || m == 9 || m == 11 )
      fv = 30;
   else // m == 2
      if ( isLeapYear(y) )  fv = 29;
                      else  fv = 28;
   return fv;
} // lastDay
```

[1] Αλλάζουμε και τα ονόματα των μελών: *dYear*, *dMonth*, *dDay* αντί για *year*, *month*, *day*. Η εξήγηση παρακάτω.

Και εκείνο το "`isLeapYear(y)`" τι είναι; Κλήση προς μια άλλη συνάρτηση

```
bool isLeapYear( int y )
{
   bool fv( false );

   if ( y%400 == 0 )                    fv = true;
   else if ( y%4 == 0 && y%100 != 0 )   fv = true;
   return fv;
} // isLeapYear
```

που τροφοδοτείται με ένα έτος *y* και μας επιστρέφει **`true`**, αν το έτος είναι δίσεκτο, αλλιώς **`false`**. Θα μπορούσαμε βέβαια να τη γράψουμε πιο απλά[2].

Αυτές οι συναρτήσεις φαίνονται ελλιπείς: Ούτε ένας έλεγχος για τις τιμές των παραμέτρων; Μπορούμε να βάλουμε ό,τι νάναι; Σωστή η παρατήρηση, αλλά… διάβασε παρακάτω.

Όπως είπαμε, για κάθε τιμή τύπου *Date* ή –όπως θα λέμε από εδώ και πέρα– *για κάθε αντικείμενο κλάσης Date, θα πρέπει να ισχύει η αναλλοίωτη*. Αυτό μπορεί να διασφαλίσθεί με *ελεγχόμενη* πρόσβαση στα μέλη των στοιχείων κλάσης *Date*. Δες πώς μπορεί να γίνει κάτι τέτοιο:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 )
   {  year = ay;  month = am;  day = ad;  }
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;
}; // Date
```

Πρόσεξε το "**`private:`**" πριν από τις δηλώσεις των μελών· σημαίνει ότι ένα πρόγραμμα που χρησιμοποιεί αντικείμενα κλάσης *Date* δεν έχει άμεση πρόσβαση στα μέλη των αντικειμένων. Οτιδήποτε δηλώνεται ως **`private`** δεν είναι «ορατό» έξω από το αντικείμενο. Έτσι, οι εντολές "`d1.dYear = -5; d1.dMonth = 29;`" είναι παράνομες: *δεν περνούν από τον μεταγλωττιστή*. Λέμε ότι τα μέλη έγιναν **εσωτερικά** (private).

Δεν κάναμε εσωτερικό τον δημιουργό, τον αφήσαμε **ανοικτό** (public), ώστε να έχουμε τη δυνατότητα να δηλώνουμε αντικείμενα κλάσης *Date*. Ο δημιουργός –και οτιδήποτε δηλώνεται μέσα στην κλάση– έχει πρόσβαση και στα εσωτερικά μέλη και μέσω αυτού μπορούμε:

- να ορίσουμε τις τιμές των μελών ή
- να τις αλλάξουμε.

Πάντως, δεν λύσαμε ακόμη το πρόβλημά μας: *μπορούμε ακόμη να δώσουμε μη αποδεκτές τιμές*. Επιπλέον, τώρα έχει προκύψει και ένα άλλο πρόβλημα: *πώς θα χρησιμοποιήσουμε τις τιμές των μελών*;

Ας ξεκινήσουμε με τη βελτίωση του δημιουργού: θα τον εφοδιάσουμε με ελέγχους:

```
   Date( int ay=1, int am=1, int ad=1 )
   {
      if ( ay <= 0 )
         throw . . .;
   // ay > 0
      dYear = ay;
      if ( am <= 0 || 12 < am )
         throw . . .;
   // dYear > 0 && ( 0 < am && am <= 12 )
      dMonth = am;
```

---

[2] Να η απλή μορφή:

```
bool isLeapYear( int y )
{  return ( y%400 == 0 || (y%4 == 0 && y%100 != 0) );  } // isLeapYear
```

```
      if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
         throw . . .;
   // dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
   // ( 0 < ad && ad <= lastDay(dYear, dMonth) )
      dDay = ad;
   // dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
   // ( 0 < dDay && dDay <= lastDay(dYear, dMonth) )
   } // Date
```

Όπως φαίνεται, *όταν ο δημιουργός τελειώσει τη δουλειά του –αν την τελειώσει– θα έχει δημιουργήσει ένα αντικείμενο που συμμορφώνεται με την αναλλοίωτη της κλάσης*.

Εδώ να παρατηρήσουμε δύο πράγματα:

- Όταν καλείται η *lastDay()* είναι σίγουρο ότι οι τιμές των ορισμάτων είναι «νόμιμες». Έτσι, δεν χρειάζονται οι έλεγχοι μέσα στη συνάρτηση που λέγαμε ότι λείπουν. Και πώς σιγουρεύουμε ότι δεν θα κληθεί από κάπου αλλού με λάθος ορίσματα; Θα την κρύψουμε, μαζί με την *isLeapYear()*, στην περιοχή **private** της κλάσης:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   unsigned int lastDay( int y, int m )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
}; // Date
```

Τώρα, οι δύο συναρτήσεις είναι προσβάσιμες από οτιδήποτε υπάρχει μέσα στον ορισμό της κλάσης αλλά όχι έξω από αυτήν.

- Δεν γράψαμε τον τύπο της εξαίρεσης· θα πρέπει να περιμένουμε να δούμε τις κλάσεις εξαιρέσεων της εφαρμογής-πελάτη; Όχι! *Μαζί με κάθε κλάση θα ορίζουμε και την αντίστοιχη κλάση εξαιρέσεων*. Εδώ θα ορίσουμε προς το παρόν μια:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr };
   char funcName[100];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
   DateXptn( const char* mn, int ec,
             int ev1=0, int ev2=0, int ev3=0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = ev1;  errVal2 = ev2;  errVal3 = ev3;  }
}; // DateXptn
```

Αργότερα θα την αλλάξουμε κάπως.

- Αν δεν βάζαμε τα "`// ΟΠΩΣ ΠΑΡΑΠΑΝΩ`" ο ορισμός της κλάσης θα ήταν ήδη μεγάλος και δεν θα ήταν εύκολο να ψάχνουμε κάτι που θέλουμε. Η C++ μας δίνει την εξής δυνατότητα: να βάλουμε μόνο τις επικεφαλίδες των συναρτήσεων και να τις ορίσουμε ολόκληρες έξω από την κλάση. Δηλαδή:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
private:
   unsigned int  dYear;
   unsigned char dMonth;
```

```
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

και

```
bool Date::isLeapYear( int y )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
unsigned int Date::lastDay( int y, int m )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Δηλαδή: εδώ ορίζουμε την *isLeapYear()* που δηλωσαμε στην κλάση (και ονοματοχώρο) *Date*. Τα ίδια κάνουμε και για τη *lastDay()*.

Τα ίδια ισχύουν και για τον δημιουργό αλλά εδώ δεν θα βάλουμε "`// ΟΠΩΣ ΠΑΡΑΠΑΝΩ`"· θα τον γράψουμε ολόκληρο:

```
Date::Date( int ay, int am, int ad )
{
   if ( ay <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, ay );
// ay > 0
   dYear = ay;
   if ( am <= 0 || 12 < am )
      throw DateXptn( "Date", DateXptn::monthErr, am );
// dYear > 0 && ( 0 < am && am <= 12 )
   dMonth = am;
   if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
      throw DateXptn( "Date", DateXptn::dayErr,
                      dYear, dMonth, ad );
// dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
// ( 0 < ad && ad <= lastDay(dYear, dMonth) )
   dDay = ad;
// dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
// ( 0 < dDay && dDay <= lastDay(dYear, dMonth) )
} // Date
```

Πρόσεξε τα εξής:

- Οι *προκαθορισμένες τιμές των παραμέτρων εμφανίζονται μόνο στη δήλωση του δημιουργού*, μέσα στον ορισμό της κλάσης *αλλά δεν εμφανίζονται ξανά στον ορισμό του δημιουργού*.
- Μέσα στο σώμα της συνάρτησης δεν χρειάζεται να βάλουμε "`Date::`" στα ονόματα των μελών. Αφού βάλαμε "`Date::Date`" στην επικεφαλίδα είμαστε μέσα στον ονοματοχώρο *Date* και το "`dYear`" σημαίνει "`Date::dYear`".
- Όταν έχουμε λάθος στο έτος ή στον μήνα στέλνουμε με την εξαίρεση μια τιμή αλλά όταν έχουμε λάθος στην ημέρα στέλνουμε τρεις. Γιατί;
  - Διότι το να πούμε ότι το *ad* είχε τιμή 31 δεν λέει και πολλά πράγματα αν δεν ξέρουμε ότι το *dMonth* έχει τιμή 6.
  - Και το να πούμε ότι το *ad* είχε τιμή 29 δεν δείχνει κάποιο πρόβλημα αν δεν ξέρουμε ότι το *dMonth* έχει τιμή 2 και το *dYear* έχει τιμή 2001.

Να δούμε τώρα το άλλο πρόβλημα: «*πώς θα χρησιμοποιήσουμε τις τιμές των μελών;*» Για την πρόσβαση στα μέλη εφοδιάζουμε την κλάση με κατάλληλες *ανοικτές* **συναρτήσεις-μέλη** (member fumctions) ή **μεθόδους** (methods):

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth()
                       const {  return static_cast<unsigned int>( dMonth );  }
   unsigned int getDay() const {  return static_cast<unsigned int>( dDay );  }
private:
```

```
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

Οι *getYear()*, *getMonth()*, *getDay()* είναι τέτοιες μέθοδοι. Πώς τις χρησιμοποιούμε; Δες το πρόγραμμα:

```
   Date d1( 2010, 10, 13 );
   cout << "d1: "
        << d1.getDay() << '.' << d1.getMonth() << '.' << d1.getYear() << endl;

   d1 = Date( 2008, 7, 9 );
   cout << "d1: "
        << d1.getDay() << '.' << d1.getMonth() << '.' << d1.getYear() << endl;
```

θα δώσει:

```
d1: 13.10.2010
d1: 9.7.2008
```

Όπως βλέπεις, χρησιμοποιήσαμε τον δημιουργό όχι μόνον για να δηλώσουμε τη *d1* με αρχική τιμή, αλλά και για να αλλάξουμε την τιμή της. Στην §15.3 μάθαμε ότι με τη "`Date(2008, 7, 9)`" καλούμε τον δημιουργό για να δημιουργήσει μια τιμή κλάσης *Date*. Αυτήν την τιμή εκχωρούμε στη *d1*.

Πρόσεξε όμως τη χρήση των μεθόδων: γράφουμε

"`d1.getDay()`", "`d1.getMonth()`", "`d1.getYear()`"

και όχι

"`Date::getDay(d1)`", "`Date::getMonth(d1)`", "`Date::getYear(d1)`"

Δηλαδή *οι μέθοδοι ανήκουν στο αντικείμενο και όχι στην κλάση!* Δηλαδή:

- ***Ένα αντικείμενο περικλείει –εκτός από δεδομένα– και τις μεθόδους για τον χειρισμό τους.***

Η **περίκλειση** (encapsulation) δεδομένων και εργαλείων χειρισμού τους είναι βασικό χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού.

Έχουμε ήδη χρησιμοποιήσει κατά κόρον την κλήση μεθόδων αντικειμένου με τον τελεστή "`.`"· θυμίσου τα `s.length()` ή `s.c_str()` για ένα αντικείμενο *s* κλάσης (*std*`::`)*string* ή τα `bin.open(…)` ή `bin.read(…)` για ένα αντικείμενο *bin* κλάσης (*std*`::`)*ifstream*. Μην αμφιβάλλεις ότι παρομοίως χρησιμοποιούμε και τον "`->`". Αν έχουμε:

```
   Date* pd( new Date(2008, 7, 9) );
```

η

```
   cout << "*pd: "
        << pd->getDay() << '.' << pd->getMonth() << '.'
        << pd->getYear() << endl;
```

θα δώσει:

```
*pd: 9.7.2008
```

Είδαμε πιο πάνω ότι με χρήση δημιουργού (και της εκχώρησης) μπορούμε να αλλάξουμε την τιμή ενός αντικειμένου. Αν όμως θέλουμε να αλλάξουμε, ας πούμε, μόνο τον μήνα τι κάνουμε; Να ένα παράδειγμα:

```
   d1 = Date( d1.getYear(), 5, d1.getDay() );
```

Με αυτήν την εντολή βάζουμε τον μήνα "5" κρατώντας χωρίς αλλαγή έτος και ημέρα.

Καλό θα είναι όμως να μπορούμε να αλλάξουμε την τιμή οποιουδήποτε μέλους ενός αντικειμένου. Δηλώνουμε λοιπόν τρεις νέες μεθόδους:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
```

```
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth() const {  return dMonth;  }
   unsigned int getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

και τις ορίζουμε ως εξής:

```
void Date::setYear( int ay )
{
   if ( ay <= 0 )
      throw DateXptn( "setYear", DateXptn::yearErr, ay );
   if ( dDay <= 0 || lastDay(ay, dMonth) < dDay )
      throw DateXptn( "setYear", DateXptn::dayErr,
                      ay, dMonth, dDay );
   dYear = ay;
} // Date::setYear
```

Καλά ο πρώτος έλεγχος, ο δεύτερος τι είναι; Σκέψου την περίπτωση που η *d* έχει ως τιμή την `29.02.2000` και κάνουμε απόπειρα (`d.setYear(2001)`) να αλλάξουμε το έτος σε `2001`. Παρόμοιο πρόβλημα υπάρχει και στην

```
void Date::setMonth( int am )
{
   if ( am <= 0 || 12 < am )
      throw DateXptn( "setMonth", DateXptn::monthErr, am );
   if ( dDay <= 0 || lastDay(dYear, am) < dDay )
      throw DateXptn( "setMonth", DateXptn::dayErr,
                      dYear, am, dDay );
   dMonth = am;
} // Date::setMonth
```

Στην περίπτωση αυτή ο δεύτερος έλεγχος χρειάζεται αν –για παράδειγμα– έχουμε τιμή της *d* την `31.12.2010` και δώσουμε `d.setMonth(6)`.

```
void Date::setDay( int ad )
{
   if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
      throw DateXptn( "setDay", DateXptn::dayErr,
                      dYear, dMonth, ad );
   dDay = ad;
} // Date::setDay
```

Η *setDay* είναι απλή και δεν χρειάζεται εξηγήσεις.

Σημείωση: ▶

Οι τρεις συναρτήσεις "*set*" που γράψαμε είναι καλές για επίδειξη αλλά, για την κλάση *Date*, σχεδόν άχρηστες σε πραγματικό πρόγραμμα. Για σκέψου τι θα γίνεται όποτε έχεις να αλλάξεις τις τιμές δύο μελών: Για να μην πέσει εξαίρεση θα πρέπει να αλλάξεις πρώτα τη μέρα ή τον μήνα (ή το έτος). Πρόσεξε το εξής παράδειγμα: Αν η *d1* έχει τιμή 31.05.2007 και θέλουμε να την αλλάξουμε σε 30.04.2007 θα πρέπει να αλλάξουμε πρώτα τη μέρα και μετά τον μήνα. Αν θέλουμε να αλλάξουμε την 30.04.2007 σε 31.05.2007 θα πρέπει να αλλάξουμε πρώτα τον μήνα. Συμπέρασμα: στη *Date* δεν βγαίνει άκρη με τις "*set*". *Η αλλαγή τιμής με τη χρήση του δημιουργού είναι η πιο σίγουρη.*

```
   Date d1( 2007, 5, 31 );                    // d1 == 31.05.2007

   d1 = Date( d1.getYear(), 4, 30 );          // d1 == 30.04.2007
   d1 = Date( d1.getYear(), 5, 31 );          // d1 == 31.05.2007
   d1 = Date( d1.getYear(), 12, d1.getDay() ); // d1 == 31.12.2007
```

```
   d1 = Date( d1.getYear()+1,
              d1.getMonth(), d1.getDay() );    // d1 == 31.12.2008
```

Με αυτόν τον τρόπο κάνει περισσότερες πράξεις αλλά είναι αυτός που δουλεύει σίγουρα. ◀

Για να λύσεις την άσκ. 15-1 θα πρέπει να έγραψες για τη *Date* μια:

```
void Date_load( Date& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.year), sizeof(a.year) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&a.month),
                sizeof(a.month) );
      bin.read( reinterpret_cast<char*>(&a.day), sizeof(a.day) );
      if ( bin.fail() )
         throw XXX_Xptn( "Date_load", XXX_Xptn::cannotRead );
   }
}; // Date_load
```

Πώς θα τη μετατρέψουμε ώστε να την περικλείσουμε στα αντικείμενα κλάσης *Date;* Σε αυτήν τη μορφή τη χρησιμοποιούσαμε ως εξής: Αν

```
   ifstream dateFile( ..., ios_base::binary );
   Date d;
```

γράφουμε:

```
   Date_load( d, dateFile );
```

Τώρα, η *load()* θα είναι «ιδιοκτησία» του αντικειμένου και θα γράφουμε:

```
   d.load( dateFile );
```

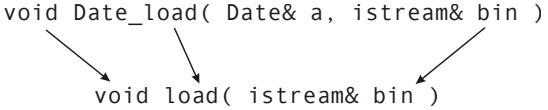
Θα πρέπει λοιπόν να τη δηλώσουμε μέσα στην κλάση:

```
struct Date
{
// . . .
   void load( istream& bin );
// . . .
private:
// . . .
}; // Date
```

και να την ορίσουμε:

```
void Date::load( istream& bin )
{
   bin.read( reinterpret_cast<char*>(&dYear), sizeof(dYear) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&dMonth),
                sizeof(dMonth) );
      bin.read( reinterpret_cast<char*>(&dDay), sizeof(dDay) );
      if ( bin.fail() )
         throw DateXptn( "load", DateXptn::cannotRead );
   }
}; // Date::load
```

Τη σχέση μεταξύ της εξωτερικής συνάρτησης και της περικλειόμενης μεθόδου τη δίνουμε διαγραμματικώς κάπως έτσι:



Δεν υπάρχει αλλαγή στον τύπο (‘**void**” στην περίπτωσή μας) και στις παραμέτρους εκτός από την παράμετρο-αντικείμενο. Η παράμετρος-αντικείμενο εξαφανίζεται αφού τώρα το αντικείμενο είναι ο «ιδιοκτήτης» της μεθόδου.

19.1.1 "const"

Στην άσκ. 15-1 δίνεται και η συνάρτηση

```
void Date_save( const Date& a, ostream& bout )
{
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&a.year),
               sizeof(a.year) );
   bout.write( reinterpret_cast<const char*>(&a.month),
               sizeof(a.month) );
   bout.write( reinterpret_cast<const char*>(&a.day),
               sizeof(a.day) );
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::cannotWrite );
}; // Date_save
```

που τη μετατρέπουμε σε μέθοδο ως εξής: Τη δηλώνουμε μέσα στην κλάση:

```
// . . .
   void save( istream& bin ) const;
// . . .
```

και την ορίζουμε:

```
void Date::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw DateXptn( "save", DateXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(dYear),
               sizeof(dYear) );
   bout.write( reinterpret_cast<const char*>(dMonth),
               sizeof(dMonth) );
   bout.write( reinterpret_cast<const char*>(dDay),
               sizeof(dDay) );
   if ( bout.fail() )
      throw DateXptn( "save", DateXptn::cannotWrite );
}; // Date::save
```

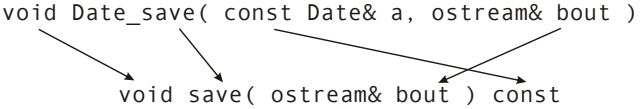
Το νέο στοιχείο εδώ είναι

- το "`const`" στον τύπο του αντικειμένου στην *Date_save()*
- που εμφανίζεται στο τέλος της επικεφαλίδας της μεθόδου *Date*::*save()*.

Τι σημαίνει; Στη *Date_save()* εγγυάται ότι η συνάρτηση δεν αλλάζει την τιμή του αντικειμένου *a*. Το ίδιο σημαίνει και στη μέθοδο:

♦ ***Με το χαρακτηριστικό "const" στο τέλος της επικεφαλίδας της μεθόδου σημειώνουμε ότι η μέθοδος δεν αλλάζει την τιμή οποιουδήποτε μέλους του αντικειμένου που την περικλείει.***

Τώρα οι αντιστοιχίες φαίνονται διαγραμματικώς ως εξής:



Πρόσεξε ότι βλέπουμε το "`const`" και στις τρεις μεθόδους "*get*" αλλά όχι στις "*set*" ούτε στη *load()*.

19.1.2 Βοηθητικές Συναρτήσεις

Βάλαμε τις συναρτήσεις *lastDay()* και *isLeapYear()* στην περιοχή `private` της κλάσης ώστε να είναι διαθέσιμες μόνο στις μεθόδους της. Από αυτές καλούμε τη *lastDay* και μάλιστα, πολύ προσεκτικά, με «σίγουρες» τιμές ορισμάτων!

Ακόμη, πρόσεξε ότι οι δύο συναρτήσεις δεν αναφέρονται σε συγκεκριμένο αντικείμενο· αναφέρονται στην κλάση. Λέμε ότι είναι **βοηθητικές συναρτήσεις** (helper functions) της κλάσης.[3]

Για να το καταλάβεις αυτό το «*δεν αναφέρονται σε συγκεκριμένο αντικείμενο*», δες πώς θα ήταν αν αναφέρονταν:

```
bool isLeapYear() const
{
   bool fv( false );

   if ( dYear%400 == 0 )                          fv = true;
   else if ( dYear%4 == 0 && dYear%100 != 0 )  fv = true;
   return fv;
} // isLeapYear
```

Δηλαδή, έχουμε μια συνάρτηση χωρίς παραμέτρους που υπολογίζει την τιμή της με βάση τις τιμές των μελών του αντικειμένου. Το ίδιο και η

```
unsigned int lastDay() const
{
   unsigned int fv;

   if ( dMonth == 1 || dMonth == 3 || dMonth == 5 || dMonth == 7
        || dMonth == 8 || dMonth == 10 || dMonth == 12 )
      fv = 31;
   else if ( dMonth == 4 || dMonth == 6 || dMonth == 9 ||
             dMonth == 11 )
      fv = 30;
   else // month == 2
      if ( isLeapYear(dYear) )  fv = 29;
                          else  fv = 28;
   return fv;
} // lastDay
```

Αν όμως γυρίσεις πίσω και δεις ότι καλούμε τη *lastDay()* δύο φορές με παραμέτρους *dMonth* και *dYear* και άλλες δύο φορές με άλλες παραμέτρους. Καταλαβαίνεις ότι μάλλον θα περιπλέκαμε τα προγράμματά μας.

Το σοβαρότερο πρόβλημα όμως είναι το νοηματικό. Αν έχουμε

```
Date d;
```

τι νόημα έχει το `d.lastDay()`; Τελευταία ημέρα του μήνα που υπάρχει στην τιμή της *d*!!! Και το `d.isLeapYear()` θα μας δώσει `true` αν είναι δίσεκτη η ημερομηνία *d*;! Όχι βέβαια, αφορά μόνον το έτος της ημερομηνίας. Καταλαβαίνεις ότι αυτά δεν στέκουν.

19.1.3 "class" και "public"

Η κλάση μας τώρα έχει γίνει έτσι:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned char getMonth() const {  return dMonth;  }
   unsigned char getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   void load( istream& bin );
   void save( ostream& bout ) const;
private:
   unsigned int  dYear;
   unsigned char dMonth;
```

[3] Αργότερα θα δεις ότι θα τις διακοσμήσουμε και με ένα "`static`"!

```
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

Ένας άλλος τρόπος να τη γράψουμε είναι ο εξής:

```
class Date
{
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned char getMonth() const {  return dMonth;  }
   unsigned char getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   void load( istream& bin );
   void save( ostream& bout ) const;
}; // Date
```

Δηλώνοντας μια κλάση με το "`struct`" εννοούμε ότι *κατ' αρχήν έχει όλα τα μέλη της ανοικτά*. Αν θέλουμε να κρύψουμε κάποια από αυτά θα πρέπει να τα βάλουμε σε περιοχή "`private`".

Αν η δήλωση γίνεται με το "`class`" η κλάση *κατ' αρχήν έχει όλα τα μέλη της εσωτερικά*. Αν θέλουμε να ανοίξουμε κάποια από αυτά θα πρέπει να τα βάλουμε σε περιοχή "`public`".

Με οποιονδήποτε από τους δύο τρόπους και αν κάνουμε τη δήλωση:

- Έχουμε τη δυνατότητα για πολλές περιοχές "`public`" και "`private`".
- Η οποιαδήποτε επαφή του κάθε προγράμματος με το αντικείμενο γίνεται μέσω αυτών που υπάρχουν στις περιοχές "`public`". Αυτά αποτελούν και το **τμήμα διεπαφής** (interface part) του αντικειμένου.

Πάντως παρακάτω θα δώσουμε ένα κριτήριο για να επιλέγεις μεταξύ "`class`" και "`struct`".

Τη συγκεκριμένη κλάση θα την ορίζουμε ως εξής:

```
class Date
{
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth() const {  return dMonth;  }
   unsigned int getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   void load( istream& bin );
   void save( ostream& bout ) const;
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

Οι τρεις τρόποι ορισμού της *Date* που είδες παραπάνω είναι ισοδύναμοι.

Η κλάση εξαιρέσεων είναι:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[100];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
   DateXptn( const char* mn, int ec,
             int ev1 = 0, int ev2 = 0, int ev3 = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = ev1;  errVal2 = ev2;  errVal3 = ev3;  }
}; // DateXptn
```

19.1.4 Επιφόρτωση Τελεστών

Για τη *Date* είχαμε επιφορτώσει και κάποιους τελεστές (§15.5). Τι θα γίνει με αυτούς; Θα τους επιφορτώσουμε με μεθόδους;

Για τους συγκεκριμένους ("`<<`", "`<`", "`==`") κρατούμε τις υλοποιήσεις όπως τις έχουμε αλλά με μια διαφορά: αφού είναι εξωτερικές για την κλάση δεν είναι δυνατόν να έχουν πρόσβαση στα μέλη του αντικειμένου και πρέπει να χρησιμοποιήσουμε τις αντίστοιχες μεθόδους "*get*".

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << rhs.getDay() << '.' << rhs.getMonth() << '.'
               << rhs.getYear();
} // operator<<( ostream& tout, const Date

bool operator<( const Date& lhs, const Date& rhs )
{
   bool fv;

   if ( lhs.getYear() < rhs.getYear() )           fv = true;
   else if ( lhs.getYear() > rhs.getYear() )      fv = false;
   else // lhs.getYear() == rhs.getYear()
   {
      if ( lhs.getMonth() < rhs.getMonth() )      fv = true;
      else if ( lhs.getMonth() > rhs.getMonth() ) fv = false;
      else // lhs.getYear() == rhs.getMonth() &&
           // lhs.getMonth() == rhs.getMonth()
         fv = ( lhs.getDay() < rhs.getDay() );
   }
   return fv;
} // operator<( const Date . . .

bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.getYear() == rhs.getYear() &&
            lhs.getMonth() == rhs.getMonth() &&
            lhs.getDay() == rhs.getDay() );
}; // operator==( const Date
```

Θα πεις «ε, όχι και "*όπως είναι αλλά με μια διαφορά*"! Η διαφορά είναι σοβαρή αφού φορτώνουμε τις συναρτήσεις με τόσες κλήσεις μεθόδων, δηλαδή άλλων συναρτήσεων». Ναι μεν αλλά, όπως θα δεις στη συνέχεια, αυτές είναι "`inline`".

Πάντως, στη συνέχεια θα δούμε έναν τρόπο για να δίνουμε δικαίωμα σε μια (εξωτερική) συνάρτηση να έχει κατ' ευθείαν πρόσβαση στα μέλη αντικειμένου που βρίσκονται σε περιοχή **private**.

19.1.5 Ονοματολογία

Γιατί αλλάξαμε τα ονόματα των μελών της *Date* σε *dYear*, *dMonth* και *dDay*; Αυτή είναι μια συνηθισμένη σύμβαση για να φαίνεται ότι πρόκειται για μέλη μιας κλάσης (που το όνομά της αρχίζει από *d*). Μπορεί να δεις και δύο ή τρία γράμματα (στην αρχή).

Άλλοι προτιμούν να βάζουν πριν από το όνομα του μέλους το «*m*» –από το «member» (μέλος)– και στην περίπτωσή μας θα έγραφαν: *mYear*, *mMonth* και *mDay*. Άλλοι πάλι θα έγραφαν *_year*, *_month* και *_day*.

19.2 Το Είδος των Μεθόδων

Οι κανόνες που έχουμε μάθει για το είδος των συναρτήσεων ισχύουν και για τις μεθόδους μιας κλάσης. Αλλά για να τους εφαρμόσεις θα πρέπει να παίρνεις υπόψη σου ότι *και το αντικείμενο παίζει ρόλο ορίσματος* όπως είδαμε στο τέλος της §19.1 και στην §19.1.1.

Ας δούμε τη *setYear()*. Αν τη γράφαμε για τη «παλιά» "`struct Date`" πώς θα ήταν; Κάπως έτσι:

```
void Date_setYear( Date& d, int ay )
```

Εδώ, σε συμφωνία με τους κανόνες μας, βάζουμε "`void`" διότι η συνάρτηση αλλάζει την τιμή της πρώτης παραμέτρου. Με τις αντιστοιχίες που είδαμε στην §19.1 γράφουμε μέθοδο:

```
void setYear( int ay )
```

Η *getYear()* για τη «παλιά» "`struct Date`" θα ήταν κάπως έτσι:

```
unsigned int Date_getYear( Date d )
```

ή, καλύτερα,

```
unsigned int Date_getYear( const Date& d )
```

Σε συμφωνία με τους κανόνες μας, βάζουμε γράφουμε συνάρτηση με τύπο διότι το μόνο που έχει να κάνει είναι να επιστρέψει μια τιμή. Με τις αντιστοιχίες που είδαμε στην §19.1.1 γράφουμε μέθοδο:

```
unsigned int getYear() const
```

Δηλαδή: *Για να επιλέξουμε το είδος μιας μεθόδου για μια κλάση K βάζουμε στη μέθοδο μια επιπλέον παράμετρο* –`K& a` *ή* `const K& a`– *και παίρνουμε την απόφασή μας.*

Όπως καταλαβαίνεις, αν γράφεις τις μεθόδους σύμφωνα με τους κανόνες που έχουμε βάλει, όποτε γράφεις μέθοδο με τύπο θα πρέπει να της δίνεις και το χαρακτηριστικό "`const`".

Πάντως και τώρα, για να είμαστε σε συμφωνία με τη «φιλοσοφία της C++» (C), σε ορισμένες περιπτώσεις θα παραβιάζουμε τους κανόνες μας.

19.3 Κατανομή σε Αρχεία

Ο συνηθισμένος τρόπος κατανομής του ορισμού της κλάσης σε αρχεία είναι ο εξής –για την *Date*:

- Στο αρχείο `Date.h` (μετά και την αλλαγή των ονομάτων):

```
#ifndef _DATE_H
#define _DATE_H

#include <fstream>
#include <string>

using namespace std;

class Date
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

```
struct DateXptn
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

ostream& operator<<( ostream& tout, const Date& rhs );
bool operator<( const Date& lhs, const Date& rhs );
bool operator==( const Date& lhs, const Date& rhs );

#endif // _DATE_H
```

- Στο αρχείο `Date.cpp`:

```
#ifndef _DATE_CPP
#define _DATE_CPP

#include <fstream>
#include "Date.h"

Date::Date( int ay, int am, int ad )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setYear( int ay )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setMonth( int am )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setDay( int ad )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::load( istream& bin )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::save( ostream& bout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool Date::isLeapYear( int y )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
unsigned int Date::lastDay( int y, int m )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

ostream& operator<<( ostream& tout, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool operator<( const Date& lhs, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool operator==( const Date& lhs, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _DATE_CPP
```

Τώρα, μέσα στο αρχείο του προγράμματος που χρησιμοποιεί τη *Date*, αρκεί να βάλεις "`#include "Date.cpp"`".

Αν θέλεις μπορείς να κάνεις και χωριστή μεταγλώττιση και να δώσεις στο project του προγράμματός σου για σύνδεση το `Date.o` (ή `Date.obj`). Στην περίπτωση αυτή στο πρόγραμμά σου θα βάλεις "`#include "Date.h"`".[4] Θα δούμε τέτοιο παράδειγμα αργότερα.

19.3.1 Τα «Μυστικά» της Υλοποίησης

Σε προηγούμενη παράγραφο λέγαμε για τις βοηθητικές συναρτήσεις: «*Βάλαμε τις συναρτήσεις lastDay και isLeapYear στην περιοχή* `private` *της κλάσης ώστε να είναι διαθέσιμες μόνο στις μεθόδους της.*» Και στην υποσημείωση, πιο πάνω: «*Στον "πελάτη", για τον οποίον συζητούσαμε στο προηγούμενο κεφάλαιο, θα δώσεις τα* `Date.h` *και* `Date.obj`.» Μα τι γίνεται εδώ; Όλα τα κρύβουμε, όλα είναι μυστικά; Ναι! Και μάλιστα με δύο έννοιες:

- **Απόκρυψη πληροφορίας** (information hiding): Το πώς υλοποιείται η κάθε μέθοδος δεν αφορά αυτόν που ζητάει κάτι από ένα αντικείμενο. Του λέει «θέλω να γίνει αυτό» ή «θέλω αυτήν την πληροφορία» και –αν δεν ζητείται κάτι παράνομο, δηλαδή παραβίαση

[4] Στον "πελάτη", για τον οποίον συζητούσαμε στο προηγούμενο κεφάλαιο, θα δώσεις τα `Date.h` και `Date.obj`.

της αναλλοίωτης– το αντικείμενο αποκρίνεται με την αντίστοιχη μέθοδο. Αν αυτός που διατυπώνει την απαίτηση (ένα πρόγραμμα-πελάτης ή ένα άλλο αντικείμενο) δεν γνωρίζει τον –για την ακρίβεια: δεν στηρίζεται στον– τρόπο υλοποίησης της μεθόδου τότε έχουμε δυνατότητα να την αλλάξουμε – συνήθως να τη βελτιώσουμε– *χωρίς να χρειαστεί να μεταγλωττίσουμε ολόκληρη την εφαρμογή*: μεταγλωττίζουμε μόνον την υλοποίηση της κλάσης και συνδέουμε το νέο αρχείο obj που προκύπτει με την εφαρμογή. Άλλωστε η πείρα λέει ότι όταν οι προγραμματιστές χρησιμοποιούν μια κλάση βασιζόμενοι μόνο στις προδιαγραφές των μεθόδων –και χωρίς να ξέρουν την υλοποίηση– γράφουν προγράμματα καλύτερης ποιότητας (χωρίς «εξυπνάδες»!)

- **Εμπορικό μυστικό υλοποίησης**: Όταν ένας προγραμματιστής γράφει μια κλάση για έναν πελάτη μπορεί να του δώσει
  - τον ορισμό της κλάσης και το αρχείο obj ή
  - όλον τον αρχικό κώδικα, αλλά σε πολύ υψηλότερη τιμή.

  Στην πρώτη περίπτωση ο προγραμματιστής έχει δικαίωμα να κρατήσει μυστικές από τον πελάτη τις λεπτομέρειες της υλοποίησης.

19.4 Μέθοδοι "inline"

Στην *Date*, όπως την είδαμε πιο πάνω, πρόσεξε και κάτι άλλο: Όλο το τμήμα διεπαφής περιέχει δηλώσεις εκτός από τις τρεις (απλούστερες) μεθόδους, τις "*get*" για τις οποίες δίνουμε πλήρεις ορισμούς. Κάτι τέτοιο θα το βλέπεις πολύ συχνά. Γιατί;

♦ ***Όποια μέθοδος ορίζεται μέσα στην κλάση θεωρείται ότι έχει και το χαρακτηριστικό "inline" παρ' όλο που δεν το γράφουμε.***

Οι τρεις μέθοδοι "*get*" είναι αρκετά απλές και το "**inline**" θα έχει αποτέλεσμα. Αν θέλεις να τις βγάλεις έξω από τον ορισμό της κλάσης και να μην χάσεις το πλεονέκτημα τις δηλώνεις ως:

```
class Date
// . . .
   unsigned int getYear() const;
   unsigned char getMonth() const;
   unsigned char getDay() const;
// . . .
}; // Date
```

και τις ορίζεις:

```
inline unsigned int Date::getYear() const {  return dYear;  }
inline unsigned int getMonth()
                      const {  return static_cast<unsigned int>( dMonth );  }
inline unsigned int getDay() const
                             {  return static_cast<unsigned int>( dDay );  }
```

Καταλαβαίνεις τώρα γιατί η επιφόρτωση των τελεστών δεν θα επιβαρύνει τον χρόνο εκτέλεσης του προγράμματος παρά το ότι καλούμε μεθόδους "*get*".

Φυσικά, μπορείς να βάλεις το "**inline**" και σε οποιαδήποτε άλλη μέθοδο και να αφήσεις τον μεταγλωττιστή να αποφασίσει τι γίνεται και τι δεν γίνεται.

19.5 Αναλλοίωτη της Κλάσης – Κλάσεις Εξαιρέσεων

Όπως είπαμε, για κάθε αντικείμενο μιας κλάσης θα πρέπει πάντοτε να ισχύει η **αναλλοίωτη της κλάσης** (class invariant). Η συνθήκη

$$(dYear > 0) \wedge (0 < dMonth \leq 12) \wedge (0 < dDay \leq lastDay(dYear, dMonth))$$

είναι η αναλλοίωτη της κλάσης *Date*.

Η αναλλοίωτη μπορεί να παραβιάζεται προσωρινώς, όταν εκτελείται κάποια μέθοδος, αλλά ισχύει πριν την έναρξη και μετά τον τερματισμό της εκτέλεσής της.

Για κάθε κλάση που θα γράφουμε θα γράφουμε και μιαν αντίστοιχη κλάση (δομή) εξαιρέσεων σαν και αυτές που γράφαμε μέχρι τώρα. *Η αναλλοίωτη είναι ο οδηγός μας για το τι εξαιρέσεις (με ποιον κωδικό σφάλματος) ρίχνουμε και από πού.*

♦ ***Κάθε φορά που οδηγούμαστε σε παραβίαση της αναλλοίωτης ρίχνουμε εξαίρεση.***

Στη *DateXptn* οι κωδικοί σφάλματος αντιστοιχούν σε παραβιάσεις τμημάτων της αναλλοίωτης:

- Ο *yearErr* αντιστοιχεί στην παραβίαση της $dYear > 0$.
- Ο *monthErr* στην παραβίαση της $0 < dMonth \leq 12$ και
- ο *dayErr* στην παραβίαση της $0 < dDay \leq lastDay(dYear, dMonth)$.

Εξαιρέσεις ρίχνουμε και όταν διαγνώσουμε πρόβλημα στην αλληλεπίδραση ενός αντικειμένου με το περιβάλλον του. Με άλλα λόγια: αφού η αλληλεπίδραση με το περιβάλλον γίνεται με τις μεθόδους ρίχνουμε εξαίρεση όταν κατά την εκτέλεση κάποιας μεθόδου παραβιάζονται οι προδιαγραφές της.

Για παράδειγμα, ας πάρουμε τη *Date* και την *DateXptn*. Ποιες είναι οι προδιαγραφές της *load()*;

Προϋπόθεση: Το ρεύμα *bin* δεν έχει πρόβλημα εκτός από *eof*.
Απαίτηση: Στο αντικείμενο (ιδιοκτήτη της *load()*) φορτώνεται η τιμή του επόμενου αντικειμένου από το αρχείο.

Κανονικά θε πρέπει να γράψουμε:

```
void Date::load( istream& bin )
{
   if ( bin.fail() && !bin.eof() )
         throw DateXptn( "load", DateXptn::fileNotOpen );
   bin.read( reinterpret_cast<char*>(&dYear), sizeof(dYear) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&dMonth),
                sizeof(dMonth) );
      bin.read( reinterpret_cast<char*>(&dDay), sizeof(dDay) );
      if ( bin.fail() )
         throw DateXptn( "load", DateXptn::cannotRead );
   }
}; // Date::load
```

- Αν η *αρχική* `bin.fail()` επιστρέψει `true` παραβιάζεται η προϋπόθεση και ρίχνουμε εξαίρεση *DateXptn* με κωδικό σφάλματος *fileNotOpen* (δεν είναι και τόσο κυριολεκτικός).
- Αν επιστρέψει `true` η *τελική* `bin.fail()` σημαίνει ότι απέτυχε η ανάγνωση (ολικώς ή μερικώς) και έτσι δεν θα υπάρξει ανταπόκριση στη απαίτηση. Για τον λόγο αυτόν ρίχνουμε εξαίρεση *DateXptn* με κωδικό *cannotRead*.

Παρατηρήσεις: ▶

1. Γιατί χειριζόμαστε ξεχωριστά την περίπτωση *eof*; Διότι, όπως έχεις δει ήδη, το να διαβάζουμε μέχρι το τέλος αρχείου είναι πολύ συνηθισμένο.

Στη συνέχεια, όπως θα δεις, δεν θα βάζουμε τον πρώτο έλεγχο `bin.fail()`. Θα χρησιμοποιούμε την πρώτη μορφή της *load()*.

2. Πάντως η *load()* χρειάζεται περισσότερη προσοχή. Για παράδειγμα, αν αποτύχει το διάβασμα, θα πρέπει να μην καταστρέφει την τιμή του αντικειμένου. Θα την ξαναδούμε στη συνέχεια.◀

Πρόσεξε ακόμη κάτι πολύ σημαντικό που προκύπτει από τη διαχείριση με μεθόδους μελών που έχουμε περικλείσει σε περιοχές **private**: Αν δεις ότι σε κάποιο μέλος περνάνε απαράδεκτες τιμές ξέρεις πού ακριβώς –σε ποιες μεθόδους– πρέπει να ψάξεις για το πρόβλημα. Για παράδειγμα, αν δεις σε κάποιο πρόγραμμά σου ότι σε αντικείμενα κλάσης *Date* το *dYear*

παίρνει απαράδεκτες τιμές θα πρέπει να ξαναδείς τον *δημιουργό* και τη *setYear* αφού μόνο εκεί ορίζεται ή αλλάζει η τιμή αυτού του μέλους.

Τώρα θα συζητήσουμε δύο προβλήματα που έχουμε με τις κλάσεις εξαιρέσεων όπως τις έχουμε τυποποιήσει.

Το *πρώτο πρόβλημα* έχει σχέση με τα αντικείμενα: Η εξαίρεση θα μας φέρει τη μέθοδο που εμφανίστηκε το πρόβλημα, το είδος του προβλήματος και τις τιμές που το προκάλεσαν –π.χ. «*αποπειράθηκες να δώσεις με τη setYear αρνητική τιμή* (–7) *στο dYear*»– αλλά δεν θα μας πει σε ποιο αντικείμενο έγιναν όλα αυτά. Σε ένα δοκιμαστικό προγραμματάκι που έχει 3 ή 4 αντικείμενα δεν υπάρχει πρόβλημα. Αν όμως, σε κάποιο πρόγραμμα, έχουμε 4793 αντικείμενα τύπου *Date* αυτή η πληροφορία δεν μας λέει και πολλά. Μια πρώτη σκέψη είναι να αντιγράφουμε στην εξαίρεση και το *κλειδί* (§15.5.1) του αντικειμένου που έχει το πρόβλημα (τέτοιο παράδειγμα θα δούμε αργότερα). Αλλά πολύ συχνά αυτό δεν είναι αρκετό. Σκέψου την περίπτωση που τα 4780 αντικείμενα από αυτά που λέγαμε παραπάνω, ανήκουν σε 2390 στοιχεία ενός πίνακα κλάσης *Employee* (δύο ημερομηνίες στο καθένα, §15.1). Προφανώς δεν μπορούμε να αποκλείσουμε την εμφάνιση της ίδιας ημερομηνίας περισσότερες από μία φορές. Αν εξοπλίσουμε κάθε αντικείμενο με ένα επιπλέον μέλος, μια **ταυτότητα αντικειμένου** (object identifier) που θα την αντιγράφουμε και στο αντικείμενο της εξαίρεσης λύνουμε το πρόβλημά μας.[5]

Το *δεύτερο* πρόβλημα μπορεί να το έχεις αντιμετωπίσει ήδη αν έγραψες κάπως μεγάλα προγράμματα: «Ναι, η εξαίρεση ρίχτηκε από τη συνάρτηση *myFunc*, αλλά η *myFunc* καλείται στο πρόγραμμά μου σε 37 διαφορετικές «διαδρομές» εκτέλεσης!» Αργότερα θα δώσουμε μια τεχνική για να μπορείς να έχεις ακριβέστερη πληροφορία.

Και στα δύο προβλήματα μπορεί να πάρεις σημαντική βοηθεια, στη φάση των δοκιμών, από ένα καλό **πρόγραμμα διόρθωσης λαθών** (debugger). Αλλά αν το πρόβλημα προκύψει όταν το πρόγραμμα είναι σε εκμετάλλευση, τα πράγματα δεν είναι τόσο εύκολα.

19.6 "class" ή "struct";

Όπως είδαμε πιο πριν, τίποτε δεν μας εμποδίζει να ορίσουμε οποιαδήποτε κλάση είτε με `struct` είτε με `class`. Θα πρέπει όμως να έχουμε κάποιον τρόπο για να αποφασίζουμε κατά περίπτωση πώς θα κάνουμε τον ορισμό μας.

Τι διαφέρει το "`class`" από το "`struct`"; Όπως είπαμε:

- Όταν δίνουμε ορισμό κλάσης με το "`class`" τότε όλα τα μέλη είναι *εσωτερικά*, υπάρχει δηλαδή αυτόματη δήλωση `private` για τα πάντα. Ό,τι θέλουμε να ανοίξουμε προς τα έξω θα πρέπει να δηλωθεί `public`.
- Όταν δίνουμε ορισμό κλάσης με το "`struct`" τότε όλα τα μέλη είναι *ανοικτά*, υπάρχει δηλαδή αυτόματη δήλωση `public` για τα πάντα. Ό,τι θέλουμε να "κρύψουμε" θα πρέπει να δηλωθεί `private`.

Θα μπορούσαμε λοιπόν να πούμε ότι η C++ μας δίνει ένα κριτήριο για να πάρουμε την αποφασή μας: Αν έχουμε να «κρύψουμε» κάτι δίνουμε ορισμό με "`class`" αλλιώς με "`struct`". Αυτή είναι περίπου η απάντηση στο πρόβλημά μας.[6]

- Η δήλωση με `struct` χρησιμοποιείται συνήθως για απλές δομές όπου δεν έχουμε να κρύψουμε κάτι και όλα είναι ανοικτά. Αυτή ήταν η χρήση της και στη C που δεν είχε κλάσεις.
- Όταν έχουμε περιοχές `private` και `public` χρησιμοποιούμε τη δήλωση με `class`.

---

[5] *Δεν* θα δώσουμε τέτοιο παράδειγμα.

[6] Να υπενθυμίσουμε ότι και η C# βλέπει τις δομές ως «*κλάσεις με όλα τα μέλη ανοικτά που δεν κληρονομούν ούτε κληρονομούνται.*»

Να το πούμε και χρησιμοποιώντας την αναλλοίωτη:[7]

- ***Για να χρησιμοποιήσεις* `struct` *θα πρέπει να έχεις οπωσδήποτε τετριμμένη αναλλοίωτη (*`true`*).***

Παράδειγμα 1

Για τη

```
struct complex
{
   double re;
   double im;
   complex( double rp = 0.0, double ip = 0.0 )
   {  re = rp;  im = ip;  };
}; // complex
```

τι αναλλοίωτη να γράψουμε;

$$(\texttt{-DBL\_MAX} \le re \le \texttt{DBL\_MAX}) \wedge (\texttt{-DBL\_MAX} \le im \le \texttt{DBL\_MAX})$$

Αυτό όμως ισχύει πάντοτε για κάθε τιμή τύπου `double` και όχι ειδικώς για τα μέλη της *complex*. Εδώ λοιπόν η αναλλοίωτη είναι `true`.

Αυτή όμως είναι αναγκαία αλλά όχι και ικανή συνθήκη για να επιλέξουμε τη `struct`.

Παράδειγμα 2

Θέλουμε να υλοποιήσουμε έναν τύπο συνόλων που στοιχεία τους θα είναι κεφαλαία γράμματα του λατινικού αλφαβήτου. Για την υλοποίηση επιλέγουμε τη χρήση *ψηφιοχάρτη*. Δηλαδή ένα σύνολο θα παριστάνεται με τα πρώτα 26 δυαδικά ψηφία μιας τιμής τύπου `long int` και:

- το δυαδικό ψηφίο 0 έχει τιμή 1 αν και μόνον αν το `'A'` ανήκει στο σύνολο,
- το δυαδικό ψηφίο 1 έχει τιμή 1 αν και μόνον αν το `'B'` ανήκει στο σύνολο,

  . . .
- το δυαδικό ψηφίο 25 έχει τιμή 1 αν και μόνον αν το `'Z'` ανήκει στο σύνολο.

Θα γράψουμε μια

```
class SetOfUCL
{
public:
// . . .
private:
   long int bitmap;
}; // SetOfUCL
```

ή μια `struct`;

Εδώ η αναλλοίωτη είναι `true`. Αλλά αν επιλέξουμε `struct` έχουμε το εξής πρόβλημα: Δίνουμε τη δυνατότητα στο πρόγραμμα που τη χρησιμοποιεί να χειρίζεται την τιμή του (μοναδικού) μέλους ενώ η κλάση γράφεται για να του δώσει τη δυνατότητα να χειρίζεται σύνολα και τα μέλη τους. Πώς αποφεύγεται κάτι τέτοιο; Με το να την κάνουμε `class`, να βάλουμε το `long int bitmap` («μυστικό» της υλοποίησης) σε περιοχή `private` και *να μην γράψουμε μεθόδους getBitmap ούτε setBitmap*.

Ας πούμε λοιπόν ότι: Αν μια κλάση

1. έχει αναλλοίωτη `true` (τα πάντα δεκτά) και
2. δεν υπάρχουν «μυστικά» της υλοποίησης, δηλαδή μέλη στα οποία δεν θέλουμε να δώσουμε άμεση πρόσβαση

---

[7] Στο (Lockheed-Martin 2005) ο *AV Rule* 66 λέει: "A class **should** be used to model an entity that maintains an invariant", δηλσδή: για μια οντότητα που διατηρεί μια αναλλοίωτη **πρέπει** να χρησιμοποιείται μοντέλο `class` (και όχι `struct`).

θα προτιμούμε να ομαδοποιούμε τα στοιχεία μας με μια δομή.

Πάντως, όταν ορίζουμε μια `struct` μπορεί να ορίζουμε δημιουργό (ή δημιουργούς), να ορίζουμε συναρτήσεις-μέλη και να επιφορτώνουμε τελεστές αν αυτό μας διευκολύνει στην ανάπτυξη της εφαρμογής.

Και ένα ακόμη ερώτημα σχετικό με τα παραπάνω. Ας πούμε ότι έχουμε μια κλάση *Circle* που κάθε αντικείμενό της παριστάνει έναν κύκλο και έχει τρία μέλη: τις συντεταγμένες του κέντρου *cXC*, *cYC* και την ακτίνα *cR*. Φυσικά για την ακτίνα έχουμε $cR \geq 0$. Για το κέντρο όμως δεν έχουμε περιορισμό, θα μπορούσε να είναι οπουδήποτε. Τι κάνουμε σε αυτήν την περίπτωση; Βάζουμε `private` την ακτίνα και `public` τις συντεταγμένες του κέντρου; Όχι! Ο μεταγλωττιστής δεν θα έχει αντίρρηση για κάτι τέτοιο αλλά γενικώς θα τηρούμε τον κανόνα:[8]

- ***Σε μια κλάση που υλοποιείται με*** `class` ***τα μέλη είναι σε περιοχή*** `private`**.**

19.7 Από τη "`struct GrElmn`" στην "`class GrElmn`"

Ας εφαρμόσουμε τώρα αυτά που μάθαμε –και μερικά άλλα– στην κλάση *GrElmn* που είδαμε στην §15.14:[9]

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

Ποια είναι η αναλλοίωτη της κλάσης; Ας ξεκινήσουμε με την:

$$0 < geANumber < geAWeight$$

και να παρατηρήσουμε τα εξής:

- Αν δηλώσουμε

  ```
  GrElmn a;
  ```

  τι τιμές θα έχουν τα *a*.*geANumber*, *a*.*geAWeight*; Μια λογική επιλογή είναι: *a*.*geANumber* `==` *a*.*geAWeight* `==` 0.
- Το μήκος του *geSymbol* είναι 1 (π.χ. "U") ή 2 (π.χ. "Ca"). Αυτό θα πρέπει να περιληφθεί στην αναλλοίωτη.
- Τα *geSymbol* και *geName* έχουν μονον λατινικά γράμματα ενώ το *geGrName* μόνον ελληνικά. Και αυτό θα πρέπει να φαίνεται στην αναλλοίωτη.
- Στον ατομικό αριθμό δεν θα πρέπει να βάλουμε κάποιο απόλυτο (και χαμηλότερο) άνω όριο; Κάτι σαν 105 ή 112 ας πούμε; Ε, τώρα τέτοιο όριο δεν υπάρχει, διότι κάθε τόσο οι πυρηνικοί φυσικοί συνθέτουν στο εργαστήριο και έναν νέο (ασταθή) πυρήνα. Για το πρόγραμμά μας, το μόνο όριο που υπάρχει είναι αυτό που βάζει το περιεχόμενο του αρχείου μας.

  Γράφουμε λοιπόν την εξής αναλλοίωτη:

---

[8] Σχετική σύσταση του (CERT 2009) η OBJ00: "Declare data members `private`". Στο (Lockheed-Martin 2005) ο *AV Rule 67* λέει: "`public` and `protected` data should only be used in `struct`s—not `class`es" (το "`protected`" θα το μάθουμε αργότερα).

[9] Πρόσεξε ότι την είχαμε γράψει εξ αρχής τηρώντας την ονοματολογική σύμβαση που δώσαμε παραπάνω

$$((0 < geANumber < geAWeight) \;||$$
$$(geANumber == 0 \;\&\&\; geAWeight \geq 0) \;||\; (geAWeight == 0 \;\&\&\; geANumber \geq 0))$$
$$\&\&\; (geSymbol \leq 2)$$

Δηλαδή: αν τα *geANumber*, *geAWeight* έχουν αμφότερα μη μηδενικές τιμές θα πρέπει να ισχύει η 0 < *geANumber* < *geAWeight*· αν το ένα από τα δύο έχει τιμή 0 το άλλο θα πρέπει να έχει μη αρνητική τιμή. Οι μηδενικές τιμές θα επιτρέπονται μόνον από τον (ερήμην) δημιουργό αλλά όχι από τις σχετικές *set*.

Αφήνουμε ως άσκηση τους ελέγχους για λατινικά στα *geSymbol* και *geName* και ελληνικά στο *geGrName*.

Αφού η αναλλοίωτη δεν είναι τετριμμένη θα πρέπει να έχουμε υλοποίηση:

```
class GrElmn
{// I: ((0 < geANumber < geAWeight) ||
 //     (geANumber == 0 && geAWeight ≥ 0) ||
 //     (geAWeight == 0 && geANumber ≥ 0))
 // && (geSymbol ≤ 2)
public:
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
// . . .
private:
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[4];
   char               geName[14];
   char               geGrName[14];
}; // GrElmn
```

Ας ξεκινήσουμε με έναν («2 σε 1») δημιουργό: Δηλώνεται ως

```
   GrElmn( int aan=0, float aaw=0,
           string as="", string anm="", string agn="" );
```

και ορίζεται

```
GrElmn::GrElmn( int aan, float aaw,
                string as, string anm, string agn )
{
   if ( aan < 0 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::negANumber, aan );
   if ( aaw < 0 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::negANumber, aaw );
// aan >= 0 && aaw >= 0
   if ( (aan != 0 && aaw != 0) && aan >= aaw )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::an_gt_aw,
                        aan, aaw );
// (aan == 0 && aaw >=  0) || (aaw == 0 && aan >= 0) ||
// (0 < aan < aaw)
   geANumber = aan;
   geAWeight = aaw;
// (geANumber == 0 && geAWeight >= 0) ||
// (geAWeight == 0 && geANumber >= 0) ||
// (0 < geANumber < geAWeight)
   if ( as.length() > 2 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::longSymbol,
                        as.c_str() );
   strncpy( geSymbol, as.c_str(), symbolSz-1 );
   geSymbol[symbolSz-1] = '\0';
   strncpy( geName, anm.c_str(), nameSz-1 );
   geName[nameSz-1] = '\0';
   strncpy( geGrName, agn.c_str(), grNameSz-1 );
   geGrName[grNameSz-1] = '\0';
} // GrElmn::GrElmn
```

Αλλά, όπως είπαμε, στις *setAWeight()* και *setANumber()* δεν επιτρέπουμε τιμή 0:

```
void GrElmn::setAWeight( float aaw )
```

```
{
   if ( aaw <= 0 )
      throw GrElmnXptn( "setAWeight",
                        GrElmnXptn::negAWeight, aaw );
// aaw > 0
   if ( geANumber >= aaw )
      throw GrElmnXptn( "setAWeight", GrElmnXptn::an_gt_aw,
                        geANumber, aaw );
// 0 <= geANumber < aaw
   geAWeight = aaw;
// 0 <= geANumber < geAWeight
} // GrElmn::setAWeight
```

Δηλαδή: δεν επιτρέπουμε –με τη *setAWeight()*– να βάλουμε στο *geAWeight* τιμή 0. Το ίδιο πρέπει να κάνουμε και στην *setANumber()*:

```
void GrElmn::setANumber( int aan )
{
   if ( aan <= 0 )
      throw GrElmnXptn( "setANumber",
                        GrElmnXptn::negANumber, aan );
// aan > 0
   if ( geAWeight != 0 && geAWeight <= aan )
      throw GrElmnXptn( "setANumber",
                        GrElmnXptn::an_gt_aw, aan, geAWeight );
// (aan > 0) && (geAWeight != 0 ⇒ 0 < aan < geAWeight)
   geANumber = aan;
// (geANumber > 0) &&
// (geAWeight != 0 ⇒ 0 < geANumber < geAWeight)
} // GrElmn::setANumber
```

Παρατήρηση: ▶

Όπως θα δεις στη συνέχεια, για τα δύο προγράμματα της §15.14 που θα μετατρέψουμε, δεν θα χρειαστούμε μεθόδους "*set*", εκτός από την *setGrName*. Ας σκεφθούμε όμως πόσο καλό είναι γενικώς το να έχουμε μια *setANumber()*.

Κατ' αρχάς να παρατηρήσουμε ότι δεν είναι δυνατόν να έχουμε δύο στοιχεία που να έχουν ίδιο *geANumber* ή *geSymbol* ή *geName* ή *geGrName*. Όλα αυτά τα μέλη είναι **υποψήφια κλειδιά** (candidate keys) για οποιαδήποτε συλλογή αντικειμένων κλάσης *GrElmn*. Ως κλειδί μας συμφέρει να επιλέξουμε το *geANumber* αφού είναι το πιο απλό. Οι μέθοδοι "*set*" για τροποποίηση των τιμών των *geSymbol*, *geName* και *geGrName* είναι μάλλον απαραίτητες αφού οι τιμές είναι κείμενα και υπάρχουν πολλές πιθανότητες για λάθη –ακόμη και ορθογραφικά– που θα πρέπει να διορθωθούν.

Η *setANumber()* όμως μπορεί να είναι ακόμη και επικίνδυνη:

- Αφού το *geANumber* είναι κλειδί, σε οποιαδήποτε συλλογή αντικειμένων *GrElmn* –όπως το περιεχόμενο του `elementsGr.dta`– υπάρχει το πολύ ένα αντικείμενο με κάποιο συγκεκριμένο κλειδί. Άρα:
  - Όταν εισάγουμε ένα αντικείμενο στη συλλογή θα πρέπει να ελέγχουμε αν υπάρχει άλλο αντικείμενο με το κλειδί του εισαγόμενου.
  - Αν γράψουμε μια *setANumber()* αυτή θα πρέπει να απενεργοποιείται όταν το αντικείμενο εισάγεται στη συλλογή.
- Αλλαγή της τιμής του *geANumber* σημαίνει αλλαγή στοιχείου· θα πρέπει να ακολουθείται από αλλαγές όλων των άλλων μελών. Είναι απλούστερο και ασφαλέστερο να χρησιμοποιείς τον δημιουργό. Αν, ας πούμε έχεις δώσει:

```
   GrElmn oneElmn( 20, 40.08, "Ca", "Calcium", "Ασβέστιο" );
```

και στη συνέχεια θέλεις να αποθηκεύσεις τα δεδομένα για τον άνθρακα δώσε:

```
   oneElmn = GrElmn( 6, 12.011, "C", "Carbon", "Άνθρακας" );
```

Αν κάνεις τις αλλαγές με τις "*set*" όλο και κάτι μπορεί να ξεφύγει.◀

Το πρώτο πρόγραμμα –της §15.14.1– χρησιμοποιεί μόνον δύο συναρτήσεις, τις

- *GrElmn_copyFromElmn()* και
- *GrElmn_save()*.

Βλέποντας κανείς τις

```
   fv.geANumber = a.eANumber;
   fv.geAWeight = a.eAWeight;
   strcpy( fv.geSymbol, a.eSymbol );
   strcpy( fv.geName, a.eName );
   fv.geGrName[0] = '\0';
```

θα σκεφθεί ότι για τη μετατροπή του προγράμματός μας στα νέα δεδομένα χρειαζόμαστε 5 μεθόδους "*set*".

Αλλά οι "*set*" γράφονται για να κάνουν έλεγχο στις τιμές που περνούν στα μέλη του αντικειμένου. Στην περίπτωσή μας, που παίρνουμε τα δεδομένα μας από ένα αρχείο που είναι ελεγμένο (είναι άλλωστε και μη μορφοποιημένο) αυτοί οι έλεγχοι είναι χάσιμο χρόνου. Θα δώσουμε λοιπόν μια άλλη λύση, πιο καλή, με έναν δεύτερο δημιουργό για την κλάση μας:

```
GrElmn::GrElmn( const Elmn& rhs )
{
   geANumber = rhs.eANumber;
   geAWeight = rhs.eAWeight;
   strcpy( geSymbol, rhs.eSymbol );
   strcpy( geName, rhs.eName );
   geGrName[0] = '\0';
} // GrElmn::GrElmn
```

Αυτός ο δημιουργός τροφοδοτείται με ένα αντικείμενο κλάσης *Elmn* και δημιουργεί ένα αντικείμενο κλάσης *GrElmn*· είναι ένας *δημιουργός μετατροπής*, όπως θα μάθουμε αργότερα. Έτσι, στο πρόγραμμά μας θα έχουμε:

```
   // Διάβασε μια εγγραφή
      Elmn oneElmn;
      int k( 0 );
      Elmn_load( oneElmn, bin );
      while ( !bin.eof() )
      {
      // Αντίγραψε την εγγραφή
         GrElmn oneGrElmn( oneElmn );
      // Γράψε τη νέα εγγραφή
         oneGrElmn.save( bout );
         ++k;
      // Διάβασε την επόμενη εγγραφή
         Elmn_load( oneElmn, bin );
      } // while
```

Η *GrElmn_save* θα μας δώσει την:

```
void GrElmn::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw GrElmnXptn( "save", GrElmnXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&geANumber),
               sizeof(geANumber) );
   bout.write( reinterpret_cast<const char*>(&geAWeight),
               sizeof(geAWeight) );
   bout.write( geSymbol, symbolSz );
   bout.write( geName, nameSz );
   bout.write( geGrName, grNameSz );
   if ( bout.fail() )
      throw GrElmnXptn( "save", GrElmnXptn::cannotWrite );
} // GrElmn::save
```

Για το δεύτερο πρόγραμμα –της §15.14.2– πέρα από τη *GrElmn::save()*, θα πρέπει να μετατρέψουμε σε μεθόδους και τις *GrElmn_load()*, *GrElmn_display()*, *GrElmn_setGrName()*. Η μετατροπή δεν έχει δυσκολίες:

```
void GrElmn::load( istream& bin )
```

```
{
   bin.read( reinterpret_cast<char*>(&geANumber),
             sizeof(geANumber) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&geAWeight),
                sizeof(geAWeight) );
      bin.read( geSymbol, symbolSz );
      bin.read( geName, nameSz );
      bin.read( geGrName, grNameSz );
      if ( bin.fail() )
         throw GrElmnXptn( "load", GrElmnXptn::cannotRead );
   }
} // GrElmn::load

void GrElmn::display( ostream& tout ) const
{
   tout << "atomic number: " << geANumber << endl
        << "atomic weight: " << geAWeight << endl
        << "symbol: " << geSymbol << endl
        << "name: " << geName << endl
        << "greek name: " << geGrName << endl;
} // GrElmn::display

void GrElmn::setGrName( string newGrName )
{
   strncpy( geGrName, newGrName.c_str(), grNameSz-1 );
   geGrName[grNameSz-1] = '\0';
} // GrElmn::setGrName
```

Στη *writeRandom()* υπάρχει η εντολή:

```
bout.seekp( (a.geANumber-1)*GrElmn::saveSize );
```

Φυσικά, η συνάρτηση αυτή δεν έχει δικαίωμα πρόσβασης στο μέλος *geANumber* και θα χρειαστούμε τη σχετική "*get*":

```
   unsigned short int getANumber() const {  return geANumber;  }
```

Έτσι, στη *writeRandom* θα έχουμε:

```
   bout.seekp( (a.getANumber()-1)*GrElmn::saveSize );
```

Αν μετατρέψουμε σε μέθοδο και την *GrElmn_writeToTable()*, που είδαμε στην §15.14.2, παίρνουμε:

```
void GrElmn::writeToTable( ostream& tout ) const
{
   tout << geANumber << '\t' << geGrName << " (" << geName
        << ")\t" << geSymbol << '\t' << geAWeight << endl;
} // GrElmn::writeToTable
```

Τελικώς θα έχουμε στο `GrElmn.h`:

```
#ifndef _GRELMN_H
#define _GRELMN_H

#include <fstream>
#include <string>

using namespace std;

struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn

class GrElmn
{
```

```
public:
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   GrElmn( int aan=0, float aaw=0,
           string as="", string anm="", string agn="" );
   GrElmn( const Elmn& rhs );
   unsigned short int getANumber() const {  return geANumber;  }
   void setAWeight( float aaw );
   void setGrName( string newGrName );
   void save( ostream& bout ) const;
   void load( istream& bin );
   void display( ostream& tout ) const;
   void writeToTable( ostream& tout ) const;
private:
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

struct GrElmnXptn
{
   enum { negANumber, negAWeight, an_gt_aw, longSymbol,
          fileNotOpen, cannotWrite, cannotRead };
   char  funcName[100];
   int   errorCode;
   float errFltVal1;
   float errFltVal2;
   char  errStrVal[100];
   GrElmnXptn( const char* mn, int ec, float ev1, int ev2=0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errFltVal1 = ev1;  errFltVal2 = ev2;  }
   GrElmnXptn( const char* mn, int ec, const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // GrElmnXptn

#endif // _GRELMN_H
```

και στο GrElmn.cpp:

```
#ifndef _GRELMN_CPP
#define _GRELMN_CPP

#include <fstream>
#include "GrElmn.h"

//========================= δημιουργοί ==========================
GrElmn::GrElmn( int aan, float aaw,
                string as, string anm, string agn )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
GrElmn::GrElmn( const Elmn& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
//========================= setters ============================
void GrElmn::setAWeight( float aaw )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::setGrName( string newGrName )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
//====================== άλλες μέθοδοι =======================
void GrElmn::save( ostream& bout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::load( istream& bin )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::display( ostream& tout ) const
```

```
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::writeToTable( ostream& tout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _GRELMN_CPP
```

Η κλάση αυτή έχει όλα όσα χρειάζονται για να (ξανα)γράψουμε τα προγράμματα της §15.14. Δεν χρειαζόμαστε "*get*" και "*set*"; Χρειαζόμαστε! Μια κλάση γράφεται με αφορμή μια συγκεκριμένη εφαρμογή, αλλά συνήθως την εξοπλίζουμε ώστε να μπορεί να χρησιμοποιηθεί και αλλού. Στο επόμενο κεφάλαιο θα δούμε μια σχετική «συνταγή».

19.8 Μια Κλάση για Μπαταρίες[10]

Με δύο παραδείγματα μετάβασης από απλή δομή σε κλάση είδαμε, έστω και ακροθιγώς, μερικά βασικά πράγματα για τον αντικειμενοστρεφή προγραμματισμό. Τώρα θα δούμε άλλο ένα παράδειγμα όπου γράφουμε μια απλή κλάση από την αρχή (χωρίς να μετατρέψουμε κάτι που προϋπάρχει).

Το πρόβλημα:

Γράψε μια κλάση για την παράσταση μιας μπαταρίας. Ένα αντικείμενο-μπαταρία ξέρει την τάση του (σε volts), πόση ενέργεια (μέγιστη) μπορεί να αποθηκεύσει (σε joules) και πόσο είναι το τρέχον απόθεμα ενέργειας που έχει αποθηκευμένη (σε joules).

Πέρα από όποιες άλλες μεθόδους που θεωρείς απαραίτητες, να περιλάβεις, οπωσδήποτε και τις παρακάτω:

α) powerDevice(): Θα τροφοδοτείται με τον χρόνο t (σε sec), που θέλουμε να τροφοδοτεί μια συσκευή και το ρεύμα i (σε amperes) που τραβάει, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t). Θα επιστρέφει τιμή:

- `true` *αν η ενέργεια (τρέχουσα τιμή) που έχει η μπαταρία είναι αρκετή για να τροφοδοτήσει τη συσκευή. Στην περίπτωση αυτή θα αφαιρεί από την τρέχουσα τιμή της ενέργειας την ενέργεια που τράβηξε η συσκευή.*
- `false` *αν η ενέργεια δεν είναι αρκετή.*

β) maxTime(): Θα τροφοδοτείται με το ρεύμα i (σε amperes) που τραβάει μια συσκευή, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t) και θα επιστρέφει τον χρόνο (sec) που μπορεί να τροφοδοτεί τη συσκευή. Δεν αλλάζει το απόθεμα ενέργειας της μπαταρίας.

γ) recharge(): Θα επαναφορτίζει τη μπαταρία, δηλαδή θα βάζει το ενεργειακό απόθεμα στη μέγιστη τιμή του.

Γράψε πρόγραμμα που θα δοκιμάζει την κλάση που έγραψες: Θα δημιουργεί μπαταρία 12 volts με μέγιστη δυνατότητα αποθήκευσης 5×10^6 joules. Στην αρχή θα τροφοδοτεί ένα λαμπτήρα που τραβάει 4 amperes για 15 min. Στη συνέχεια θα ερωτάται για τον χρόνο που μπορεί να τροφοδοτήσει μια συσκευή που τραβάει 8 amperes. Η ερώτηση θα υποβάλλεται ξανά αφού προηγουμένως η μπαταρία επαναφορτισθεί.

Η αναλλοίωτη της κλάσης δεν μπορεί να είναι άλλη από την:

$$0 < bVoltage \;\&\&\; 0 < bMaxEnergy \;\&\&\; 0 \le bEnergy \le bMaxEnergy$$

Άρα ο ορισμός της κλάσης μας θα είναι περίπου ως εξής:

```
class Battery
{// I: 0 < bVoltage && 0 < bMaxEnergy && 0 ≤ bEnergy ≤ bMaxEnergy
public:
// ...
private:
   double bVoltage;   // volts
   double bMaxEnergy; // joules
   double bEnergy;    // joules
}; // Battery
```

[10] Από μια άσκηση του (Mansfield & Antonakos 1997).

Όπως βλέπεις, στα ονόματα τηρούμε τον κανόνα που είπαμε στην §19.1.4.

Η αντίστοιχη κλάση εξαιρέσεων είναι:

```
struct BatteryXptn
{
   enum { . . . };
   char   funcName[100];
   int    errorCode;
   double errorValue;
   BatteryXptn( const char* mn, int ec, double ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BatteryXptn
```

Για να αποφασίσουμε τι τιμές θα βάζει ο ερήμην δημιουργός μελετούμε το πρόβλημά μας. Εδώ βέβαια τα πράγματα είναι πολύ απλά: αφού μας ζητείται να δηλώσουμε μια 12βολτη μπαταρία με μέγιστη ενέργεια 5×10^6 *joules*, θα επωφεληθούμε και θα γράψουμε («2 σε 1») και τον δημιουργό με αρχικές τιμές. Θα δηλώσουμε μέσα στον ορισμό της κλάσης

```
   Battery( double v = 12, double me = 5e6 );
```

και θα ορίσουμε:

```
Battery::Battery( double v, double me )
{
   if ( v <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::voltageErr, v );
   if ( me <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::energyErr, me );
   bVoltage = v;
   bMaxEnergy = me;
   bEnergy = bMaxEnergy;
} // Battery::Battery
```

Όπως βλέπεις, αν παραβιασθεί η αναλλοίωτη από τις τιμές που καθορίζονται θα πρέπει να ρίχνονται οι κατάλληλες εξαιρέσεις. Πρέπει βέβαια να εισαχθούν στον `enum` της *BatteryXptn* τα *voltageErr* και *energyErr*.

19.8.1 Μέθοδοι "*get*", "*set*"

Χρειαζόμαστε μεθόδους "*get*" για να βλέπουμε τις τιμές μελών; Ναι, διότι θα γράψουμε μεν τη *maxTime()* (που έχει σαφώς περισσότερο νόημα από μια *getEnergy()* που πιθανότατα θα γράφαμε όπως κάναμε στα προηγούμενα παραδείγματα), αλλά θα χρειαστούμε μεθόδους για τα άλλα μέλη. Πού θα μας χρειαστούν; Ας πούμε ότι σε μια συνάρτηση έρχεται μια παράμετρος κλάσης *Battery*· πώς θα μάθουμε τα σταθερά χαρακτηριστικά της μπαταρίας; Γράφουμε λοιπόν τις:

```
   double getVoltage()  const {  return bVoltage;  };
   double getMaxEnergy()  const {  return bMaxEnergy;  };
```

Μεθόδους "*set*" χρειαζόμαστε; Θα πρέπει να κάνουμε σαφές ότι δεν θα πρέπει να υπάρχουν μέθοδοι που να αλλάζουν τις τιμές των *bVoltage* και *bMaxEnergy*, αφού αυτά παριστάνουν *σταθερά χαρακτηριστικά* της μπαταρίας που της αποδίδονται με την κατασκευή (δημιουργία) της. Η μόνη που έχει φυσικό νόημα είναι η *recharge()*, που θα γράψουμε στη συνέχεια.

19.8.2 Μέθοδος *powerDevice()*

«Θα τροφοδοτείται με τον χρόνο t (σε sec), που θέλουμε να τροφοδοτεί μια συσκευή και το ρεύμα i (σε amperes) που τραβάει, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t). Θα επιστρέφει τιμή:

- `true` *αν η ενέργεια (τρέχουσα τιμή) που έχει η μπαταρία είναι αρκετή για να τροφοδοτήσει τη συσκευή. Στην περίπτωση αυτή θα αφαιρεί από την τρέχουσα τιμή της ενέργειας την ενέργεια που τράβηξε η συσκευή.*
- `false` *αν η ενέργεια δεν είναι αρκετή.*»

Η *powerDevice()* θα αλλάζει την τιμή του αντικειμένου και θα επιστρέφει και μια τιμή τύπου `bool`. Δηλαδή, αν ήταν εξωτερική συνάρτηση θα ήταν κάπως έτσι:

```
??? Battery_powerDevice( Battery& ab,
                         double t, double i, bool& ok )
```

Αυτή έχει μια μεταβαλλόμενη παράμετρο (*ab*) και μια εξερχόμενη (*ok*). Οι κανόνες μας (§13.9) λένε ότι θα πρέπει να είναι `void`. Η μέθοδος θα είναι:

```
void Battery::powerDevice( double t, double i, bool& ok )
```

Οι προδιαγραφές της:

Προϋπόθεση: $t \geq 0$ && $i \geq 0$

Απαίτηση: (*ok* == (*bVoltage**i*t* <= $bEnergy_{\text{αρχική}}$)) &&

((*ok* ⇒ ($bEnergy_{\text{τελική}}$ == $bEnergy_{\text{αρχική}}$ - *bVoltage**i*t*))

Αν παραβιάζεται η προϋπόθεση, δηλαδή αν $t < 0$ ή $i < 0$ τότε ρίχνουμε εξαίρεση. *Αν η bVoltage*i*t <= $bEnergy_{\text{αρχική}}$ δεν ισχύει δεν ρίχνουμε εξαίρεση.* Απλώς το πρόγραμμα-πελάτης πριν προχωρήσει σε οποιαδήποτε χρήση του αντικειμένου θα πρέπει να ελέγξει την τιμή της *ok*.

```
void Battery::powerDevice( double t, double i, bool& ok )
{
   if ( t < 0 )
      throw BatteryXptn( "powerDevice", BatteryXptn::timeErr, t );
   if ( i < 0 )
      throw BatteryXptn( "powerDevice",
                         BatteryXptn::currentErr, i );
   double reqEnergy( bVoltage*i*t );
   ok = reqEnergy <= bEnergy;
   if ( ok )  bEnergy -= reqEnergy;
} // Battery::powerDevice
```

Φυσικά, θα πρέπει να προσθέσουμε στην κλάση *BatteryXptn* τις δύο νέες σταθερές (*timeErr*, *currentErr*):

```
   enum { voltageErr, energyErr, timeErr, currentErr };
```

19.8.3 Μέθοδος *maxTime()*

«*Θα τροφοδοτείται με το ρεύμα i (σε amperes) που τραβάει μια συσκευή, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t) και θα επιστρέφει τον χρόνο (σε sec) που μπορεί να τροφοδοτεί τη συσκευή. Δεν αλλάζει το απόθεμα ενέργειας της μπαταρίας.*»

Προδιαγραφές:

Προϋπόθεση: $i > 0$

Απαίτηση: *maxTime*(*i*) == *bEnergy*/(*bVoltage***i*)

Θα μπορούσαμε να γράψουμε την εξής μέθοδο:

```
double Battery::maxTime( double i )  const
{
   if ( i <= 0 )
      throw BatteryXptn( "maxTime", BatteryXptn::currentErr, i );
   return bEnergy/(bVoltage*i);
} // Battery::maxTime
```

19.8.4 Μέθοδος *reCharge*

«Θα επαναφορτίζει τη μπαταρία, δηλαδή θα βάζει το ενεργειακό απόθεμα στη μέγιστη τιμή του.»

Προδιαγραφές:

Προϋπόθεση: `true`

Απαίτηση: *bEnergy* `==` *bMaxEnergy*

```
void Battery::reCharge()
{
   bEnergy = bMaxEnergy;
} // Battery::reCharge
```

19.8.5 Η Κλάση μας Τελικώς

Στο αρχείο `Battery.h` έχουμε:

```
#ifndef _BATTERY_H
#define _BATTERY_H

#include <string>

using namespace std;

class Battery
{// I: 0 < bVoltage && 0 < bMaxEnergy &&
 //    0 <= bEnergy <= bMaxEnergy
public:
   Battery( double v = 12, double me = 5e6 );
   double getVoltage()  const {  return bVoltage;  };
   double getMaxEnergy()  const {  return bMaxEnergy;  };
   void powerDevice( double t, double i, bool& ok );
   double maxTime( double i )  const;
   void reCharge();
private:
   double bVoltage;   // volts
   double bMaxEnergy; // joules
   double bEnergy;    // joules
}; // Battery

struct BatteryXptn
{
   enum { voltageErr, energyErr, timeErr, currentErr };
   char   funcName[100];
   int    errorCode;
   double errorValue;
   BatteryXptn( const char* mn, int ec, double ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BatteryXptn

#endif // _BATTERY_H
```

και στο `Battery.cpp`:

```
#ifndef _BATTERY_CPP
#define _BATTERY_CPP

#include <fstream>
#include "Battery.h"

Battery::Battery( double v, double me )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Battery::powerDevice( double t, double i, bool& ok )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
double Battery::maxTime( double i )  const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

```
void Battery::reCharge()
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _BATTERY_CPP
```

19.8.6 Το Πρόγραμμα

Ας δούμε τώρα πώς θα είναι το πρόγραμμά μας:

«Γράψε πρόγραμμα που θα δοκιμάζει την κλάση που έγραψες: Θα δημιουργεί μπαταρία 12 volts με μέγιστη δυνατότητα αποθήκευσης $5 \cdot 10^6$ joules. Στην αρχή θα τροφοδοτεί έναν λαμπτήρα που τραβάει 4 amperes για 15 min. Στη συνέχεια θα ερωτάται για τον χρόνο που μπορεί να τροφοδοτήσει μια συσκευή που τραβάει 8 amperes. Η ερώτηση θα υποβάλλεται ξανά αφού προηγουμένως η μπαταρία επαναφορτισθεί.»

Μπορούμε να δημιουργήσουμε τη ζητούμενη μπαταρία είτε με την

```
   Battery btr;
```

είτε με την

```
   Battery btr( 12, 5e6 );
```

Στη συνέχεια τροφοδοτούμε τον λαμπτήρα με την (το πρώτο όρισμα θα πρέπει να είναι σε *sec*):

```
   btr.powerDevice( 15*60, 4, ok );
```

Παίρνουμε τον επιτρεπόμενο χρόνο τροφοδοσίας με την:

```
   cout << "μπορεί να τροφοδοτήσει με 8 A επί " << btr.maxTime(8)
        << " sec" << endl;
```

Τέλος, επαναφορτίζουμε και ξαναζητούμε τον χρόνο:

```
   btr.reCharge();
   cout << "μπορεί να τροφοδοτήσει με 8 A επί " << btr.maxTime(8)
        << " sec" << endl;
```

Φυσικά, θα πρέπει να βάλουμε και εντολές που συλλαμβάνουν και διαχειρίζονται τις εξαιρέσεις. Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <string>
#include "Battery.h"
#include "Battery.cpp"

using namespace std;

int main()
{
   bool ok;

   try
   {
      Battery btr( 12, 5e6 );

      btr.powerDevice( 15*60, 4, ok );
      cout << "μπορεί να τροφοδοτήσει με 8 A επί "
           << btr.maxTime(8) << " sec" << endl;

      btr.reCharge();
      cout << "μπορεί να τροφοδοτήσει με 8 A επί "
           << btr.maxTime(8) << " sec" << endl;
   }
   catch ( BatteryXptn& xpt )
   {
      switch ( xpt.errorCode )
      {
         case BatteryXptn::voltageErr:
            cout << xpt.funcName << ": Λάθος τάση ("
```

```
                << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::energyErr:
            cout << xpt.funcName << ": Λάθος ενέργεια ("
                 << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::timeErr:
            cout << xpt.funcName << ": Λάθος χρόνος ("
                 << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::currentErr:
            cout << xpt.funcName << ": Λάθος ρεύμα ("
                 << xpt.errorValue << ')' << endl;    break;
         default:
              cout << xpt.funcName
                   << ": Μη αναμενόμενη εξαίρεση" << endl;
      } // switch
   } // catch
} // main
```

19.9 Τι (Πρέπει Να) Έμαθες στο Κεφάλαιο Αυτό

- Οι κλάσεις είναι τύποι δεδομένων που οι μεταβλητές τους (*αντικείμενα*) έχουν τη δυνατότητα να ελέγχουν το πώς τις διαχειρίζεται το περιβάλλον (πρόγραμμα) μέσα στο οποίο υπάρχουν και δρουν και να μην επιτρέπουν την οποιαδήποτε «κακομεταχείριση».
- Κάθε αντικείμενο έχει τα *μέλη* του και τις *συναρτήσεις-μέλη* ή *μεθόδους* του. Τα μέλη είναι συνήθως *κρυμμένα* από το περιβάλλον. Οι μέθοδοι έχουν πρόσβαση σε όλα τα μέλη του αντικειμένου.
- *Η διαχείριση κάθε αντικειμένου γίνεται μέσω μεθόδων* που έχει το κάθε αντικείμενο. *Οι μέθοδοι περιγράφουν και το πώς μπορεί να δράσει* το κάθε αντικείμενο. Οι μέθοδοι του κάθε αντικειμένου καθορίζονται όταν ορίζεται η κλάση στην οποίαν ανήκει.
- Οι τιμές των μελών κάθε αντικειμένου πληρούν μια συνθήκη που είναι η *αναλλοίωτη* της κλάσης.
- Στον ορισμό της κλάσης –και σε μορφή συνάρτησης που ονομάζεται *δημιουργός* ή *κατάσκευαστής*– της κλάσης μπορεί να περιλαμβάνονται οδηγίες για το πώς δημιουργείται ένα αντικείμενο.

Στη C++:

- Μια κλάση ορίζεται ως `struct` (τα πάντα ανοικτά) ή ως `class` (τα πάντα κρυμμένα). Και στις δύο περιπτώσεις μπορείς να καθορίζεις τι θα είναι τελικώς ανοικτό ή κρυμμένο καθορίζοντας περιοχές `public` και `private`.
- Ο ορισμός της κλάσης είναι ταυτοχρόνως και ορισμός ενός *ονοματοχώρου με το όνομα της κλάσης*.
- Τα μέλη δηλώνονται όπως οι μεταβλητές. Οι μέθοδοι δηλώνονται και ορίζονται όπως οι συναρτήσεις.
- Οι μέθοδοι ενός αντικειμένου έχουν πρόσβαση προς τα μέλη αντικειμένων της ίδιας κλάσης.

Αρχές καλού προγραμματισμού:

- Οι μέθοδοι κάθε κλάσης καθορίζονται από τις ανάγκες του *πελάτη* δηλαδή του προγράμματος που τη χρησιμοποιεί. (Αυτόν τον κανόνα θα τον αλλάξουμε κάπως στη συνέχεια.)
- Μαζί με κάθε κλάση ορίζουμε και αντίστοιχη *κλάση εξαιρέσεων*. Σε περίπτωση απόπειρας «κακομεταχείρισης» (παραβίασης της αναλλοίωτης) ενός αντικειμένου η «υπεύθυνη» μέθοδος ρίχνει εξαίρεση αυτής της κλάσης.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

19-1 Είδαμε ότι μπορούμε να γράψουμε με τρεις διαφορετικούς αλλά ισοδύναμους τρόπους:

```
struct Date
{
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

```
class Date
{
   unsigned int dYear;
// . . .
public:
   Date( . . . );
// . . .
}; // Date
```

```
class Date
{
public:
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

Αλλά, με αυτά που είδαμε στη συνέχεια, μάλλον υπάρχουν και άλλοι ισοδύναμοι τρόποι.

1

```
struct Date
{
public:
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

2

```
class Date
{
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

3

```
struct Date
{
   unsigned int dYear;
// . . .
public:
   Date( . . . );
// . . .
}; // Date
```

Ισοδύναμοι με τους τρεις πρώτους είναι:

- Ο 1
- Ο 2
- Οι 1, 3
- Όλοι

19-2 Έστω ότι γράφουμε μια μέθοδο, *getDayOfWeek*, για τη *Date*, που επιστρέφει την ημέρα εβδομάδας για τη συγκεκριμένη ημερομηνία· δηλαδή μια τιμή τύπου:

```
typedef enum { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday }  WeekDay;
```

Η δήλωσή της θα είναι:

- `WeekDay getDayOfWeek( Date d ) const;`
- `WeekDay getDayOfWeek( Date d );`
- `WeekDay getDayOfWeek() const;`
- `WeekDay getDayOfWeek();`
- `void getDayOfWeek( WeekDat& wd ) const;`

Β Ομάδα

19-3 Γράψε συνάρτηση *isValidDate* που θα τροφοδοτείται με τρεις ακέραιους *ay*, *am* και *ad* και θα επιστρέφει `true` αν αποτελούν έγκυρη ημερομηνία (αλλιώς `false`). Θα μπορούσαμε να την κάνουμε μέθοδο της της *Date*;
Υπόδ.: χρησιμοποίησε τον δημιουργό της *Date*.

19-4 Με «μπούσουλα» τη μετατροπή της *GrElmn*, μετάτρεψε σε κλάση τη δομή *Elmn*. Γράψε πρόγραμμα που θα δημιουργεί το αρχείο `elements.dta`. Προσπάθησε να βοηθήσεις τον χρήστη όσο πιο πολύ μπορείς ώστε να μην κάνει λάθη όταν κάνει εισαγωγή των δεδομένων. Ψάξε στο διαδίκτυο να βρεις πίνακα των χημικών στοιχείων στα αγγλικά για να κάνεις τα πειράματά σου.

Γ Ομάδα

19-5 *Με οδηγό τη λύση του προβλήματος της μπαταρίας λύσε το παρακάτω πρόβλημα:*

Θέλουμε μια κλάση:

```
class Piscina...
```

για να παριστάνουμε μια (θερμαινόμενη) πισίνα. Υποθέτουμε ότι η πισίνα είναι σχήματος ορθογωνίου παραλληλεπιπέδου και έχουμε μήκος (*l*), πλάτος (*w*) και βάθος (*d*) σε *m*. Οι πάγιες διαστάσεις είναι 6×4×2 αλλά μπορεί να είναι και άλλες (ό,τι θέλει ο πελάτης). Ένα χαρακτηριστικό της κατάστασης της πισίνας είναι το ύψος *h* (σε *m*) του νερού, που δεν μπορεί να είναι μεγαλύτερο από $d - 0.3$. Ο όγκος του νερού στην πισίνα σε m^3, είναι $l \cdot w \cdot h$. Ένα άλλο χαρακτηριστικό είναι η θερμοκρασία θ του νερού σε °*C*.

Η κλάση θα έχει ακόμη έναν ή περισσότερους δημιουργούς και τις εξής μεθόδσυς που θα πρέπει να γράψεις εσύ:

- *fill()*: Θα τροφοδοτείται με δύο πραγματικούς αριθμούς
  - *dh*, που θα παριστάνει την αύξηση του ύψους του νερού σε *m*,
  - *ta*, που θα παριστάνει τη θερμοκρασία (σε °*C*) του νερού που θα προστεθεί

  και θα επιστρέφει τον όγκο νερού που πρέπει να προσθέσουμε για να ανέβει το ύψος του νερού κατά *dh*. Θυμίσου ότι το ύψος του νερού δεν μπορεί να γίνει μεγαλύτερο από $d - 0.3$ *m*. Η θερμοκρασία του νερού μετά την πρόσθεση θα είναι: $(\theta * h + ta * dh) / (h + dh)$.
- *empty()*: Θα τροφοδοτείται με έναν πραγματικό αριθμό *dh* που θα παριστάνει την μείωση του ύψους του νερού σε *m* και θα επιστρέφει τον όγκο νερού που πρέπει να αδειάσουμε για να κατέβει το ύψος του νερού κατά *dh*. Φυσικά το ύψος του νερού δεν μπορεί να γίνει μικρότερο από 0.
- *heat()*: Θα τροφοδοτείται με έναν πραγματικό αριθμό *dt* που θα παριστάνει την επιθυμητή αύξηση της θερμικρασίας του νερού σε βαθμούς *C* και θα επιστρέφει την απαιτούμενη ενέργεια σε *cal*. Αυτή υπολογίζεται ως $V \cdot \rho \cdot c \cdot dt$ όπου *V* ο όγκος του νερού σε m^3, $\rho = 10^3\ kgr/m^3$ η πυκνότητα του νερού και $c = 10^3\ cal \cdot kgr^{-1} \cdot grad^{-1}$ η ειδική θερμότητα του νερού. Φυσικά, η θερμοκρασία του νερού δεν μπορεί να υπερβεί τους 100° *C*.

Γράψε πρόγραμμα όπου θα δηλώνονται δύο πισίνες: μια με τις πάγιες διαστάσεις και μια 5×3×1.2. Θα τις γεμίζει μέχρι τη μέση (1 και 0.6 *m* αντιστοίχως) με νερό θερμοκρασίας 10° *C* και θα θερμαίνει το νερό κατά 15° *C*. Μετά θα τις γεμίζει μέχρι το μέγιστο επιτρεπόμενο σημείο με νερό ίδιας θερμοκρασίας με αυτό που υπάρχει στην πισίνα. Τέλος, θα μας δίνει το νερό (σε m^3) και την ενέργεια (σε *cal*) που χρειάστηκε η κάθε μια.

project

3

Φοιτητές και Μαθήματα

Περιεχόμενα:

Prj03.1 Το Πρόβλημα

Μας δίνονται

- *ένα μη-μορφοποιημένο (binary) αρχείο με όνομα* **`gCourses.dta`** *με περιεχόμενο άγνωστο πλήθος αντικειμένων τύπου:*

```
class SylCourse
{
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
// . . .
private:
   char         cCode[cCodeSz];   // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   char         cPrereq[cCodeSz]; // προαπαιτούμενο
}; // SylCourse
```

που, όπως φαίνεται, περιγράφει ένα μάθημα προγράμματος σπουδών.

- *ένα μορφοποιημένο (text) αρχείο, με όνομα* **enrllmnt.txt**, *που περιέχει στοιχεία φοιτητών/τριών και δηλώσεων μαθημάτων στο τρέχον εξάμηνο. Ένα δείγμα περιεχομένου του αρχείου είναι:*

```
2149\tΜΠΡΟΥΜΟΥΤΗ\tΑΛΕΞΙΑ
4
ΕΥ0160Ε
ΕΥ02400
ΕΥ03610
ΤΠ02030

2059\tΜΥΛΩΝΑΣ\tΣΤΑΥΡΟΣ
6
ΕΥ02210
ΕΥ0240Ε
ΕΥ0331Ε
ΤΠ01010
ΤΠ0305Ε
ΤΠ03140
```

*Εδώ έχουμε στοιχεία και δηλώσεις μαθημάτων για δύο φοιτητές. Στην πρώτη γραμμή έχουμε τον αριθμό μητρώου του/της φοιτητή/τριας (π.χ. "*2149*" ή "*2059*"), μετά το επώνυμό του/της ("*ΜΠΡΟΥΜΟΥΤΗ*" ή "*ΜΥΛΩΝΑΣ*") και τέλος το όνομα ("*ΑΛΕΞΙΑ*" ή "*ΣΤΑΥΡΟΣ*"). Στην επόμενη γραμμή υπάρχει φυσικός αριθμός που είναι το πλήθος των μαθημάτων που δήλωσε ο/η φοιτητής/τρια. Στη συνέχεια ακολουθούν οι κωδικοί των μαθημάτων που δηλώθηκαν.*

Τα στοιχεία δύο φοιτητών διαχωρίζονται με μια κενή γραμμή.

Το αρχείο μαθημάτων είναι ελεγμένο. Το αρχείο δηλώσεων μπορεί να έχει λάθη δύο ειδών:

- *Περισσότερες από μια δηλώσεις του ίδιου φοιτητή.*
- *Λάθος κωδικό μαθήματος.*

Θέλουμε ένα πρόγραμμα που θα διαβάσει τα αρχεία που δίνονται και θα μας δώσει τέσσερα νέα:

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* **Students.dta**, *που θα έχει αντικείμενα τύπου*

```
class Student
{
public:
   enum { sNameSz = 20 };
// . . .
private:
   unsigned int sIdNum;            // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
   unsigned int sWH;               // ώρες ανά εβδομάδα
   unsigned int sNoOfCourses;      // αριθμός μαθημάτων που δήλωσε
}; // Student
```

ένα για κάθε φοιτητή.

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* **enrllmnt.dta**, *που θα έχει αντικείμενα τύπου*

```
class StudentInCourse
{
public:
// . . .
private:
   enum { sicCCodeSz = 8 };
   unsigned int sicSIdNum;            // αριθμός μητρώου
   char         sicCCode[sicCCodeSz]; // κωδικός μαθήματος
   float        sicMark;              // βαθμός στο μάθημα
}; // StudentInCourse
```

ένα για κάθε δήλωση (χωρίς λάθη) μαθήματος από φοιτητή.

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* **courses.dta**, *που θα έχει αντικείμενα τύπου Course, ένα για κάθε μάθημα. Η κλάση Course είναι σαν τη SylCourse με ένα επί πλέον μέλος:*

```
unsigned int cNoOfStudents;    // αριθ. φοιτητών στο μάθημα
```

Φυσικά όλα τα αντικείμενα θα έχουν ενημερωμένο το νέο μέλος.

- *Ένα μορφοποιημένο αρχείο καταγραφής λαθών, με όνομα* **log.txt**. *Εκτός των λαθών που περιγράψαμε παραπάνω θα καταγράφονται και οι αριθμοί μητρώου των φοιτητών που δήλωσαν μαθήματα με άθροισμα ωρών μεγαλύτερο των 30 ανά εβδομάδα.*

Σημείωση: ▶

Το αρχείο **gCourses.dta** *έχει γραφεί με την παρακάτω μέθοδο:*

```
void SylCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw SylCourseXptn( "save", SylCourseXptn::fileNotOpen );
   bout.write( cCode, sizeof(cCode) );                    // κωδικός μαθήματος
   bout.write( cTitle, sizeof(cTitle) );                   // τίτλος μαθήματος
   bout.write( reinterpret_cast<const char*>(&cFSem), sizeof(cFSem) );
                                                          // τυπικό εξάμηνο
   bout.write( reinterpret_cast<const char*>(&cCompuls), sizeof(cCompuls) );
                                                  // υποχρεωτικό ή επιλογής
   bout.write( &cSector, sizeof(cSector) );                       // τομέας
   bout.write( cCateg, sizeof(cCateg) );                      // κατηγορία
   bout.write( reinterpret_cast<const char*>(&cWH), sizeof(cWH) );
                                                  // ώρες ανά εβδομάδα
   bout.write( reinterpret_cast<const char*>(&cUnits), sizeof(cUnits) );
                                                 // διδακτικές μονάδες
   bout.write( cPrereq, sizeof(cPrereq) );                // προαπαιτούμενο
   if ( bout.fail() )
      throw SylCourseXptn( "save", SylCourseXptn::cannotWrite );
} // SylCourse::save
```

◀

Prj03.2 Το (Πρώτο) Σχέδιο για το Πρόγραμμα

Ας ξεκινήσουμε με δύο

Παρατηρήσεις ▶

1. Τα *cNoOfStudents* της *Course* και *sNoOfCourses*, *sWH* της *Student* φαίνονται κάπως «ξεκάφωτα». Και είναι! Όλα τα άλλα χαρακτηριστικά της *Course* είναι «πάγια» ενώ το *cNoOfStudents* αναφέρεται στο τρέχον εξάμηνο. Τα ίδια ισχύουν και για τα δύο χαρακτηριστικά της *Student*. Η σχεδίαση των κλάσεων δεν είναι σωστή! Παρ' όλα αυτά θα επιμείνουμε στη συγκεκριμένη περιγραφή για εκπαιδευτικούς λόγους: *Εδώ ενδιαφερόμαστε μόνο για την προγραμματιστική διαχείριση των κλάσεων.* Η σχεδίαση είναι αντικείμενο άλλων τομέων της τεχνολογίας λογισμικού.
2. Ένα αντικείμενο της *StudentInCourse* θα πρέπει να έχει όλες τις πληροφορίες που σχετίζονται με την εγγραφή ενός φοιτητή σε ένα μάθημα, π.χ.: δύο βαθμούς (για τις δύο εξεταστικές περιόδους) και όχι έναν, ακαδημαϊκό εξάμηνο (αν θέλουμε να έχουμε πολλά εξάμηνα) κλπ. Εδώ βάλαμε μόνον έναν βαθμό, για να απλουστεύσουμε τα πράγματα.◀

Η διαφορά αυτού του παραδείγματος από το «παράδειγμα της μπαταρίας» είναι ότι εκεί είχαμε μια κλάση και η διατύπωση του προβλήματος καθόριζε τις μεθόδους που έπρεπε να αναπτυχθούν. Εδώ, που δεν έχουμε τέτοια βοήθεια, πώς θα βρούμε τις μεθόδους που χρειαζόμαστε; Κάνουμε ένα σχέδιο του προγράμματος και από εκεί εντοπίζουμε τις ανάγκες μας. Υπάρχουν βέβαια ορισμένα σχεδόν απαραίτητα συστατικά για οποιαδήποτε κλάση που σχετίζονται με το πρόγραμμα στο οποίο θα χρησιμοποιηθεί. Θα τα μάθουμε στα επόμενα μαθήματα.

Ας κάνουμε λοιπόν ένα πρώτο σχέδιο για να δούμε τι γίνεται:

```
Φόρτωσε τον πίνακα των μαθημάτων
Μηδένισε το cNoOfStudents του κάθε μαθήματος
Άνοιξε το αρχείο enrllmnt.txt
do {
   Διάβασε τα στοιχεία ενός φοιτητή
   if ( δεν τελείωσε το αρχείο )
   {
      if ( υπάρχει στον πίνακα φοιτητών )
      {
         Αγνόησε τη δήλωση // υπάρχει ήδη άλλη δήλωση
         Γράψε στο log
      }
      else
      {
         Βάλε τον φοιτητή στον πίνακα φοιτητών
         Μηδένισε τα sNoOfCourses και sWH του φοιτητή
         for ( όλα τα μαθήματα που δήλωσε ο φοιτητής )
         {
            αναζήτησε τον κωδικό του μαθήματος
                                              στον πίνακα μαθημάτων
            if ( δεν υπάρχει )
               Γράψε στο log
            else
            {
               Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος)
                                               στον πίνακα δηλώσεων
               Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή
               Ενημέρωσε το cNoOfStudents του κάθε μαθήματος
            }
         } // for
      } // if ( υπάρχει στον πίνακα φοιτητών )
   } // if (δεν τελείωσε το αρχείο )
} while ( δεν τελείωσε το αρχείο )
for ( όλους τους φοιτητές )
   if ( sWH > 30 ) Γράψε στο log
Φύλαξε τα στοιχεία του πίνακα φοιτητών στο Students.dta
Φύλαξε τα στοιχεία του πίνακα δηλώσεων στο enrllmnt.dta
Φύλαξε τα στοιχεία του πίνακα μαθημάτων στο gCourses.dta
```

Στη συνέχεια θα καταγράψουμε τις απαιτήσεις που προκύπτουν για κάθε κλάση.

Prj03.3 Η Κλάση *Course*

Τι πρέπει να κάνουμε με την *Course*;

- Να διαβάζουμε στοιχεία αντικειμένων της κλάσης από μη-μορφοποιημένο αρχείο («*Φόρτωσε τον πίνακα των μαθημάτων*»). Αν υλοποιήσουμε την κλάση *SylCourse* θα πρέπει να γράψουμε
  - μια μέθοδο, ας την πούμε *load*, που θα φορτώνει τα στοιχεία ενός μαθήματος από (μη-μορφοποιημένο) αρχείο και
  - μια μέθοδο (ή, καλύτερα, επιφόρτωση του τελεστή εκχώρησης) που θα αντιγράφει ένα αντικείμενο *SylCourse* σε ένα αντικείμενο *Course*.

  Κάτι τέτοιο όμως είναι έξω από τους στόχους μας.[1] Αντί για αυτό θα γράψουμε μια συνάρτηση –ας την πούμε *loadSylCourse*– που θα διαβάζει τις τιμές μελών του αντικειμένου *SylCourse* και θα τις περνάει μια προς μία στα αντίστοιχα μέλη ενός αντικειμένου *Course*. Αυτό το «πέρασμα τιμών» θα πρέπει να γίνει με μεθόδους που θα δούμε στη συνέχεια.

[1] Αργότερα θα υλοποιήσουμε την κλάση *SylCourse* (με όνομα *Course*).

- Να μπορούμε να μηδενίζουμε τον μετρητή («*Μηδένισε το cNoOfStudents του κάθε μαθήματος*»). Αυτό μπορεί να γίνει είτε με μια μέθοδο *clearStudents()* που κάνει μόνον αυτό είτε με μια *setNoOfStudents()* που είναι γενικότερη.
- Να βρίσκουμε ένα αντικείμενο με συγκεκριμένο κωδικό («*Αναζήτησε τον κωδικό του μαθήματος στον πίνακα μαθημάτων*»). Επειδή η αναζήτηση θα γίνεται με βάση τον κωδικό θα χρειαστούμε μια μέθοδο που να μας δίνει τον κωδικό (*getCode()*).
- Να παίρνουμε τις ώρες διδασκαλίας ανά εβδομάδα για να τις προσθέσουμε στις αντίστοιχες του φοιτητή («*Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή*»). Θα πρέπει να γράψουμε μια *getWH*.
- Να αυξάνουμε τον αριθμό των φοιτητών που γράφονται στο μάθημα κατά 1 («*Ενημέρωσε το cNoOfStudents του κάθε μαθήματος*»). Μπορούμε είτε να γράψουμε μια *add1Student()* που αυξάνει την τιμή του *cNoOfStudents* κατά 1 είτε να γράψουμε μια *getNoOfStudents()* και να τη χρησιμοποιήσουμε μαζί με τη *setNoOfStudents()*.
- Να γράφουμε στοιχεία αντικειμένων της κλάσης σε μη-μορφοποιημένο αρχείο («*Φύλαξε τα στοιχεία του πίνακα μαθημάτων στο* `courses.dta`»). Ας πούμε *save* τη σχετική μέθοδο.

Ξεκινούμε με τη *loadSylCourse()*. Όπως μπορούμε να δούμε στη δήλωση της *SylCourse* έχουμε:

- Ένα μέλος τύπου `bool` (*cCompuls*).
- Τρία μέλη τύπου `unsigned int` (*cFSem*, *cWH*, *cUnits*).
- Πέντε μέλη τύπου `char` (*cCode* και *cPrereq* με μήκος *cCodeSz* (8), *cTitle* με μήκος *cTitleSz* (80), *cCateg* με μήκος *cCategSz* (4) και *cSector* με μήκος 1).

Αν λοιπόν δηλώσουμε τρεις ενταμιευτές:[2]

```
   bool bBuf;
   unsigned int iBuf;
   char cBuf[Course::cTitleSz];
```

μπορούμε να διαβάσουμε αυτά που γράφηκαν με τη *SylCourse*`::`*save* με τις εξής εντολές:

```
   bin.read( cBuf, Course::cCodeSz );                    // κωδικός μαθήματος
   bin.read( cBuf, Course::cTitleSz );                    // τίτλος μαθήματος
   bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                          // τυπικό εξάμηνο
   bin.read( reinterpret_cast<char*>(&bBuf), sizeof(bool) );
                                                 // υποχρεωτικό ή επιλογής
   bin.read( cBuf, sizeof(char) );                          // τομέας
   bin.read( cBuf, Course::cCategSz );                   // κατηγορία
   bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                 // ώρες ανά εβδομάδα
   bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                 // διδακτικές μονάδες
   bin.read( cBuf, Course::cCodeSz );                    // προαπαιτούμενο
```

Η *loadSylCourse()* θα πρέπει να είναι:

```
void loadSylCourse( ifstream& bin, Course& oneCourse )
{
   bool bBuf;
   unsigned int iBuf;
   char cBuf[Course::cTitleSz];

   bin.read( cBuf, Course::cCodeSz );                    // κωδικός μαθήματος
   if ( !bin.eof() )
   {
                                                 oneCourse.setCode( cBuf );
```

[2] και με την προϋπόθεση ότι και στην *Course* θα δηλώσουμε:

```
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
```

```
      bin.read( cBuf, Course::cTitleSz );          oneCourse.setTitle( cBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                   oneCourse.setFSem( iBuf );
      bin.read( reinterpret_cast<char*>(&bBuf), sizeof(bool) );
                                                  oneCourse.setCompuls( bBuf );
      bin.read( cBuf, sizeof(char) );           oneCourse.setSector( cBuf[0] );
      bin.read( cBuf, Course::cCategSz );          oneCourse.setCateg( cBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                     oneCourse.setWH( iBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                   oneCourse.setUnits( iBuf );
      bin.read( cBuf, Course::cCodeSz );          oneCourse.setPrereq( iBuf );
      if ( bin.fail() )
         throw ProgXptn( "loadSylCourse", ProgXptn::cannotRead );
   }
} // loadSylCourse
```

Όπως βλέπεις, θα χρειαστούμε μεθόδους *setCode*() *setFSem(), setCompuls(), setSector(), setCateg(), setWH(), setUnits()* και *setPrereq()* για να αλλάζουμε (καθορίζουμε) τις τιμές των *cCode, cTitle, cFSem, cCompuls, cSector, cCateg, cWH, cUnits* και *cPrereq* αντιστοίχως.

Αυτές οι μέθοδοι θα πρέπει να γραφούν προσεκτικά διότι δεν θα πρέπει να αφήνουν «σκουπίδια» να περνούν μέσα στα αντικείμενα. Αργότερα θα δούμε πώς κάνουμε αυτήν τη δουλειά με πιο συστηματικό τρόπο. Εδώ, έχοντας τη σιγουριά ότι διαβάζουμε ένα αρχείο με σωστά στοιχεία, θα γράψουμε μεθόδους που θα περνούν τα στοιχεία χωρίς ελέγχους. Αλλά, για να δεις πώς περίπου γίνονται αυτοί οι έλεγχοι, θα γράψουμε

- Τη *setFSem()* έτσι ώστε να δέχεται ως τυπικό εξάμηνο διδασκαλίας του μαθήματος κάτι μεταξύ 1 και 8.

```
void Course::setFSem( int aFSem )
{
   if ( aFSem < 1 || 8 < aFSem)
      throw CourseXptn( "setFSem", CourseXptn::rangeError, aFSem );
   cFSem = aFSem;
} // Course::setFSem
```

- Τη *setPrereq()* έτσι ώστε να μην επιτρέπει ένα μάθημα να φέρεται ως προαπαιτούμενο του εαυτού του.

```
void Course::setPrereq( const string& prCode )
{
   if ( (prCode.length() > 0) && (prCode == cCode) )
      throw CourseXptn( "setPrereq", CourseXptn::autoRef, prCode.c_str() );
   strncpy( cPrereq, prCode.c_str(), cCodeSz-1 );  cPrereq[cCodeSz-1] = '\0';
} // Course::setPrereq
```

Φυσικά, για να διασφαλίσουμε ότι ένα μάθημα δεν φέρεται ως προαπαιτουμενο του εαυτού του θα πρέπει να προσέξουμε και τη

```
void Course::setCode( string aCode )
{
   if ( (aCode.length() > 0) && (aCode == cPrereq) )
      throw CourseXptn( "setCode", CourseXptn::autoRef, aCode.c_str() );
   strncpy( cCode, aCode.c_str(), cCodeSz-1 );        cCode[cCodeSz-1] = '\0';
} // Course::setCode
```

Οι υπόλοιπες, οι «απρόσεκτες» προς το παρόν, θα είναι:

```
void Course::setTitle( string aTitle )
{
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
} // Course::setTitle
void Course::setSector( char aSector )  {  cSector = aSector;  }
void Course::setCateg( string aCateg )
{
   strncpy( cCateg, aCateg.c_str(), cCategSz-1 );  cTitle[cCategSz-1] = '\0';
} // Course::setCateg
void Course::setWH( int aWH )  {  cWH = aWH;  }
```

```
void Course::setUnits( int aUnits )  {  cUnits = aUnits;  }
```

Ας δούμε τώρα τα υπόλοιπα· και αρχίζουμε από τα εύκολα:

```
   unsigned int getNoOfStudents() {  return cNoOfStudents;  }
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   unsigned int getWH() const {  return cWH;  }
```

Γιατί δεν γράψαμε μια *setNoOfStudents*; Διότι αυτές που γράψαμε είναι πιο λειτουργικές: Αρχικώς, για κάποιο μάθημα "`Course ac`", βάζουμε:

```
   ac.clearStudents();
```

και κάθε φορά που βρίσκουμε φοιτητή που το έχει δηλώσει δίνουμε:

```
   ac.add1Student();
```

Αν υλοποιούσαμε τη *setNoOfStudents* θα είχαμε αντιστοίχως:

```
   ac.setNoOfStudents( 0 );
```

και:

```
   ac.setNoOfStudents( ac.getNoOfStudents()+1 );
```

Αλλά, η *setNoOfStudents* χρειάζεται έλεγχο (για αρνητικές τιμές) ενώ και οι μη αρνητικές είναι «προβληματικές» (τι νόημα έχει `ac.setNoOfStudents(20);`) Η *getNoOfStudents()* χρειάζεται γενικότερα.

Εύκολη είναι και η

```
   const char* getCode() const {  return cCode;  }
```

αλλά εδώ χρειάζονται εξηγήσεις για το πρώτο "`const`". Γιατί χρειάζεται; Βγάλε τα δύο "`const`" και δοκίμασε τα παρακάτω:

```
   cout << ac.getCode() << endl;
   char* p( ac.getCode() );
   p[3] = 'X';  p[4] = 'Z';
   cout << ac.getCode() << endl;
```

Αποτέλεσμα:

```
EY0140E
EY0XZ0E
```

Δηλαδή, ανοίξαμε μια «κερκόπορτα» από όπου τροποποιείται αυθαιρέτως το αντικείμενό μας.

Κρατώντας τα δύο "`const`" (που πάνε «πακέτο») το μόνο που μπορείς να κάνεις με το *p* είναι:

```
   char* p( new char[strlen(ac.getCode())+1] );
   strcpy( p, ac.getCode() );
```

Τώρα ο *p* δείχνει *αντίγραφο* του *ac.cCode* και δεν έχει σχέση με το αντικείμενό μας.

Ας έλθουμε τώρα στις *load()* και *save()*. Η *save()* θα είναι σαν τη *SylCourse::save()*, που μας δόθηκε, αλλά θα φυλάγει και την τιμή του *cNoOfStudents*:

```
void Course::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw CourseXptn( "save", CourseXptn::fileNotOpen );
   bout.write( cCode, sizeof(cCode) );                    // κωδικός μαθήματος
   bout.write( cTitle, sizeof(cTitle) );                   // τίτλος μαθήματος
   bout.write( reinterpret_cast<const char*>(&cFSem), sizeof(cFSem) );
                                                          // τυπικό εξάμηνο
   bout.write( reinterpret_cast<const char*>(&cCompuls), sizeof(cCompuls) );
                                                 // υποχρεωτικό ή επιλογής
   bout.write( &cSector, sizeof(cSector) );                        // τομέας
   bout.write( cCateg, sizeof(cCateg) );                        // κατηγορία
   bout.write( reinterpret_cast<const char*>(&cWH), sizeof(cWH) );
                                                       // ώρες ανά εβδομάδα
   bout.write( reinterpret_cast<const char*>(&cUnits), sizeof(cUnits) );
                                                    // διδακτικές μονάδες
   bout.write( cPrereq, sizeof(cPrereq) );                 // προαπαιτούμενο
```

```
   bout.write( reinterpret_cast<const char*>(&cNoOfStudents),
               sizeof(cNoOfStudents) );              // αριθ. φοιτ. στο μάθημα
   if ( bout.fail() )
      throw CourseXptn( "save", CourseXptn::cannotWrite );
} // Course::save
```

Η *load()* θα γραφεί με οδηγό τη *save*, αφού θα διαβάσουμε όπως γράψαμε:

```
void Course::load( istream& bin )
{
   bin.read( cCode, cCodeSz );                              // κωδικός μαθήματος
   if ( !bin.eof() )
   {
      bin.read( cTitle, cTitleSz );                          // τίτλος μαθήματος
      bin.read( reinterpret_cast<char*>(&cFSem), sizeof(cFSem) );
                                                            // τυπικό εξάμηνο
      bin.read( reinterpret_cast<char*>(&cCompuls), sizeof(cCompuls) );
                                                          // υποχρεωτικό ή επιλογής
      bin.read( &cSector, sizeof(cSector) );                          // τομέας
      bin.read( cCateg, cCategSz );                                // κατηγορία
      bin.read( reinterpret_cast<char*>(&cWH), sizeof(cWH) );
                                                          // ώρες ανά εβδομάδα
      bin.read( reinterpret_cast<char*>(&cUnits), sizeof(cUnits) );
                                                        // διδακτικές μονάδες
      bin.read( cPrereq, cCodeSz );                          // προαπαιτούμενο
      bin.read( reinterpret_cast<char*>(&cNoOfStudents),
                sizeof(cNoOfStudents) );              // αριθ. φοιτ. στο μάθημα
      if ( bin.fail() )
         throw CourseXptn( "load", CourseXptn::cannotRead );
   } // if ( !bin.eof() ). . .
} // Course::load
```

Αυτή είναι η *load()* αλλά δεν είναι καλή. Γιατί; Όταν δίνεις την εντολή:

```
   ac.load( bin );
```

το αντικείμενο *ac* έχει κάποια τιμή. Αν κάτι «πάει στραβά» κατά την ανάγνωση αυτή η τιμή καταστρέφεται. Θα πρέπει να διαβάζουμε πιο προσεκτικά, όπως θα μάθουμε αργότερα. Προς το παρόν δεν θα έχουμε πρόβλημα, όπως θα δεις στη συνέχεια.

Για να μπορέσεις να κάνεις δοκιμές με την κλάση την εξοπλίζουμε και με μια:

```
void Course::display( ostream& tout ) const
{
   if ( tout.fail() )
      throw CourseXptn( "display", CourseXptn::fileNotOpen );
   tout << cCode << '\t' << cTitle << '\t';
   if ( cCompuls )  tout << "Y\t";
              else  tout << "YE\t";
   tout << cSector << '\t' << cCateg << '\t' << cWH << '\t'
        << cUnits << '\t' << cPrereq << '\t' << cFSem << '\t'
        << cNoOfStudents << endl;
   if ( tout.fail() )
      throw CourseXptn( "display", CourseXptn::cannotWrite );
} // Course::display
```

Να λοιπόν η «προχειρογραμμένη»

```
class Course
{
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
   Course( string aCode="" );
   const char* getCode() const {  return cCode;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) { cCompuls = aCompuls; };
   void setSector( char aSector )  {  cSector = aSector;  };
   void setCateg( string aCateg );
```

```
   void setWH( int aWH )  {  cWH = aWH;  };
   void setUnits( int aUnits )  {  cUnits = aUnits;  };
   void setPrereq( const string& prCode );
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   void save( ostream& bout ) const;
   void load( istream& bin );
   void display( ostream& tout ) const;
private:
   char         cCode[cCodeSz];   // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   char         cPrereq[cCodeSz]; // προαπαιτούμενο
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
}; // Course
```

και η αντίστοιχη κλάση εξαιρέσεων:

```
struct CourseXptn
{
   enum {  rangeError, autoRef,
           fileNotOpen, cannotWrite, cannotRead  };
   char funcName[100];
   int  errorCode;
   int  errIntVal;
   char errStrVal[100];
   CourseXptn( const char* mn, int ec, const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const char* mn, int ec, int iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errIntVal = iv;  }
}; // CourseXptn
```

Χρήσιμο είναι να ορίσουμε:

```
typedef Course* PCourse;
```

Prj03.4 Η Κλάση *Student*

Να δούμε τώρα τι απαιτήσεις έχουμε για τη *Student*.

- Η «*Διάβασε τα στοιχεία ενός φοιτητή*» παραπέμπει σε διάφορες εναλλακτικές λύσεις:
  - Να γράψουμε μια *readFromText* που διαβάζει τα στοιχεία από αρχείο –σαν τη *load*– αλλά, αυτήν τη φορά, μορφοποιημένο (text) ή
  - να γράψουμε τρεις "*set*": *setIdNum*, *setSurname*, *setFirstname* ή
  - να γράψουμε έναν δημιουργό

```
Student( int aIdNum=0, string aSurn="", string aFirstn="" )
```

- Για να κάνουμε τον έλεγχο «`if (`*υπάρχει στον πίνακα φοιτητών*`)`» πρέπει να αναζητούμε ένα αντικείμενο κλάσης *Student*. Επειδή η αναζήτηση θα γίνεται με βάση τον αριθμό μητρώου, θα χρειαστούμε μια μέθοδο που θα μας τον δίνει (*getIdNum*). Ανάγκη για την ίδια μέθοδο προκύπτει και από την «*Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος) στον πίνακα δηλώσεων*». Θα χρειαστούμε και εδώ έναν δημιουργό για αναζήτηση με τη *linSearch()*; Όχι, διότι στην περίπτωση αυτήν αναζητούμε ένα αντικείμενο που έχουμε διαβάσει από το αρχείο. Πάντως η επιφόρτωση του "`!=`" είναι αναγκαία.

- Ακολουθώντας το σκεπτικό που αναπτύξαμε για τη διαχείριση του μέλους *cNoOfStudents* της *Course*, από την «*Μηδένισε τα sNoOfCourses και sWH του φοιτητή*» συνάγουμε ανάγκη για μέθοδο *clearCourses()*.
- Για να ενημερώσουμε τους δύο μετρητές («*Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή*») θα χρειαστούμε μέθοδο *add1Course()*.
- Για να ελέγξουμε τις εβδομαδιαίες ώρες του φοιτητή («`if (sWH > 30)`») θα χρειαστούμε μια *getWH()*.
- Τέλος, θα χρειαστούμε μια μέθοδο, ας την πούμε *save()*, για να γράφουμε στοιχεία αντικειμένων της κλάσης σε μη-μορφοποιημένο αρχείο («*Φύλαξε τα στοιχεία του πίνακα φοιτητών στο* `Students.dta`»).

  Ξεκινώντας και εδώ από τα εύκολα, έχουμε:

```
   unsigned int getIdNum() const {  return sIdNum;  }
   unsigned int getWH() const {  return sWH;  }
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
   void add1Course( int aIncrWH );
```

όπου:

```
void Student::add1Course( int aIncrWH )
{
   if ( aIncrWH <= 0 )
      throw StudentXptn( "add1Course", StudentXptn::nonPosIncr,
                         aIncrWH );
   ++sNoOfCourses;
   sWH += aIncrWH;
} // Student::add1Course
```

Ο αριθμός μητρώου είναι ένα άλλο παράδειγμα *υποκατάστατου κλειδιού*. Μπορούμε λοιπόν να επιφορτώσουμε τον "`!=`" με σύγκριση αριθμών μητρώου μόνο:

```
bool operator!=( const Student& a, const Student& b )
{  return ( a.getIdNum() != b.getIdNum() );  }
```

Για το διάβασμα θα επιλέξουμε –για εκπαιδευτικούς λόγους, για να δεις τη διαφορά από τη *load()*– να γράψουμε μια:

```
void Student::readFromText( istream& tin )
{
   string line;
   getline( tin, line, '\n' );
   if ( !tin.eof() )
   {
      size_t t1Pos( line.find("\t") );
      if ( t1Pos >= line.length() )
         throw StudentXptn( "readFromText",
                            StudentXptn::incomplete );
      size_t t2Pos( line.find("\t", t1Pos+1) );
      if ( t2Pos >= line.length() )
         throw StudentXptn( "readFromText",
                            StudentXptn::incomplete );
      string str1( line.substr(0, t1Pos) );
      int    iStr1;
      iStr1 = atoi( str1.c_str() );
      if ( iStr1 <= 0 )
         throw StudentXptn( "readFromText",
                            StudentXptn::negIdNum, iStr1 );
      sIdNum = iStr1;
      str1 = line.substr( t1Pos+1, t2Pos-t1Pos-1 );
      strncpy( sSurname, str1.c_str(), sNameSz-1 );
                                       sSurname[sNameSz-1] = '\0';
      str1 = line.substr( t2Pos+1 );
      strncpy( sFirstname, str1.c_str(), sNameSz-1 );
                                    sFirstname[sNameSz-1] = '\0';
      sNoOfCourses = 0;
      sWH = 0;
```

```
   } // if ( !tin.eof . . .
} // Student::readFromText
```

Εδώ πρόσεξε τα εξής:

- Αν διαβάσουμε γραμμή που δεν έχει δύο '\t' ρίχνουμε εξαίρεση με κωδικό σφάλματος "*incomplete*". Γιατί το βάλαμε αυτό; Αφού έχουμε τη διαβεβαίωση ότι δεν υπάρχουν λάθη τέτοιου είδους. Η εμβέλεια της μεθόδου είναι πέρα από την τρέχουσα εφαρμογή.
- Ένα πρόβλημα μπορεί να παρουσιαστεί είναι το εξής: να πάρουμε όνομα ή επώνυμο που δεν χωράει στον πίνακα που έχουμε προβλέψει. Αλλά, σε τέτοια περίπτωση, δεν υπάρχει λόγος να ρίξουμε εξαίρεση. Απλώς αποθηκεύουμε όσο χωράει.

Η *save()* γράφεται όπως αυτή της *Course*:

```
void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum),
               sizeof(sIdNum) );                          // αριθμός μητρώου
   bout.write( sSurname, sNameSz );
   bout.write( sFirstname, sNameSz );
   bout.write( reinterpret_cast<const char*>(&sNoOfCourses),
               sizeof(sNoOfCourses) );              // αριθ. μαθημ. που δήλωσε
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
                                                        // ώρες ανά εβδομάδα
   if ( bout.fail() )
      throw StudentXptn( "save", StudentXptn::cannotWrite );
} // Student::save
```

Η *display()* μπορεί να είναι χρήσιμη και εδώ:

```
void Student::display( ostream& tout )
{
   tout << sIdNum << '\t' << sSurname << '\t' << sFirstname
        << endl << sNoOfCourses << '\t' << sWH << endl;
} // Student::display
```

Καταλήγουμε λοιπόν στην:

```
class Student
{
public:
   unsigned int getIdNum() const {  return sIdNum;  }
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
   unsigned int getWH() const {  return sWH;  }
   void add1Course( int aIncrWH );
   void readFromText( istream& tin );
   void save( ostream& bout ) const;
   void display( ostream& tout );
private:
   enum { sNameSz = 20 };
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
   unsigned int sWH;           // ώρες ανά εβδομάδα
   unsigned int sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
}; // Student
```

και στην αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentXptn
{
   enum { incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[100];
   int  errorCode;
   int  errValue;
   StudentXptn( char* mn, int ec, int ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errValue = ev;  }
```

```
}; // StudentXptn
```

Και εδώ θα ορίσουμε:

```
typedef Student* PStudent;
```

Prj03.5 Η Κλάση *StudentInCourse*

Για την κλάση αυτήν έχουμε:

- «*Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος) στον πίνακα δηλώσεων.*» Αυτή η απαίτηση μπορεί να αντιμετωπισθεί με έναν δημιουργό, με δύο "*set*" ή με μια "*set*".
- «*Φύλαξε τα στοιχεία του πίνακα δηλώσεων στο* `enrllmnt.dta`.» Από αυτό προκύπτει ανάγκη για μια *save*.

Εδώ, η *save()* είναι εύκολη:

```
void StudentInCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentInCourseXptn( "save", StudentInCourseXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sicSIdNum), sizeof(sicSIdNum) );
                                                          // αριθμός μητρώου
   bout.write( sicCCode, sicCCodeSz );                   // κωδικός μαθήματος
   bout.write( reinterpret_cast<const char*>(&sicMark), sizeof(sicMark) );
                                                          // βαθμός στο μάθημα
   if ( bout.fail() )
      throw StudentInCourseXptn( "save",
                                 StudentInCourseXptn::cannotWrite );
} // StudentInCourse::save
```

Ας δούμε τώρα την πρώτη απαίτηση. Για να κάνουμε εισαγωγή ενός ζεύγους στον πίνακα θα πρέπει να έχουμε τη σιγουριά ότι *υπάρχουν και ο φοιτητής και το μάθημα*. Αυτό

- εξασφαλίζεται με αναζήτηση στους αντίστοιχους πίνακες και
- γίνεται με *μια* μέθοδο.

Επιπλέον, αυτή η μέθοδος θα πρέπει να ενημερώνει και τα αντίστοιχα αντικείμενα φοιτητή και μαθήματος.

Άρα οι επιλογές μας περιορίζονται σε δύο: ένας δημιουργός ή μια "*set*". Και οι δύο είναι σωστές, αλλά δεν είναι και τόσο «κομψό» σε δημιουργό μιας κλάσης να περνάς πίνακες με στοιχεία δύο άλλων κλάσεων και να ενημερώνει αντικείμενα άλλων κλάσεων. Για τον λόγο αυτόν επιλέγουμε τη «διπλή» "*set*":

```
void StudentInCourse::setIdNumCCode( unsigned int aIdNum,
                                     string aCCode,
                                     Student stArr[], int nOfStudents,
                                     Course crsArr[], int nOfCourses )
```

Τώρα βέβαια, παρατηρώντας το σχέδιο που κάναμε βλέπουμε ότι όταν έρχεται η ώρα να κάνουμε κάτι τέτοια ξέρουμε ότι ο φοιτητής υπάρχει σίγουρα. Επομένως μπορούμε να περάσουμε στη μέθοδο το αντικείμενο με τα στοιχεία του φοιτητή. Να λοιπόν μια απλοποιημένη μορφή της μεθόδου:

```
void StudentInCourse::setIdNumCCode( Student& aStudent,
                                     string aCCode,
                                     Course crsArr[], int nOfCourses )
{
   crsArr[nOfCourses].setCode( aCCode.c_str() );
   int cPos( 0 );
   while ( strcmp(crsArr[nOfCourses].getCode(),
                  crsArr[cPos].getCode())!=0 )
      ++cPos;
   if ( cPos == nOfCourses )
     throw StudentInCourseXptn( "StudentInCourse",
                                StudentInCourseXptn::unknownCCode,
                                aCCode.c_str() );
```

```
   sicSIdNum = aStudent.getIdNum();
   aStudent.add1Course( crsArr[cPos].getWH() );
   strcpy( sicCCode, aCCode.c_str() );
   crsArr[cPos].add1Student();
} // StudentInCourse::setIdNumCCode
```

Η αναζήτηση μαθήματος με βάση τον κωδικό γίνεται με γραμμική αναζήτηση με φρουρό, αλλά η τοποθέτηση του φρουρού δημιουργεί μια ανάγκη για την κλάση *Course*: χρειαζόμαστε μέθοδο *Course*::*setCode*.

Έτσι, η κλάση γίνεται:

```
class StudentInCourse
{
public:
   void setIdNumCCode( Student& aStudent, string aCCode,
                       Course crsArr[], int nOfCourses );
   void save( ostream& bout ) const;
   void display( ostream& tout ) const;
private:
   unsigned int sicSIdNum;      // αριθμός μητρώου
   char         sicCCode[8];    // κωδικός μαθήματος
}; // StudentInCourse
```

όπου:

```
void StudentInCourse::display( ostream& tout ) const
{
   tout << sicSIdNum << ' ' << sicCCode << endl;
} // StudentInCourse::display
```

Η κλάση εξαιρέσεων είναι:

```
struct StudentInCourseXptn
{
   enum { unknownCCode };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   StudentInCourseXptn( const char* mn, int ec,
                        const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // StudentInCourseXptn
```

Και εδώ θα ορίσουμε:

```
typedef StudentInCourse* PStudentInCourse;
```

Prj03.6 Το Πρόγραμμα

Ας δούμε τώρα πώς θα υλοποιήσουμε το σχέδιο προγράμματος που κάναμε. Και πρώτα η «*Φόρτωσε τον πίνακα των μαθημάτων.*» Αλλά, πώς θα γίνει η «φόρτωση»; Η φύλαξη έγινε με τη *save* που είδαμε παραπάνω. Αυτό σημαίνει ότι και η φόρτωση θα γίνει με τη *load* (αργότερα θα εξετάσουμε και άλλες δυνατότητες.)

Δηλώνουμε λοιπόν στο πρόγραμμά μας (στη **main**):

```
   Course*      crsArr;
   unsigned int nOfCourses;
```

και στη συνέχεια καλούμε:

```
   loadCourses( "gCourses.dta", crsArr, nOfCourses );
```

Prj03.6.1 Η *loadCourses()*

Τι θα κάνει η *loadCourses()*; Θα πάρει την απαραίτητη μνήμη, θα διαβάσει το αρχείο και θα αποθηκεύσει αυτά που διάβασε στον πίνακα crsArr. Αν κάτι δεν πάει καλά (αν δεν ανοίγει το αρχείο, αν δεν πάρουμε δυναμική μνήμη), θα έχουμε crsArr == 0 και *nOfCourses* == 0 και θα ρίξει εξαίρεση. Δηλαδή:

```
crsArr = 0;   nOfCourses = 0;
Πάρε μνήμη
if ( αποτυχία )  throw ...
Άνοιξε το αρχείο
if ( αποτυχία )
{  επίστρεψε τη μνήμη;  throw ...  }
while ( δεν τελείωσε το αρχείο )
{
   if ( δεν έχεις μνήμη )
      Πάρε και άλλη μνήμη (renew)
   Διάβασε ένα μάθημα
   Μηδένισε το cNoOfStudents του κάθε μαθήματος
   ++nOfCourses;
} // while
```

Στο αρχείο `MyTmpltLib.h` υπάρχει το (γνωστό μας) περίγραμμα

```
// renew - παίρνει μνήμη για πίνακα με nf στοιχεία τύπου T και
//         στα πρώτα ni (<= nf) στοιχεία του αντιγράφουμε αρχικώς
//         αυτά των ni θέσεων του αρχικού p.
template< typename T >
void renew( T*& p, int ni, int nf )
```

και θα το χρησιμοποιήσουμε. Η *renew()* όμως κάνει αντιγραφή του πίνακα κάθε φορά που τον «μεγαλώνει». Για να αποφύγουμε τις πολλές αντιγραφές δεν θα παίρνουμε ένα επιπλέον στοιχείο κάθε φορά αλλά *cIncr* (=10) στοιχεία.[3] Σε μια μεταβλητή, *resCourses*, κρατούμε τον αριθμό στοιχείων που έχουμε δεσμεύσει. Έτσι, η «*Πάρε μνήμη*» γίνεται:

```
   const int cIncr( 10 );
   unsigned int resCourses;    // θέσεις που έχουμε δεσμεύσει
   crsArr = 0;   nOfCourses = 0;
   try {  crsArr = new Course[cIncr];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadCourses", ProgXptn::allocFailed );  }
   resCourses = cIncr;
```

και η «*if (δεν έχεις μνήμη) πάρε και άλλη μνήμη*»:

```
      if ( nOfCourses+1 == resCourses )
      {
         renew( crsArr, nOfCourses, resCourses+cIncr );
         resCourses += cIncr;
      }
```

Εκείνο το "+1", στην *nOfCourses*+1 == *resCourses*, το χρειαζόμαστε για να μπορούμε να κάνουμε γραμμική αναζήτηση με φρουρό.

Να ολόκληρη η *loadCourses()*:

```
void loadCourses( string flNm,
                  Course*& crsArr, unsigned int& nOfCourses )
{
// Πάρε (λίγη) μνήμη
   const int cIncr( 10 );
   unsigned int resCourses;
   crsArr = 0;   nOfCourses = 0;
   try {  crsArr = new Course[cIncr];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadCourses", ProgXptn::allocFailed );  }
   resCourses = cIncr;
// Άνοιξε το αρχείο
```

[3] Αν θέλεις να κάνεις κάτι καλύτερο ξαναδιάβασε αυτά που είπαμε στην §16.13.3.

```
   istream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
   {  delete[] crsArr;   // επίστρεψε τη μνήμη
      throw ProgXptn( "loadCourses", ProgXptn::cannotOpen, flNm.c_str() );
// Τώρα διάβαζε
   loadSylCourse( bin, crsArr[nOfCourses] );
   while ( !bin.eof() )
   {
      ++nOfCourses;
      crsArr[nOfCourses-1].clearStudents();
      if ( nOfCourses+1 == resCourses )
      {
         renew( crsArr, nOfCourses, resCourses+cIncr );
         resCourses += cIncr;
      } // if
      loadSylCourse( bin, crsArr[nOfCourses] );
   } // while
   bin.close();
} // loadCourses
```

Παρατηρήσεις ▶

- Πώς γίνεται το διάβασμα: διαβάζουμε αποθηκεύοντας στο πρώτο στοιχείο μετά το τελευταίο του πίνακα: στο `crsArr[nOfCourses]` (το οποίο υπάρχει πάντοτε). Αν όλα πάνε καλά (αν δεν ρίξει εξαίρεση η *load()*) αυξάνουμε τον *nOfCourses* (`++nOfCourses`) και το νέο στοιχείο περιλαμβάνεται στον πίνακα.
- Αν η *load()* δεν ρίξει εξαίρεση, για κάθε μάθημα που διαβάζουμε, μηδενίζουμε και τον *cNoOfStudents*.
- Η `catch` θα πιάσει οποιαδήποτε *CourseXptn*. Αν ο τύπος του λάθους δεν είναι αυτός που μας ενδιαφέρει (*cannotRead* ή *fileNotOpen*) την ξαναρίχνουμε (`throw`).◀

Prj03.6.2 Η Ανάγνωση του Αρχείου των Δηλώσεων

Η ανάγνωση του αρχείου `enrllmnt.txt` θα «γεμίσει» τον πίνακα φοιτητών και τον πίνακα δηλώσεων φοιτητών σε μαθήματα. Σε μεγάλο ποσοστό θα δουλέψουμε όπως δουλέψαμε παραπάνω. Κατ' αρχήν οι δηλώσεις:

```
   Student*    stArr;   unsigned int nOfStudents( 0 );
   const int sIncr( 10 );
   unsigned int resStudents;

   StudentInCourse* sic; unsigned int nOfStdInCrs( 0 );
   const int sicIncr( 30 );
   unsigned int resStdInCrs;
```

και οι πρώτες «φέτες» δυναμικής μνήμης:

```
      try { stArr = new Student[sIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStudents = sIncr;
      try { sic = new StudentInCourse[scIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStdInCrs = sicIncr;
```

Η «*Άνοιξε το αρχείο enrllmnt.txt*» γίνεται:

```
      ifstream tin( "enrllmnt.txt" );
      if ( tin.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "enrllmnt.txt" );
```

Πριν από αυτό όμως θα πρέπει να ανοίξουμε το αρχείο καταγραφής προβλημάτων (*log*):

```
      log.open( "log.txt" );
      if ( log.fail() )
```

```
         throw ProgXptn( "main",
                         ProgXptn::cannotOpen, "log.txt" );
```

Prj03.6.3 Τα Στοιχεία Ενός Φοιτητή . . .

Η «*Διάβασε τα στοιχεία ενός φοιτητή*» είναι η *readFromText()*; Όχι! Πέρα από τη *readFromText()* θα πρέπει το αντικείμενο που θα διαβαστεί να εισαχθεί στον stArr με ό,τι συνεπάγεται (δυναμική μνήμη.)

```
void readAStudent( istream& tin,
                   PStudent& stArr, unsigned int& nOfStudents,
                   unsigned int& resStudents, int sIncr )
{
   if ( nOfStudents+1 == resStudents )
   {
      renew( stArr, nOfStudents, resStudents+sIncr );
      resStudents += sIncr;
   }
   stArr[nOfStudents].readFromText( tin );
   ++nOfStudents;
   stArr[nOfStudents-1].clearNoOfCourses();
   stArr[nOfStudents-1].clearWH();
} // readAStudent
```

Όπως βλέπεις διαβάζοντας έναν φοιτητή κάνουμε περίπου τα ίδια με αυτά που κάνουμε διαβάζοντας ένα μάθημα. Εδώ όμως υπάρχει μια διαφορά: Υπάρχει πιθανότητα να έχουμε διαβάσει ήδη δήλωση μαθημάτων του φοιτητή. Αυτό

- Γίνεται αντιληπτό από την ύπαρξη αντικειμένου-φοιτητή πιο πριν στον πίνακα μετά από αναζήτηση με τον αριθμό μητρώου.
- Αντιμετωπίζεται με το να αγνοήσουμε την τελευταία δήλωση.

```
         readAStudent( tin,
                       stArr, nOfStudents, resStudents, sIncr );
         if ( !tin.eof() )
         {
            int pos( 0 );
            while ( stArr[pos].getIdNum() !=
                    stArr[nOfStudents-1].getIdNum() ) ++pos;
            if ( pos < nOfStudents-1 )  // υπάρχει ήδη
            {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
```

Τι ακολουθεί για τον φοιτητή και πώς το αγνοούμε;

- Ακολουθεί μια γραμμή με έναν αριθμό, ας τον πούμε *noc*, και
- *noc* γραμμές με έναν κωδικό μαθήματος στην κάθε μια.

Η παρακάτω συνάρτηση τα αγνοεί (τα διαβάζει και τα «ξεχνάει»):

```
void ignoreStudentData( istream& tin )
{
   string str1;
   getline( tin, str1, '\n' );
   int noc( atoi(str1.c_str()) );
   for ( int k(0); k < noc; ++k )  getline( tin, str1, '\n' );
} // ignoreStudentData
```

Συμπληρώνουμε λοιπόν:

```
            if ( pos < nOfStudents-1 )  // υπάρχει ήδη
            {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
               --nOfStudents;
               ignoreStudentData( tin );
               log << " multiple entry for student with id num "
                   << stArr[pos].getIdNum() << endl;
            }
            else  // δεν υπάρχει
            { . . .
```

Prj03.6.4 . . . Και οι Δηλώσεις Μαθημάτων

Και αν δεν υπάρχει; Ε, τότε ακολουθούμε το σχέδιο:

```
for ( όλα τα μαθήματα που δήλωσε ο φοιτητής )
{
   αναζήτησε τον κωδικό του μαθήματος στον πίνακα μαθημάτων
   if ( δεν υπάρχει )
      Γράψε στο log
   else
   {
      Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος)
                                        στον πίνακα δηλώσεων
      Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή
      Ενημέρωσε το cNoOfStudents του κάθε μαθήματος
   }
} // for
```

Αυτά τα κάνει η:

```
void readCourseCodes( istream& tin, Student& aStudent,
                      PCourse crsArr, int nOfCourses,
                      PStudentInCourse& sic, unsigned int& nOfStdInCrs,
                      unsigned int& resStdInCrs, int sicIncr,
                      ostream& log )
{
   if ( !tin.eof() )
   {
      string str1;
      getline( tin, str1, '\n' );
      if ( !tin.eof() )
      {
         int noc( atoi(str1.c_str()) );
         for ( int k(0); k < noc; ++k )
         {
            if ( nOfStdInCrs+1 == resStdInCrs )
            {
               renew( sic, nOfStdInCrs, resStdInCrs+sicIncr );
               resStdInCrs += sicIncr;
            }
            getline( tin, str1, '\n' );
            if ( !tin.eof() )
            {
               try
               {
                  sic[nOfStdInCrs].setIdNumCCode( aStudent,
                                                  str1.c_str(),
                                                  crsArr,
                                                  nOfCourses );
                  ++nOfStdInCrs;
               }
               catch( StudentInCourseXptn& x )
               {
                  log << " student with id num "
                      << aStudent.getIdNum()
                      << " asking course " << str1 << endl;
               }
            } // if ( !tin.eof()...
         } // for
      }
   }
} // readCourseCodes
```

Prj03.6.5 Η main

Δες πώς (περίπου) θα είναι η **main**:

```
int main()
```

```
{
   PCourse   crsArr;  unsigned int nOfCourses;

   PStudent  stArr;   unsigned int nOfStudents( 0 );
   const int sIncr( 10 );
   unsigned int resStudents;

   PStudentInCourse sic; unsigned int nOfStdInCrs( 0 );
   const int sicIncr( 30 );
   unsigned int resStdInCrs;

   ofstream log;
   try
   {
      loadCourses( "gCourses.dta", crsArr, nOfCourses );

      try { stArr = new Student[sIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStudents = sIncr;
      try { sic = new StudentInCourse[sicIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStdInCrs = sicIncr;

      log.open( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt" );
      ifstream tin( "enrllmnt.txt" );
      if ( tin.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "enrllmnt.txt" );
      while ( !tin.eof() )
      {
         readAStudent( tin, stArr, nOfStudents, resStudents, sIncr );
         if ( !tin.eof() )
         {
            int pos( 0 );
            while ( stArr[pos].getIdNum() !=
                    stArr[nOfStudents-1].getIdNum() ) ++pos;
            if ( pos < nOfStudents-1 )  // υπάρχει ήδη
            {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
               --nOfStudents;
               ignoreStudentData( tin );
               log << " multiple entry for student with id num "
                   << stArr[pos].getIdNum() << endl;
            }
            else  // δεν υπάρχει
            {
               readCourseCodes( tin, stArr[nOfStudents-1],
                                crsArr, nOfCourses,
                                sic, nOfStdInCrs, resStdInCrs,
                                sicIncr, log );
            }
         }
         if ( !tin.eof() )
         {
            string str1;
            getline( tin, str1, '\n' ); // blank line
         }
      } // while
      tin.close();
      for ( int k(0); k < nOfStudents; ++k )
      {
         if ( stArr[k].getWH() > 30 )
            log << "student with id num " << stArr[k].getIdNum()
                << ": " << stArr[k].getWH() << " hours/week"
                << endl;
```

```
      } // for
      save1DTable( "Students.dta", stArr, nOfStudents );
      save1DTable( "enrllmnt.dta", sic, nOfStdInCrs );
      save1DTable( "gCourses.dta", crsArr, nOfCourses );
      delete[] sic;
      delete[] stArr;
      delete[] crsArr;
   } // try
   catch( ProgXptn& x )  { . . . } // catch( ProgXptn
   catch( MyLibXptn& x ) { . . . } // catch( MyLibXptn
   catch( CourseXptn& x ) { . . . } // catch( CourseXptn
   catch( StudentXptn& x ) { . . . } // catch( StudentXptn
   catch( StudentInCourseXptn& x )
   { . . . } // catch( StudentInCourseXptn
   catch( ... ) { . . . }
   log.close();
} // main
```

Παρατηρήσεις ▶

- Ο έλεγχος για την υπέρβαση των 30 ωρών ανά εβδομάδα γίνεται (φυσικά) στο τέλος.
- Η *save1DTable()* υπάρχει στο `MyTmpltLib.h`:

```
// save1DTable -- Φυλάγει σε αρχείο binary με όνομα flNm τα n
//                στοιχεία του πίνακα tbl. Πρέπει να έχει ορισθεί
//                η T::save( ostream& )
template < class T >
void save1DTable( string flNm, T tbl[], int n )
{
   if ( tbl == 0 && n > 0 )
      throw MyTmpltLibXptn( "save1DTable",
                            MyTmpltLibXptn::noArray );
   ofstream bout( flNm.c_str(),
                  ios_base::binary|ios_base::trunc );
   if ( bout.fail() )
      throw MyTmpltLibXptn( "save1DTable",
                            MyTmpltLibXptn::cannotCreate,
                            flNm.c_str() );
   for ( int k(0); k < n; ++k )
   {
      tbl[k].save( bout );
      if ( bout.fail() )
         throw MyTmpltLibXptn( "save1DTable",
                               MyTmpltLibXptn::cannotWrite,
                               flNm.c_str() );
   }
   bout.close();
} // save1DTable
```

- Δηλώσαμε το *log* πριν από την `try`, το ανοίξαμε μέσα στη σύνθετη εντολή της `try` και τον κλείνουμε μετά την τελευταία `catch`. Γιατί; Για να έχουμε τη δυνατότητα να γράφουμε στο *log* σε όποιαν `catch` χρειαστεί.
- Πολλές `catch`! Μια για τις εξαιρέσεις του προγράμματος (*ProgXptn*), μια για τις *MyTmpltLibXptn*, μια για τις εξαιρέσεις της κάθε κλάσης και μια για οτιδήποτε άλλο! ◀

Prj03.7 char* ή *string*;

Τι γίνεται με τα κείμενα (λεκτικά); Θα τα χειριζόμαστε ως `char*` ή ως *string*; Πριν φθάσουμε στο «δια ταύτα» ξαναδιάβασε την §15.13.1 και ας καταγράψουμε μερικά βασικά σημεία.

Γενικώς, *ο τύπος string είναι βολικότερος και τα προγράμματά μας γράφονται πιο εύκολα. Αλλά, το κείμενο αποθηκεύεται σε δυναμική μνήμη.* Συνέπειες σε αυτά που είδαμε μέχρι τώρα:

- Αν στην κλάση εξαιρέσεων δηλώσουμε `string funcName` πρόσεξε τι μπορεί να συμβεί: Προσπαθείς ανεπιτυχώς να πάρεις δυναμική μνήμη και θέλεις να ρίξεις σχετική εξαίρεση. Αφού στην εξαίρεση θα χρειαστεί δυναμική μνήμη το πιο πιθανό είναι να ριχτεί (καινούρια) `std::bad_alloc`. Φυσικά, τέτοια εξαίρεση θα μπορεί να ριχτεί και κατά την (όποια) αντιγραφή της εξαίρεσης και –όπως θα μάθουμε αργότερα– να διακοπεί η εκτέλεση του προγράμματός σου.
- Αν στη *Student* είχαμε δηλώσει

```
string  sSurname;
string  sFirstname;
```

πώς θα κάναμε τη φύλαξη αυτού του μέλους στη *save()*; Θα κάναμε κάτι σαν:

```
char    local[sNameSz];
strncpy( local, sSurname.c_str(), sNameSz-1 );  local[sNameSz-1] = '\0';
bout.write( local, sNameSz );
strncpy( local, sFirstname.c_str(), sNameSz-1 );  local[sNameSz-1] = '\0';
bout.write( local, sNameSz-1 );
```

Δεν σου αρέσει; Στην §15.13.1 λέγαμε: «*Δεν θα μπορούσαμε να φυλάγουμε τις τιμές των ... μελών τύπου string με το ακριβές περιεχόμενο που έχουν κάθε φορά οποιοδήποτε μήκος και αν έχει; Ναι, αρκεί να φυλάγουμε και την τιμή του (κάθε) μήκους.*» Δες λοιπόν και αυτό:

```
unsigned int len( sSurname.length() );
char*        local( new char[len+1] );
strcpy( local, sSurname,c_str() );
bout.write( reinterpret_cast<const char*>(&len), sizeof(len) );
bout.write( local, len+1 );
delete[] local;
len = sFirstname,length();
local = new char[len+1];
strcpy( local, sFirstname );
bout.write( reinterpret_cast<const char*>(&len), sizeof(len) );
bout.write( local, len+1 );
```

Εδώ έχεις αυτό που λέμε «εγγραφές μεταβλητού μήκους».

Είναι φανερό ότι η λύση που έχουμε επιλέξει είναι απλούστερη και από τους δύο τρόπους που είδαμε εδώ.

Φυσικά τίποτε δεν μας εμποδίζει

- να έχουμε τα μέλη των κλάσεων δηλωμένα ως πίνακες `char` αλλά
- να κάνουμε όλους τους άλλους χειρισμούς στον τύπο *string*.

Αυτό κάναμε παραπάνω και έτσι θα συνεχίσουμε. Αλλά, όπως θα δούμε στο επόμενο κεφάλαιο, υπάρχει πιθανότητα –κατ' αρχήν τουλάχιστον– στις αντιγραφές τιμών τύπου *string* να εγερθεί εξαίρεση *bad_alloc*![4]

Prj03.8 Ένα Άλλο Σχέδιο για τις Κλάσεις

Η ιδέα να έχουμε στη *StudentInCourse* μια μέθοδο σαν την

```
void setIdNumCCode( Student& aStudent, string aCCode,
                    Course crsArr[], int nOfCourses );
```

ή, εναλλακτικώς, έναν δημιουργό σαν τον:

```
StudentInCourse::StudentInCourse( Student& aStudent, string aCCode,
                                  Course crsArr[], int nOfCourses )
// . . .
```

δεν είναι και τόσο «ορθόδοξη».

[4] Είπαμε: «*κατ' αρχήν*». Αν είναι να ψάχνουμε για εξαιρέσεις κάθε φορά που κάνουμε κάτι με τιμές τύπου *string*…

Θα έλεγε κανείς ότι καλύτερο θα ήταν να δηλώσουμε τον πίνακα crsArr *μέσα* στη *StudentInCourse*. Δηλαδή θα έχουμε αυτόν τον πίνακακ σε κάθε αντικείμενο *StudentInCourse*; Θα δούμε στα επόμενα μαθήματα ότι υπάρχει τρόπος να το αποφύγουμε.

Πάντως, ούτε αυτό είναι καλό αφού ένας τέτοιος πίνακας πιθανόν να μας χρειάζεται και για άλλες χρήσεις.

Ο συνήθης τρόπος χειρισμού αυτής της κατάστασης είναι να *έχουμε τον πίνακα* crsArr *έξω από την κλάση και μέσα στην κλάση StudentInCourse να έχουμε ένα βέλος προς αυτόν*:

```
class StudentInCourse
{
public:
// . . .
private:
   Course*       crsArr;
   unsigned int* pNOfCourses;
   unsigned int  sicSIdNum;      // αριθμός μητρώου
   char          sicCCode[8];    // κωδικός μαθήματος
}; // StudentInCourse
```

Αλλα το βέλος προς τον πίνακα δεν είναι σταθερό αφού αλλάζει κάθε τόσο με τη *renew()*. Θα το ξαναδούμε αργότερα...

Ερωτήσεις – Ασκήσεις

Α Ομάδα

Prj03-1 Ας πούμε ότι ο κωδικός μαθήματος έχει την εξής δομή:

- Δύο κεφαλαία γράμματα του ελληνικού αλφαβήτου που μπορεί να είναι "ΕΥ" ή "ΤΠ".
- Τέσσερα ψηφία του δεκαδικού συστήματος.
- Ένα κεφαλαίο γράμμα του ελληνικού αλφαβήτου που μπορεί να είναι "Θ" ή "Ε".

Κάνε τις απαραίτητες συμπληρώσεις/διορθώσεις σε αυτά που γράψαμε.

κεφάλαιο

20

Κλάσεις και Αντικείμενα – Βασικές Έννοιες

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα δούμε βασικές έννοιες σχετικά με αντικείμενα και κλάσεις και με τις υλοποιήσεις τους στη C++.

Προσδοκώμενα αποτελέσματα:
Όταν θα έχεις μελετήσει αυτό το κεφάλαιο θα μπορείς να σχεδιάσεις και να υλοποιήσεις ακόμη πιο απαιτητικές κλάσεις.

Έννοιες κλειδιά:

- *μήνυμα προς αντικείμενο*
- *διεπαφή αντικειμένου*
- *απόκρυψη πληροφορίας*
- *επιφόρτωση τελεστή εκχώρησης*
- *βέλος* `this`
- *συσχετίσεις κλάσεων*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Στο προηγούμενο κεφάλαιο ξεκινήσαμε με μια «γεφυρα» ανάμεσα σε αυτά που ξέραμε και στις νέες έννοιες και τεχνικές. Αυτό όμως έγινε μέσα στα πλαίσια της C++. Ο **αντικειμενοστρεφής** (object oriented) προγραμματισμός είναι κάτι ευρύτερο και πριν προχωρήσουμε στο τι κάνουμε και πώς το κάνουμε στη C++ καλό θα είναι να κάνουμε μια αφαίρεση και να δούμε μερικές έννοιες του αντικειμενοστρεφούς προγραμματισμού γενικότερα.

Ένα **αντικείμενο** (object) είναι ένα σύνολο **λειτουργιών** (operations) που δρούν πάνω σε μια κοινή **κατάσταση** (state). Μπορείς να καταλάβεις την κατάσταση ως το σύνολο τιμών μεταβλητών που βρίσκονται μέσα στο αντικείμενο και καλούνται **μεταβλητές στιγμιοτύπου** (instance variables) Η κατάσταση είναι κρυμμένη από τον έξω κόσμο. Ο έξω κόσμος έχει πρόσβαση στην κατάσταση μόνο μέσω των λειτουργιών. Μπορείς να καταλάβεις τις λειτουργίες ως συναρτήσεις που καλούνται **μέθοδοι** (methods). Το σύνολο των λειτουργιών καθορίζουν τη **διεπαφή** (interface) του και τη **συμπεριφορά** (behavior) του. (Wegner 1990)

Και τι είναι η **κλάση** (class); Η κλάση είναι ένα περίγραμμα, ένα σχέδιο με βάση το οποίο δημιουργούνται αντικείμενα. Έχει τις προδιαγραφές της κοινής συμπεριφοράς όλων των αντικειμένων αυτής της κλάσης. Οι μεταβλητές-μέλη της κλάσης είναι **δυνάμει** (potential) μεταβλητές σε αντιδιαστολή με αυτές του αντικειμένου που είναι **πραγματικές** (actual). Στην κλάση περιγράφονται και οι λειτουργίες με τις οποίες θα εξοπλισθεί και κάθε αντικείμενο-στιγμιότυπο της κλάσης. Περιγράφεται επίσης και η λειτουργία **δημιουργίας στιγμιοτύπου** (instance creation).

Για παράδειγμα, ο ορισμός

```
class Date
{ // I: (dYear > 0) && (0 < dMonth <= 12) &&
  //    (0 < dDay <= lastDay(dYear, dMonth))
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth()
                       const {  return static_cast<unsigned int>( dMonth );  }
   unsigned int getDay() const {  return static_cast<unsigned int>( dDay );  }
   void load( istream& bin );
   void save( ostream& bout ) const;
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

είναι ένα περίγραμμα που δείχνει πώς θα δημιουργούνται τα αντικείμενα τύπου *Date*. Η αναλλοίωτη *I* είναι μια συνθήκη με την οποίαν συμμορφώνεται η κατάσταση του κάθε αντικειμένου κλάσης *Date*. Οι μεταβλητές `Date::dYear`, `Date::dMonth`, `Date::dDay` είναι οι δυνάμει μεταβλητές της κλάσης. Η λειτουργία δημιουργίας στιγμιοτύπου περιγράφεται από τον δημιουργό (στην πραγματικότητα τέσσερις δημιουργοί).

Οι `Date::lastDay()` και `Date::leapYear()` τι είναι; Αυτό που ήδη είπαμε: βοηθητικές συναρτήσεις που μας χρειάζονται για την υλοποίηση των μεθόδων. Τις κρατούμε «μυστικές» όπως άλλωστε και όλα τα συστατικά της υλοποίησης.

Μετά τη δήλωση:

```
   Date d1( 2007, 5, 31 ), d2( 2004, 2, 29 );
```

έχουμε δύο αντικείμενα, τα *d1* και *d2*.

Η κατάσταση του πρώτου περιγράφεται από τις τιμές των (πραγματικών) μεταβλητών περίπτωσης *d1.dYear*, *d1.dMonth* και *d1.dDay* που είναι 2007, 5 και 31 αντιστοίχως. Κρατούμε αυτές τις μεταβλητές κρυμμένες αφού δηλώσαμε τις αντίστοιχες δυνάμει μεταβλητές σε περιοχή `private`.

Η κατάσταση του δεύτερου περιγράφεται από τις τιμές των (επίσης κρυμμένων) μεταβλητών στιγμιοτύπου *d2.dYear*, *d2.dMonth* και *d2.dDay* που είναι 2004, 2 και 29 αντιστοίχως.

Στην κατάσταση του *d1* δρουν οι μέθοδοι: *d1.getYear*`()`, *d1.getMonth*`()`, *d1.getDay*`()`, *d1.load*`(`*istream*`&)` και *d1.save*`(`*ostream*`&)`. Αυτές οι μέθοδοι βγάζουν προς τον έξω κόσμο πληροφορίες σχετικώς με την κατάσταση του *d1*. Πέρα από αυτές όμως υπάρχει και μια «κρυφή» αλλά πολύ ουσιαστική μέθοδος που επιτρέπει στον έξω κόσμο να αλλάξει την κατάσταση του αντικειμένου: ο *τελεστής εκχώρησης* (“`=`”). Για τα αντικείμενα τύπου *Date*, αυτή η μέθοδος έχει ορισθεί αυτομάτως από τον μεταγλωττιστή.

Όλο το κομμάτι που είναι `public` καθορίζει τη συμπεριφορά του κάθε αντικειμένου. Αυτό είναι το τμήμα *διεπαφής* του κάθε αντικειμένου κλάσης *Date*.

Στο τμήμα διεπαφής καθορίζονται τα είδη των **μηνυμάτων** (messages) που αναγνωρίζει, που μπορεί να δέχεται, ένα αντικείμενο από τον έξω κόσμο: είτε από το πρόγραμμα-πελάτη είτε από άλλα αντικείμενα που «ζούν» στο πρόγραμμα. Το αντικείμενο ανταποκρίνεται στα μηνύματα αυτά είτε αλλάζοντας την κατάστασή του είτε δίνοντας πληροφορίες σχετικώς με αυτήν. Για παράδειγμα, στέλνοντας στο *d1* το μήνυμα:

```
d1.load( bin );
```

του ζητούμε να αλλάξει την κατάστασή του διαβάζοντας νέες τιμές για τα μέλη από το ρεύμα *bin*.

Αν έχουμε δηλώσει:

```
Battery btr;
```

μπορούμε να στείλουμε τα μηνύματα *btr*`.`*powerDevice*`(double,double,bool&)`, *btr*`.`*maxTime*`( double )`, *btr*`.`*reCharge*`()`. Στέλνοντας στο *btr* το μήνυμα:

```
btr.powerDevice( 15*60, 4, ok );
```

ζητούμε από το *btr* να αλλάξει την κατάστασή του, αν αυτό είναι δυνατόν. Μέσω της *ok* το αντικείμενο μας γνωστοποιεί αν την άλλαξε.

Με το μήνυμα `btr.maxTime(8)` ζητούμε από το *btr* μια πληροφορία που έχει σχέση με την κατάστασή του.

Ένα αντικείμενο δεν ξέρει ποιος του στέλνει ένα μήνυμα που λαμβάνει, εκτός αν ο αποστολέας περιλαμβάνει τέτοια πληροφορία μέσα στο μήνυμα.

Από εδώ και πέρα . . .

Από εδώ και πέρα θα δούμε πώς γίνονται τα παραπάνω στη C++. Είδαμε ήδη μερικά στο προηγούμενο κεφάλαιο.

Θα δεις ότι η δουλειά μας έχει να κάνει κατά κύριο λόγο με τη διεπαφή:

- Τι θα βάλουμε εκεί και
- Πώς θα το υλοποιήσουμε.

Με το κλειστό μέρος δεν θα ασχοληθούμε; Όχι και πολύ αφού δεν έχει και πολλά πράγματα να βάλουμε εκεί:

- Όπως είδαμε, εκεί κρύβουμε τις βοηθητικές συναρτήσεις.
- Αργότερα θα δούμε ότι εκεί «κρύβουμε» μερικές μεθόδους επειδή θέλουμε να τις «αχρηστεύσουμε».

20.1 Άλλο Ένα Παράδειγμα: *BString*

Πριν προχωρήσουμε παρακάτω θα κάνουμε μια εισαγωγή σε άλλο ένα παράδειγμα κλάσης που θα χρησιμοποιούμε στη συνέχεια. Πρόκειται για την κλάση *BString* που θα μιμείται την `std::string`. Φυσικά, δεν θα υλοποιήσουμε την *BString* όπως ακριβώς είναι υλοποιημένη η *string*. Μάλλον θα κάνουμε την *BString* έτσι ώστε τα αντικείμενά της να συμπεριφέρονται όπως (περίπου) αυτά της *string*.

Για να μπορούμε να κρατούμε ως τιμή οποιονδήποτε ορμαθό θα χρησιμοποιήσουμε *δυναμικό πίνακα* με στοιχεία τύπου `char`.

```
char*  bsData;
```

Αυτό δεν σημαίνει κατ' ανάγκην ότι θα παίρνουμε πάντοτε τόση μνήμη όση μας χρειάζεται για να αποθηκεύσουμε την τιμή που θέλουμε.[1] Γιατί όχι; Ας πούμε ότι το πρόγραμμά μας δημιουργεί, με κάποιον τρόπο, 100 τιμές τύπου `char` (χαρακτήρες) και τους αποθηκεύει έναν προς ένα σε μια μεταβλητή τύπου *BString*. Ας πούμε ότι για κάθε χαρακτήρα που έρχεται ακολουθούμε τον αλγόριθμο της *renew()*: παίρνουμε νέα μνήμη, αντιγράφουμε το περιεχόμενο της παλιάς στη νέα, ανακυκλώνουμε την παλιά.

- Όταν έλθει ο *δεύτερος* χαρακτήρας παίρνουμε πίνακα δύο χαρακτήρων, *αντιγράφουμε έναν χαρακτήρα*, ανακυκλώνουμε πίνακα ενός χαρακτήρα.
- Όταν έλθει ο *τρίτος* χαρακτήρας παίρνουμε πίνακα τριών χαρακτήρων, *αντιγράφουμε δύο χαρακτήρες*, ανακυκλώνουμε πίνακα δύο χαρακτήρων.
- Όταν έλθει ο *τέταρτος* χαρακτήρας παίρνουμε πίνακα τεσσάρων χαρακτήρων, *αντιγράφουμε τρεις χαρακτήρες*, ανακυκλώνουμε πίνακα τριών χαρακτήρων, κ.ο.κ.

Συνολικώς: θα πάρουμε μνήμη 100 φορές, θα ανακυκλώσουμε μνήμη 99 φορές και θα κάνουμε 1+2+ ... +99 = 4950 αντιγραφές χαρακτήρων.

Αν, κάθε φορά που χρειάζεται να πάρουμε μνήμη, αντί να την αυξάνουμε κατά 1 την αυξάνουμε κατά 50, θα πάρουμε μνήμη –συνολικώς– 2 φορές, θα ανακυκλώσουμε μνήμη 1 φορά και θα κάνουμε 50 αντιγραφές χαρακτήρων.

Για να αποφύγουμε λοιπόν το διαρκές πάρε-δώσε με τον σωρό και τις αντιγραφές θα παίρνουμε κάτι περισσότερο.[2] Επομένως θα χρειαστούμε άλλο ένα μέλος

```
size_t bsReserved;
```

που θα μας δίνει το πλήθος θέσεων του πίνακα που έχουν παραχωρηθεί στο αντικείμενο.

Θα κάνουμε λοιπόν την εξής συμφωνία: *Οποτε έχουμε να πάρουμε μνήμη θα παίρνουμε πολλαπλάσιο των* 16 *ψηφιολέξεων*. Γιατί 16; Δεν υπάρχει κάποιος ιδιαίτερος λόγος. Εδώ θα κάνουμε απλώς επίδειξη ορισμένων τεχνικών και τίποτε παραπάνω. Αλλά, σε άλλες περιπτώσεις, όταν έχεις αντικείμενα με δυναμικούς πίνακες, μπορείς να βρεις πόσο είναι ένα καλό «κβάντο αύξησης».

Όταν αποθηκεύουμε μια τιμή (ορμαθό), σε ένα αντικείμενο κλάσης *BString*, πώς θα ξέρουμε πόσες θέσεις του δυναμικού πίνακα καταλαμβάνει;

- Ένας τρόπος είναι να βάζουμε, ως φρουρό, ένα `'\0'` στο τέλος του ορμαθού και να μετρούμε μέχρι να το βρούμε. Αυτό μας είναι χρήσιμο και σε κάτι άλλο: Η *string* έχει τη μέθοδο *c_str()* που επιστρέφει έναν ορμαθό χαρακτήρων της C, δηλαδή ένα βέλος προς την αρχή ενός πίνακα τύπου `char` με `'\0'` στο τέλος. Αν περιλάβουμε το `'\0'` στο τέλος η υλοποίησή της είναι πολύ απλή.

---

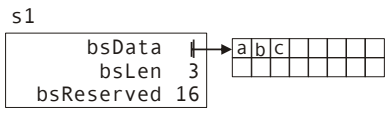
[1] Ξαναδές και αυτά που είπαμε στο Παράδ. 2 της §16.13.

[2] Η υλοποίηση της C++ χρησιμοποιεί έναν πιο πολύπλοκο τρόπο για τη διαχείριση της δυναμικής μνήμης. Ένα χαρακτηριστικό του, που πρέπει να αναφέρουμε, είναι το εξής: αντί να αυξάνει την «καπαρωμένη» μνήμη κατά ένα σταθερό «κβάντο» κάθε φορά την *διπλασιάζει*. Έτσι κάνει λιγότερες αντιγραφές (δες την §16.13.3).

- Ένας άλλος τρόπος είναι ο εξής: κρατούμε σε ένα χωριστό μέλος, *bsLen*, το μήκος του κειμένου που αποθηκεύουμε στο αντικείμενο.
- Ένας τρίτος τρόπος είναι να χρησιμοποιήσουμε δύο βέλη:

```
char* bsData;
char* bsEnd;
```

που θα δείχνουν την αρχή και το τέλος του κειμένου.



Σχ. 20-1 Η παράσταση ενός αντικειμένου *s1* κλάσης *BString* (*bsIncr* = 16.)

Πρόσεξε ότι ο δεύτερος και ο τρίτος τρόπος μας επιτρέπουν να έχουμε και χαρακτήρες `'\0'` στην τιμή που αποθηκεύουμε.

Θα χρησιμοποιήσουμε τον δεύτερο τρόπο που είναι ο πιο απλός και βολικός (Σχ. 20-1).

```
class BString
{
public:
// ...
private:
   enum { bsIncr = 16 };
   char*  bsData;
   size_t bsLen;
   size_t bsReserved;
}; // BString
```

Τι είναι εκείνο το *bsIncr*; Όπως βλέπεις είναι μια ακέραιη σταθερά με τιμή 16. Εεπειδή, όπως είπαμε θα παίρνουμε δυναμική μνήμη σε «φέτες» των 16 ψηφιολέξεων και αυτό θα φαίνεται στις υλοποιήσεις των μεθόδων, προτιμούμε να δώσουμε ένα όνομα στη σταθερά μας για να μην εμφανίζεται ως «μαγική σταθερά».

Με βάση αυτά η αναλλοίωτη –μέχρι τώρα– διαμορφώνεται ως εξής:

(0 <= *bsLen* < *bsReserved*) && (*bsReserved* % *bsIncr* == 0)

Αφού *bsLen* < *bsReserved* υπάρχει πάντοτε θέση για τον φρουρό (`'\0'`) όταν μας χρειαστεί.

Η κλάση εξαιρέσεων θα είναι κάπως έτσι:

```
struct BStringXptn
{
   enum { . . . };
   char funcName[100];
   int  errorCode;
   int  errorValue;
   BStringXptn( char* mn, int ec, int ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BStringXptn
```

20.1.1 Οι Απλές Μέθοδοι

Δες πόσο εύκολα υλοποιείται η *c_str*:

```
   const char* c_str() const
   {  bsData[bsLen] = '\0';  return bsData;  }
```

Βάζουμε τον φρουρό και επιστρέφουμε το βέλος. Το "`const`" *στην αρχή* είναι απαραίτητο διότι αλλιώς, όποιος πάρει το βέλος *bsData*, μπορεί να αλλάζει τον ορμαθό που είναι αποθηκευμένος.

Προσοχή όμως! Αν μέσα στο κείμενο υπάρχει ο χαρακτήρας `'\0'` και προσπαθήσουμε να επεξεργαστούμε αυτό που μας δίνει η *c_str* με τις συναρτήσεις *str…* η επεξεργασία θα τελειώνει στον πρώτο `'\0'`.

Παρατήρηση: ▶

Το "`const`" *στο τέλος* δεν μας απαγορεύει να βάλουμε τιμή `'\0'` στο τέλος του κειμένου (`bsData[bsLen]`). Μας απαγορεύει να αλλάξουμε τις τιμές των μελών *bsData* (βέλος), *bsLen*, *bsReserved*. Για να βάλουμε απαγόρευση αλλαγής του στόχου του *bsData* θα έπρεπε να είχαμε δηλώσει

```
const char* bsData;
```

Αλλά αν κάνουμε κάτι τέτοιο, κάθε φορά που θέλουμε να αλλάξουμε το κείμενο θα πρέπει να κάνουμε τυποθεώρηση **const**. Υπερβολές... ◀

Εύκολα μπορούμε να υλοποιήσουμε και τη μέθοδο *length()*:

```
size_t length() const {  return bsLen;  }
```

όπως και την *empty()*:

```
bool empty() const {  return ( bsLen == 0 );  }
```

Να σημειώσουμε ότι:""

- "*c_str*" είναι αυτό που θα λέγαμε "*getData*" και
- "*length*" αυτό που θα λέγαμε "*getLen*".
- Η *empty()* είναι ένα **κατηγόρημα** (predicate). Θα το ονομάζαμε "*isEmpty*" αλλά κρατούμε την ονοματολογία του τύπου *string*.

Παρατήρηση: ▶

Η *string* έχει και μια άλλη μέθοδο, συνώνυμη της *length()*, τη *size()*. Να την υλοποιήσουμε και αυτήν; Απλό:

```
size_t size() const {  return bsLen;  }
```

Σωστό! Αλλά όχι καλό. Δες μια άλλη επιλογή:

```
size_t size() const {  return length();  }
```

Μια άλλη επιλογή είναι: να ορίσουμε πρώτα τη *size()* και μετά τη *length()* με βάση τη *size()*. Πάντως, όποτε έχεις συνώνυμες μεθόδους

- Πρώτα ορίζεις τη μια αυτές.
- Μετά ορίζεις τις συνώνυμες με βάση αυτήν που ορίστηκε.

Έτσι, αν χρειαστεί να κάνεις κάποια αλλαγή αυτή θα πρέπει να γίνει σε ένα μόνο σημείο. Θα επανέλθουμε…◀

Ας πάμε τώρα να δούμε έναν δημιουργό. Όταν δηλώσουμε:

```
BString s0;
```

το *s0* θα έχει ως τιμή τον ορμαθό μηδενικού μήκους.

```
BString::BString()
{
   try {  bsData = new char[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BString::BString
```

Κατά τη συνήθειά μας, για να έχουμε ενιαία διαχείριση των εξαιρέσεων από τα αντικείμενα της κλάσης, πιάνουμε τη *bad_alloc* και ρίχνουμε μια *BStringXptn* (*allocFailed*).

Μια άλλη περίπτωση είναι η εξής: δίνουμε ως αρχική τιμή έναν ορμαθό της C (πίνακα `char` με φρουρό `'\0'`):

```
BString s1( "abc" );
```

Η περίπτωση αυτή καλύπτεται από τον δημιουργό:

```
BString( const char* rhs );
```

Η υλοποίησή του γίνεται ως εξής:

```
BString::BString( const char* rhs )
{
```

```
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   try {  bsData = new char [bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BString
```

Αν γράψουμε την επικεφαλίδα του δεύτερου ως εξής:

```
   BString( const char* rhs="" );
```

ο δεύτερος γίνεται «2 σε 1» και ο πρώτος δεν μας χρειάζεται.

Εδώ πρόσεξε τα εξής:

1. Για να υπολογίσουμε το μήκος του ορίσματος χρησιμοποιούμε μια *εσωτερική μέθοδο*:[3]

```
size_t BString::cStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *p != '\0' )  ++p;
   return p-cs;
} // BString::cStrLen
```

Πρόσεξε ότι στο μήκος δεν περιλαμβάνεται η θέση για το `'\0'`. Γιατί δεν χρησιμοποιούμε τη *strlen*; Γενικώς δεν χρησιμοποιούμε τις συναρτήσεις διαχείρισης ορμαθών της C. Αργότερα θα καταλάβεις τον λόγο.

2. Πόσες ψηφιολέξεις μνήμης παίρνουμε; Όπως είπαμε θα παίρνουμε πάντοτε μνήμη που είναι ακέραιο πολλαπλάσιο των *bsIncr* == 16 ψηφιολέξεων. Εδώ θα πρέπει να πάρουμε μνήμη ίση με το πρώτο πολλαπλάσιο των 16 ψηφιολέξεων που είναι μεγαλύτερο από το *bsLen*+1 (για τον φρουρό).

3. Ο *BStringXptn*::*allocFailed* είναι ο πρώτος κωδικός σφάλματος που χρησιμοποιούμε. Τώρα δηλαδή θα έχουμε:

```
   enum { allocFailed };
```

20.1.2 … Και η Μέθοδος *at()*

Ας δούμε και μία ακόμη απλή μέθοδο, την *at*, για την οποίαν έχουμε αφήσει και κάποια «χρέη» από παλιά. Γράφαμε στην §10.10: «*Αυτό που γράψαμε παραπάνω μπορεί να γραφεί και ως εξής:*

```
   if ( s3.at(0) == 'κ' )  s3.at( 0 ) = 'K';
```

Αυτός ο τρόπος είναι ασφαλέστερος από αυτόν με τον δείκτη, διότι αν κάνεις λάθος και βγεις έξω από τα όρια της μεταβλητής σου θα ειδοποιηθείς, όπως θα μάθουμε αργότερα.» Τώρα πια μπορούμε να πούμε ότι θα ειδοποιηθούμε με μια εξαίρεση κλάσης *out_of_range* που δηλώνεται στο `stdexcept`. Δοκίμασε το:

```
#include <iostream>
#include <stdexcept>
#include <string>

using namespace std;

int main()
{
   string s3( "abc" );

   try
   {
      s3.at( 7 ) = '@';
```

[3] Κάτι σου θυμίζει; Είναι η δεύτερη μορφή της *myStrLen* από το παράδ. 1 της §12.3.3.

```
   }
   catch( out_of_range& x )
   {
      cout << "out_of_range" << endl;
   }
}
```

Τώρα πρόσεξε το εξής: αν η *s3* δηλωθεί ως αντικείμενο κλάσης *BString*, η "`s3.at(m)`" θα πρέπει να μας δίνει όχι απλώς την τιμή του `s3.bsData[m]` αλλά το ίδιο το στοιχείο ώστε να μπορούμε να αλλάξουμε την τιμή του. Πώς γίνεται αυτό; Από την §13.4 αντιγράφουμε: «*Βάζοντας ως τύπο επιστροφής της συνάρτησης "*`int&`*" η συνάρτηση επιστρέφει το στοιχείο με τη μέγιστη τιμή (τιμή-l) και όχι απλώς την τιμή του.*» Αυτή είναι η λύση για το πρόβλημά μας:

```
char& BString::at( int k ) const
{
   if ( k < 0 || bsLen <= k )
      throw BStringXptn( "at", BStringXptn::outOfRange, k );
   return bsData[k];
} // BString::at
```

Η συνάρτηση επιστρέφει τιμή τύπου "`char&`" –στην περίπτωσή μας το στοιχείο `bsData[k]`– και όχι "`char`".

Η *at* μπορεί να ρίξει εξαίρεση κλάσης *BStringXptn* (με κωδικό *outOfRange*).

Να επιστήσουμε πάντως την προσοχή σου στο εξής: Με την *at* δίνουμε στο πρόγραμμα που φιλοξενεί το αντικείμενο δυνατότητα να αλλάζει την τιμή του. Στη συνέχεια θα δεις ότι πολύ συχνά θα γράφουμε μεθόδους που βγάζουν τιμή με τύπο αναφοράς απλώς και μόνο για να αποφεύγουμε μια αντιγραφή. Σε τέτοιες περιπτώσεις μάλλον θα χρειάζεται και κάποιο "`const`".

20.1.3 Ο Καταστροφέας

Τώρα έχουμε και ένα άλλο πρόβλημα: Ας πούμε ότι σε μια συνάρτηση έχουμε δηλώσει και χρησιμοποιήσει κάποιο αντικείμενο κλάσης *BString*. Τι θα γίνει όταν τελειώσει η εκτέλεση της συνάρτησης; Θα «καταστραφούν» αυτομάτως τα μέλη αλλά δεν θα ανακυκλωθεί η (δυναμική) μνήμη –που δείχνει το μέλος *bsData*– όπου έχουμε αποθηκεύσει το κείμενο. Αυτό θα το φροντίσει ένας **καταστροφέας** (destructor) που θα γράψουμε εμείς. Ενώ, όπως είδες, μπορεί να έχουμε πολλούς δημιουργούς έχουμε έναν και μόνο καταστροφέα που το όνομά του είναι "`~`", *όνομα κλάσης*:

```
BString::~BString()
{
   delete[] bsData;
} // BString::~BString
```

Για καταστροφείς, όπως και για δημιουργούς, θα μιλήσουμε εκτενώς στη συνέχεια.

20.2 Ο Δημιουργός Αντιγραφής

Ας πούμε ότι δίνουμε:

```
      BString s1( "abc" );
      cout << "s1: " << s1.c_str() << endl;

      BString s2( s1 ), s3( s1.c_str() );
      cout << "s2: " << s2.c_str() << endl;
      cout << "s3: " << s3.c_str() << endl;
```

Αποτέλεσμα:

```
s1: abc
s2: abc
```

```
s3: abc
```

Παρατήρηση: ▶

Αν προτιμάς να γράφεις

```
BString s2 = s1,
        s3 = s1.c_str();
```

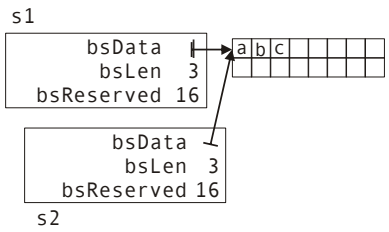
κανένα πρόβλημα· θα πάρεις τα ίδια ακριβώς απότελέσματα.◀

Μέχρι εδώ κανένα πρόβλημα. Να σημειώσουμε μόνον ότι η *s3* παίρνει τιμή με τον ίδιο τρόπο που παίρνει και η *s1*. Πρόσεξε τώρα τη συνέχεια:

```
   s1.at( 1 ) = 'v';
   cout << "s1: " << s1.c_str() << endl;
   cout << "s2: " << s2.c_str() << endl;
   cout << "s3: " << s3.c_str() << endl;
```

Αποτέλεσμα:

```
s1: avc
s2: avc
s3: abc
```



Σχ. 20-2 Η παράσταση της κατάστασης των *s1* και *s2* μετά τη "`BString s2(s1));`"

Εμείς αλλάξαμε την τιμή του *s1*· γιατί άλλαξε και η τιμή του *s2*; Δες το Σχ. 20-2 για να καταλάβεις: Ο μεταγλωττιστής μη έχοντας άλλες οδηγίες –ειδικές για την περίπτωση αυτή– αντίγραψε στα μέλη του νέου αντικειμένου (*s2*) τις τιμές των αντίστοιχων μελών του *s1*. Το πρόβλημα δημιουργείται από την αντιγραφή του βέλους `s1.bsData` στο `s2.bsData` που κάνει τα δύο αντικείμενα να έχουν ως τιμή το ίδιο κείμενο.

Να θυμίσουμε ότι αυτό το πρόβλημα –αλλά για τον τελεστή εκχώρησης– το πρωτοείδαμε στην §16.8 χωρίς να το λύσουμε. Τώρα θα το λύσουμε. Πώς; Με τον ίδιο τρόπο που κάνουμε αντιγραφή χωρίς πρόβλημα στην *s3*: Θα ορίσουμε ειδικώς για τον ορισμό αντικειμένου με αρχική τιμή άλλο αντικείμενο του ίδιου τύπου έναν δημιουργό, που λέγεται **δημιουργός αντιγραφής** (copy constructor), που θα παίρνει για το νέο αντικείμενο-αντίγραφο (στο παράδειγμά μας: *s2*) όση δυναμική μνήμη έχει το πρωτότυπο (στο παράδειγμά μας: *s1*):

```
BString::BString( const BString& rhs )
{
   bsReserved = rhs.bsReserved;

   try {  bsData = new char[bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BString
```

Για τον δημιουργό αντιγραφής θα τα πούμε στο επόμενο κεφαλαιο. Εδώ θα σημειώσουμε τα εξής:

- Τα μέλη του αντικειμένου που δημιουργείται (στο παράδειγμά μας: *s2*) είναι *bsData*, *bsLen* και *bsReserved*, ενώ *rhs* είναι το πρωτότυπο (στο παράδειγμά μας: *s1*) με μέλη *rhs*.*bsData*, *rhs*.*bsLen* και *rhs*.*bsReserved*.
- Γιατί δεν κάνουμε έναν έλεγχο για το πόση μνήμη χρειάζεται; Μπορεί να χρειάζεται λιγότερη από *rhs*.*bsReserved*. Κάνε το έτσι αν θέλεις. Το δικό μας σκεπτικό είναι απλό: «το αντίγραφο πρέπει να είναι (πιστό) αντίγραφο.»

20.3 Η Πρόσβαση στα Μέλη "private"

Πριν προχωρήσουμε στο παρεμφερές πρόβλημα του τελεστή εκχώρησης θα πρέπει να επισημάνουμε το εξής: Το νέο αντικείμενο έχει πρόσβαση στα μέλη του πρωτότυπου (*rhs*) παρ' όλο που αυτά είναι σε περιοχή **private**. Γενικώς:

- ***Όλες οι μέθοδοι ενός αντικειμένου κλάσης T έχουν προσβαση σε όλα τα μέλη οποιουδήποτε άλλου αντικειμένου κλάσης T.***

20.4 Ο Τελεστής Εκχώρησης "="

Αφού δηλώσουμε:

```
BString s1( "abc" ), s2;
```

ζητούμε:

```
cout << "s1: " << s1.c_str() << endl;

s2 = s1;
cout << "s2: " << s2.c_str() << endl;

s1.at( 1 ) = 'v';
cout << "s1: " << s1.c_str() << endl;
cout << "s2: " << s2.c_str() << endl;
```

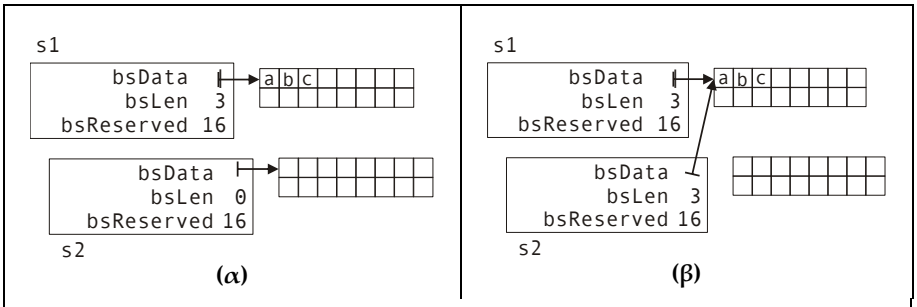
Αποτέλεσμα:

```
s1: abc
s2: abc
s1: avc
s2: avc
```

Δεν είναι σωστό· αυτό όμως –μετά από αυτά που είδαμε στην προηγούμενη παράγραφο– δεν είναι έκπληξη. Όπως καταλαβαίνεις έχουμε το ίδιο πρόβλημα αλλά σε χειρότερη μορφή: τώρα έχουμε και διαρροή μνήμης (Σχ. 20-3).

- Και εδώ έχουμε αντιγραφή στα μέλη του αντικειμένου *s2* των τιμών των αντίστοιχων μελών του *s1*.
- Και εδώ έχουμε το πρόβλημα από την αντιγραφή του βέλους `s1.bsData` στο `s2.bsData` που κάνει τα δύο αντικείμενα να έχουν ως τιμή το ίδιο κείμενο. Αλλά τώρα το `s2.bsData` έδειχνε τη δυναμική μνήμη που ήδη είχε πάρει το *s2*. Αυτή η μνήμη παραμένει δεσμευμένη από το πρόγραμμά μας αλλά δεν έχουμε εργαλείο να τη χειριστούμε.

Μια και ο τελεστής εκχώρησης είναι απαραίτητος για τον τύπο *BString* θα πρέπει να τον επιφορτώσουμε ώστε να δουλεύει σωστά. Πριν προχωρήσεις ξαναδιάβασε αυτά που λέγαμε για την επιφόρτωση των τελεστών εκχώρησης στην §14.6.3.



Σχ. 20-3 Στο (α) βλέπουμε μια παράσταση της κατάστασης των *s1* και *s2* μετά την δήλωση των αντικειμένων. Στο (β) βλέπουμε την κατάσταση των *s1* και *s2* μετά την εντολή εκχώρησης. Όπως φαίνεται, δεν υπάρχει βέλος για τον χειρισμό της δυναμικής μνήμης που έχει δεσμευθεί για το *s2*.

Αν έχουμε:

```
BString s1( "abc" ), s2, s3;
```

η

```
s2 = s1;
```

θα πρέπει να δίνει ως τιμή στο *s2* την τιμή του *s1* και να επιστρέφει αυτήν την νέα τιμή του *s2* ώστε, αν θέλουμε, να τη χρησιμοποιήσουμε· για παράδειγμα θα μπορούσαμε να δώσουμε: "`s3 = (s2 = s1)`". Πώς θα πρέπει να κάνουμε την επιφόρτωση;

Μην προσπαθήσεις να εφαρμόσεις αυτά που λέμε στην §14.6.4 (σε επόμενο κεφάλαιο θα δώσουμε νέες οδηγίες): Ο τελεστής εκχώρησης επιφορτώνεται ως *μέθοδος*:

```
BString& operator=( const BString& rhs );
```

Τι θα κάνει αυτή η μέθοδος;

```
«Καθάρισε» την παλιά τιμή
Πάρε τη μνήμη που χρειάζεσαι
Αντίγραψε την τιμή του rhs
Επίστρεψε ως τιμή το αντικείμενο
```

Και αν η «*Πάρε τη μνήμη που χρειάζεσαι*» αποτύχει; Στην περίπτωση αυτή

- θα ρίξουμε εξαίρεση
- θα πρέπει να διασφαλίσουμε ότι το αντικείμενό μας κρατάει την τιμή που έχει (όποια και αν είναι αυτή.)

Επομένως θα πρέπει να αλλάξουμε το σχέδιο: Πρώτα παίρνουμε τη μνήμη και μετά «καθαρίζουμε» την παλιά τιμή.

```
Πάρε τη μνήμη που χρειάζεσαι
if ( απέτυχες ) throw …
«Καθάρισε» την παλιά τιμή
Αντίγραψε την τιμή του rhs
return το αντικείμενο
```

Πρόσεξε τώρα κάτι ιδιαίτερο: Τι θα γίνει αν δοθεί η (φαινομενικώς ανόητη αλλά καθ' όλα νόμιμη) εντολή "`s1 = s1`"; Εδώ, *bsData* και *rhs*`.bsData` είναι το ίδιο πράγμα. Όταν έλθει η ώρα να αντιγράψεις τον πίνακα στον εαυτό του ο πίνακας έχει ήδη ανακυκλωθεί! Φυσικά, αν πρέπει να εκτελεσθεί αυτή η παράξενη «αυτοεκχώρηση», δεν χρειάζεται να κάνουμε οτιδήποτε από τα παραπάνω· αρκεί να επιστρέψουμε ως τιμή το αντικείμενο:

```
if ( !αυτοεκχώρηση )
{
   Πάρε τη μνήμη που χρειάζεσαι
   if ( απέτυχες ) throw …
   «Καθάρισε» την παλιά τιμή
   Αντίγραψε την τιμή του rhs
}
return το αντικείμενο
```

Υπάρχουν όμως δύο ανοικτά προβλήματα:

- Πώς καταλαβαίνουμε ότι έχουμε «αυτοεκχώρηση»;
- Πώς υλοποιείται η ψευδοεντολή «***return*** *το αντικείμενο*»;

Η λύση και στα δύο προβλήματα δίνεται με το βέλος **this**, για το οποίο μιλούμε στην επόμενη παράγραφο. Κάθε αντικείμενο έξοπλίζεται (αυτομάτως) με ένα βέλος, το **this** *που δείχνει το ίδιο το αντικείμενο*. Έτσι:

- Καταλαβαίνουμε ότι έχουμε «αυτοεκχώρηση» αν ισχύει η `&rhs == this`.
- Η ψευδοεντολή «***return*** *το αντικείμενο*» υλοποιείται ως:

```
return *this;
```

Για να παρακολουθήσεις τη μετάφραση του σχεδίου σε C++ πάρε υπόψη σου το εξής: Ο τελεστής επιφορτώνεται με μέθοδο του «αριστερού μέλους» της εκχώρησης. Έτσι, η "`s2 = s1`" μπορεί να γραφεί και ως "`s2.operator=(s1)`".

- Η «***if*** *(* **!***αυτοεκχώρηση)*» γίνεται:

```
if ( &rhs != this )
```

- Για να μεταφράσουμε τις «*Πάρε τη μνήμη που χρειάζεσαι*» και «***if*** *(απέτυχες)* ***throw*** *…*» γράφουμε:

```
   char* tmp;
   try {  tmp = new char[rhs.bsReserved];  }
   catch( bad_alloc& )
   { throw BStringXptn( "operator=", BStringXptn::allocFailed ); }
```

Εδώ πρόσεξε ότι παίρνουμε τη μνήμη με ένα βοηθητικό βέλος (*tmp*). Ακόμη, κατά τη συνήθειά μας, βάλαμε εντολές που πιάνουν πιθανή εξαίρεση *bad_alloc* και ρίχνουν δική μας *BStringXptn* (*allocFailed*). Οι εντολές που ακολουθούν δεν θα εκτελεσθούν αν δεν παραχωρηθεί η μνήμη που ζητούμε αφού στην περίπτωση αυτή θα ριχτεί εξαίρεση.

- Η «*"Καθάρισε" την παλιά τιμή*» γίνεται

```
   delete[] bsData;
```

- Η «*Αντίγραψε την τιμή του rhs*» γίνεται

```
   bsData = tmp;
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
```

Και η μέθοδος που επιφορτώνει τον τελεστή "=" για την κλάση μας:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )  // if ( !αυτοεκχώρηση )
   {
   // Πάρε τη μνήμη που χρειάζεσαι
      char* tmp;
      try {  tmp = new char[rhs.bsReserved];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "operator=", BStringXptn::allocFailed );  }
   // "Καθάρισε" την παλιά τιμή
      delete[] bsData;
   // Αντίγραψε την τιμή του rhs
      bsData = tmp;
      bsReserved = rhs.bsReserved;
      for ( int k(0); k < rhs.bsLen; ++k ) bsData[k] = rhs.bsData[k];
      bsLen = rhs.bsLen;
   }
   return *this;
} // BString::operator=
```

Αν, μετά την επιφόρτωση, δοκιμάσουμε το πρόγραμμά μας παίρνουμε:

```
s1: abc
s2: abc
s1: avc
s2: abc
```

20.4.1 Η Μέθοδος *assign()*

Η (*std*::)*string* έχει και τη μέθοδο *assign()* που δεν είναι κάτι διαφορετικό από τον `operator=()`. Την παίρνουμε αλλάζοντας μόνον την επικεφαλίδα της `operator=()`.

```
BString& BString::assign( const BString& rhs )
{
   if ( &rhs != this )  // if ( !αυτοεκχώρηση )
   {
   // Πάρε τη μνήμη που χρειάζεσαι
      char* tmp;
      try {  tmp = new char[rhs.bsReserved];  }
      catch( bad_alloc& )
```

```
      {  throw BStringXptn( "assign", BStringXptn::allocFailed );  }
   // "Καθάρισε" την παλιά τιμή
      delete[] bsData;
   // Αντίγραψε την τιμή του rhs
      bsData = tmp;
      bsReserved = rhs.bsReserved;
      for ( int k(0); k < rhs.bsLen; ++k )
         bsData[k] = rhs.bsData[k];
      bsLen = rhs.bsLen;
   }
   return *this;
} // BString::assign
```

Αυτή η μορφή είναι καλή για λόγους επίδειξης αλλά έχουμε το ίδιο πρόβλημα που είχαμε με τις *length()* και *size()*: τη συνωνυμία και η εγγύηση για το ότι *assign()* και `operator=()` κάνουν ακριβώς την ίδια δουλειά είναι ο **inline** ορισμός:

```
class BString
{
public:
// . . .
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs ) { return (*this = rhs); }
// . . .
private:
// . . .
}; // BString
```

και όχι το "copy/paste". Γιατί; Διότι πέρα από το αρχικό γράψιμο υπάρχουν και οι διορθώσεις, οι προσαρμογές και τα παρόμοια που μόνο η δεύτερη μορφή εγγυάται ότι θα γίνουν και στις δύο μορφές αυτομάτως. Και δεν θα «γίνει μπέρδεμα» αν πέσει καμιά εξαίρεση που δίνει για προέλευση "`operator=`" ενώ το πρόγραμμα καλούσε την *assign()*; Ε, μη πνίγεσαι σε μια κουταλιά νερό...

20.4.2 Και Μια Εκχώρηση που *δεν* Ορίσαμε

Αν έχουμε δηλώσει:

```
   std:string s0;
```

επιτρέπεται να δώσουμε:

```
   s0 = "what\'s this\?";  cout << "    s0: " << s0 << endl;
```

και να πάρουμε:

```
    s0: what's this?
```

Για να αποκτήσουμε αυτήν τη δυνατότητα και στον *BString* θα πρέπει να (ξανα)επιφορτώσουμε τον "`=`" με μια:

```
   BString& operator=( const char* rhs );
```

Δεν είναι απαραίτητο· δοκίμασε τις

```
   BString s2;

   s2 = "what\'s this\?";  cout << "s2: " << s2.c_str() << endl;
```

Θα πάρεις:

```
s2: what's this?
```

Πώς το καταφέραμε αυτό; Η εκχώρηση, πριν κάνει αυτά που περιγράφουμε στην επιφόρτωση του "`=`", κάνει και κάτι άλλο: «*η τιμή* (της παράστασης που έχουμε δεξιά) *μετατρέπεται στον τύπο της μεταβλητής* (που έχουμε αριστερά)» αν αυτό είναι εφικτό (§2.2, §11.3). Δηλαδή, εκτελείται η:

```
   s2 = static_cast<BString>( "what\'s this\?" );
```

ή, στην περίπτωσή μας:

```
   s2 = BString( "what\'s this\?" );
```

Η μετατροπή τιμής "`const char*`" σε τιμή *BString* είναι εφικτή και γίνεται με τον ερήμην («2 σε 1») δημιουργό της κλάσης. Για να πεισθείς άλλαξε (προσωρινώς) τον δημιουργό ως εξής:

```
BString::BString( const char* rhs )
{
cout << "In default constructor; rhs: " << rhs << endl;
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   try {  bsData = new char [bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BString
```

και θα πάρεις:

```
In default constructor; rhs:
In default constructor; rhs: what's this?
s2: what's this?
```

Η πρώτη γραμμή προέρχεται από τη δήλωση της *s2* και η δεύτερη από τη μετατροπή.

20.5 Το Βέλος "`this`"

Στην προηγούμενη παράγραφο χρειαστήκαμε, μέσα σε μια μέθοδο, βέλος προς το αντικείμενο-ιδιοκτήτη της μεθόδου. Πώς το βρίσκουμε αυτό; Ένας τρόπος θα ήταν να πάρουμε ένα βέλος προς το πρώτο μέλος του αντικειμένου, αφού, όπως ξέρουμε από τις δομές, τα δύο βέλη είναι ίσα. Φυσικά, για να μπορέσουμε να το χρησιμοποιήσουμε θα πρέπει να κάνουμε την κατάλληλη (ερμηνευτική) τυποθεώρηση, π.χ.:

```
reinterpret_cast<BString*>( &bsData )
```

Κάτι τέτοιο δεν είναι και τόσο κομψό, είναι δύσχρηστο και *κανείς δεν σου εγγυάται ότι ισχύει* (§15.6)· ας το έχεις δοκιμάσει σε πολλούς μεταγλωττιστές.

Η C++ μας δίνει ένα σίγουρο εργαλείο. Στον ορισμό οποιασδήποτε κλάσης μπορεί να χρησιμοποιηθεί το βέλος `this` (Σχ. 20-4).

♦ ***Για οποιοδήποτε αντικείμενο μιας κλάσης το `this` είναι βέλος προς αυτό το αντικείμενο.***

Ας πούμε, για παράδειγμα, ότι έχουμε ορίσει μια κλάση:

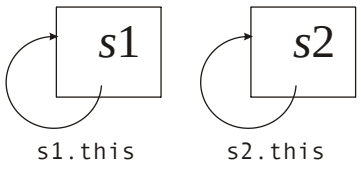
```
class AClass
{
public:
// . . .
   void printthis()
   { cout << this; }
// . . .
private:
// . . .
};
```

και δηλώνουμε:

```
   AClass a, b;
```

Στη συνέχεια δίνουμε τις εντολές:

```
   a.printthis();  cout << "  " << &a << endl;
   b.printthis();  cout << "  " << &b << endl;
```



Σχ. 20-4 Τα βέλη `this` για δύο αντικείμενα. Το βέλος `this` χρησιμοποιείται στις μεθόδους για να δείχνει το ίδιο το αντικείμενο. **Προσοχή!** Τα `s1.this` και `s2.this` είναι μόνο για να δείξουμε την «ιδιοκτησία» του βέλους από το αντικείμενο στο σχήμα και όχι για να το γράψεις στο πρόγραμμά σου.

Δηλαδή, ζητούμε να δούμε τι δείχνει το βέλος `this` για το *a* και τη διεύθυνση του *a*. Τα ίδια ζητούμε και για το *b*. Απότέλεσμα:

```
0x0064fdfc  0x0064fdfc
0x0064fdf4  0x0064fdf4
```

που σημαίνει ότι για το *a* το `this` δείχνει το ίδιο το *a*. Παρόμοια ισχύουν και για το *b*.

20.6 Επιστρέφουμε Τύπο Αναφοράς;

Ο τελεστής εκχώρησης ("`=`") και η *at()* έχουν ένα κοινό χαρακτηριστικό: επιστρέφουν τύπο αναφοράς:

- Ο "`=`" επιστρέφει αναφορά σε ολόκληρο το αντικείμενο.
- Η *at()* επιστρέφει *αναφορά σε μια συνιστώσα του αντικειμένου*.

  Τέτοιες μέθοδοι γενικώς θεωρούνται επικίνδυνες. Γιατί;
- Παράδειγμα: Αν στο πρόγραμμά σου αντί για "`x = y = z`" ή "`x = (y = z)`" γράψεις "`(x = y) = z`" θα αλλάξει η τιμή του *x* αλλά όχι του *y* (όπως θα ήθελες). Αυτό είναι ένα δύσκολο προγραμματιστικό λάθος που μπορεί να σου πάρει αρκετό χρόνο μέχρι να το βρεις. Πάντως, σε κάθε περίπτωση, αυτό που αλλάζει είναι η τιμή ολόκληρου αντικειμένου με τους κανόνες που έχουμε θέσει για αυτές τις αλλαγές.
- Η *at()* ανοίγει μια «παράπλευρη πόρτα» για να μπεί το πρόγραμμα-πελάτης στο αντικείμενο και να κάνει τροποποιήσεις ανεξέλεγκτα και –πιθανόν– να καταστρέψει το αντικείμενο.

  Φυσικά, αν βάζαμε αναφορές "`const`" τότε γλιτώνουμε από όλα αυτά. Γενικώς λοιπόν:[4]

♦ ***Απόφευγε να γράφεις μεθόδους που επιστρέφουν αναφορές χωρίς "`const`".***

  Εδώ εμείς βάλαμε αυτές τις αναφορές

- για να είμαστε σύμφωνοι με τη «φιλοσοφία της C++ (C)» (για τον "`=`") ή
- για να αντιγράψουμε τη λειτουργία της *std*`::`*string* (για την *at()*). Πάντως, θα πρέπει να παραδεχτούμε ότι η δυνατότητα διαχείρισης της τιμής ενός *BString* με τόσο απλό (και οικείο) τρόπο είναι μεγάλο δέλεαρ για να πάρουμε το ρίσκο.

20.7 Μια Κλάση για Διαδρομές Λεωφορείων

Θα δούμε τώρα άλλο ένα παράδειγμα κλάσης της οποίας κάθε αντικείμενο έχει έναν δυναμικό πίνακα του οποίου τα στοιχεία είναι αντικείμενα μιας άλλης κλάσης. Το πρόβλημα:

> *Τα λεωφορεία κάποιου ΚΤΕΛ εξυπηρετούν το επιβατικό κοινό με βάση συγκεκριμένες διαδρομές. Μια διαδρομή έχει ως χαρακτηριστικά την αρχή, το τέλος και –όπου χρειάζεται– κάποιο ενδιάμεσο σημείο (π.χ. Αθήνα – Σούνιο από Μεσόγεια). Έχει ακόμη έναν κωδικό διαδρομής (φυσικός αριθμός).[5] Σε κάθε διαδρομή υπάρχουν συγκεκριμένες στάσεις. Για κάθε μια από αυτές κρατούμε την απόσταση από την αφετηρία σε km και το κόμιστρο από την αφετηρία σε €. Δεν υπάρχουν στάσεις με το ίδιο όνομα. Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία. Σε μια διαδρομή θα πρέπει να έχουμε δυνατότητα να προσθέσουμε νέα σταση και να διαγράψουμε στάση που*

---

[4] Ο *Κανόνας OBJ35* του (CERT 2009) λέει: «*Do not return references to private data*» με σκεπτικό: Αν μια μέθοδος κλάσης επιστρέφει αναφορά ή βέλος προς εσωτερικά δεδομένα, αυτά μπορεί να αλλαχτούν από μη αξιόπιστο κώδικα.

[5] Τα *αρχή*, *τέλος* και *ενδιάμεσο* απαρτίζουν το κλειδί μιας διαδρομής. Ο κωδικός είναι ένα *υποκατάστατο* κλειδί (§15.5.1).

υπάρχει. Θέλουμε μια κλάση, ας την πούμε Route, που κάθε της αντικείμενο θα περιγράφει μια τέτοια διαδρομή.

Στο αρχείο kvadra.txt υπάρχουν τα στοιχεία μιας διαδρομής γραμμένα ως εξής (με το `'\t'` *παριστάνουμε τον οριζόντιο στηλοθέτη (tab)):*

```
102
ΚΑΒΑΛΑ\tΔΡΑΜΑ\t
1Η ΑΓ. ΑΘΑΝΑΣΙΟΥ\t22.1\t3.2
2Η ΑΓ. ΑΘΑΝΑΣΙΟΥ\t22.6\t3.2
 . . .
ΔΙΟΙΚΗΤΗΡΙΟ\t34.5\t5
1Η ΔΟΞΑΤΟΥ\t26.7\t4
 . . .
```

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα αποθηκεύει το περιεχόμενό του σε ένα αντικείμενο κλάσης Route. Στη συνέχεια:

1. Θα εισάγει μια νέα στάση με όνομα «`ΜΑΡΜΑΡΑ`*», απόσταση από την αφετηρία 25.5 km και κόμιστρο € 3.7.*

2. Θα αλλάζει σε «`ΔΙΑΣΤΑΥΡΩΣΗ`*» το όνομα της στάσης «*`ΣΤΑΥΡΟΣ`*» –αν υπάρχει.*

3. Θα φυλάγει το τροποιημένο δρομολόγιο στο αρχείο kvadranew.txt στην ίδια μορφή που ήταν και το αρχικό.

20.7.1 Η Κλάση για τις Στάσεις

Το πρώτο που θα κάνουμε είναι να ορίσουμε μια κλάση για τις στάσεις. Θα είναι κάπως έτσι:

```
class RouteStop
{
public:
// . . .
private:
   string sName;  // όνομα στάσης
   float  sDist;  // απόσταση από αφετηρία σε km
   float  sFare;  // τιμή εισιτηρίου από την αφετηρία
}; // RouteStop
```

Γιατί βάλαμε "`class`" και όχι "`struct`"; Διότι θα πρέπει να έχουμε

$$sDist \geq 0 \wedge sFare \geq 0$$

Έχουμε δηλαδή *μη τετριμμένη αναλλοίωτη*.

Κατ' αρχάς να ορίσουμε έναν δημιουργό. Δηλώνουμε:

```
   RouteStop( string aName="", float aDist=0, float aFare=0 );
```

και ορίζουμε:

```
RouteStop::RouteStop( string aName, float aDist, float aFare )
{
   if ( aName.empty() && (aDist > 0 || aFare > 0) )
      throw RouteStopXptn( "RouteStop", RouteStopXptn::noName );
   if ( aDist < 0 )
      throw RouteStopXptn( "RouteStop",
                           RouteStopXptn::negDist, aDist );
   if ( aFare < 0 )
      throw RouteStopXptn( "RouteStop",
                           RouteStopXptn::negFare, aFare );
   sName = aName;
   sDist = aDist;
   sFare = aFare;
} // RouteStop::RouteStop
```

Πρόσεξε τον πρώτο έλεγχο: Δεν επιτρέπεται να δηλώσεις στάση με απόσταση ή/και κόμιστρο χωρίς να δώσεις όνομα στάσης. Πάντως η κατάσταση:

$$sName.empty() \wedge sDist == 0 \wedge sFare == 0$$

είναι δεκτή! Είναι η κατάσταση που βρίσκεται το *oneStop* μετά τη δήλωση:

```
   RouteStop oneStop;
```

Εκείνο το «*Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία*» δεν θα το βάλουμε στην αναλλοίωτη; Πρόσεξε: αυτό έχει να κάνει όχι με μια στάση αλλά με ένα σύνολο στάσεων· άρα *αυτό έχει να κάνει με την κλάση διαδρομών και όχι με την κλάση των στάσεων*.

Στη συνέχεια θα εξοπλίσουμε την κλάση μας και με άλλες μεθόδους που θα χρειαζόμαστε για τη λύση του προβλήματός μας.

Η κλάση εξαιρέσεων για τη *RouteStop* είναι (προς το παρόν):

```
struct RouteStopXptn
{
   enum { noName, negDist, negFare };
   char         funcName[100];
   unsigned int errCode;
   float        errVal;
   RouteStopXptn( const char* fn, int ec, float ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal = ev;  }
}; // RouteStopXptn
```

Σημείωση:▶

Τώρα είδαμε πώς είναι ο δημιουργός αντιγραφής και ο τελεστής εκχώρησης της *BString* και γιατί μπορεί –κατ' αρχήν– να ρίξουν εξαίρεση *BStringXptn* με κωδικό *allocFailed*. Έτσι, καταλαβαίνεις αυτό που λέγαμε στην §Prj03.7: οι αντίστοιχες συναρτήσεις της *string* μπορεί –κατ' αρχήν– να ρίξουν *std*::*bad_alloc*.

Και ερχόμενοι στο προκείμενο, να καταλάβουμε ότι και οι αντίστοιχες συναρτήσεις της *RouteStop* –που έχει το "`string sName`"– μπορεί να ρίξουν (*std*::) *bad_alloc*. Δεν πρέπει να κάνουμε κάτι; Πρέπει (κατ' αρχήν)! Αλλά δεν θα κάνουμε! ◀

20.7.2 Η Κλάση για τις Διαδρομές

Ας πάμε τώρα να γράψουμε την κλάση *Route* με βάση την περιγραφή του προβλήματος:

```
class Route
{
public:
// . . .
private:
   unsigned int rCode;      // κωδικός διαδρομής
   string       rFrom;      // αφετηρία
   string       rTo;        // τέρμα
   string       rInBetween; // ενδιάμεσος
   RouteStop*   rAllStops;  // πίνακας στάσεων διαδρομής
   unsigned int rNoOfStops; // πλήθος στάσεων
}; // Route
```

Με το μέλος *rAllStops* θα χειριζόμαστε έναν δυναμικό πίνακα. Θα χρησιμοποιήσουμε την τεχνική που μάθαμε στη *BString*: Δηλώνουμε άλλο ένα μέλος,

```
   unsigned int rReserved;  // πλήθος στοιχείων που παραχωρήθηκαν
```

και πρέπει να αποφασίσουμε για το «κβάντο» αύξησης.

Μια διαδρομή έχει συνήθως μερικές δεκάδες στάσεις, κάτι σαν 10, 20, 30. Το 5 είναι μια λογική τιμή για το κβάντο:

```
   enum {  rIncr = 5  };   // το "κβάντο" αύξησης δυναμικής μνήμης
                           // για τον πίνακα rAllStops
```

20.7.2.1 Η Αναλλοίωτη

Ποια είναι η αναλλοίωτη της κλάσης μας; Εκτός από την $rCode \geq 0$ και τη συνθήκη διαχείρισης του δυναμικού πίνακα:

$$(0 \leq rNoOfStops < rReserved) \;\&\&\; (rReserved \;\%\; rIncr == 0)$$

το ουσιώδες μέρος είναι αυτό: «*Δεν υπάρχουν στάσεις με το ίδιο όνομα. Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία.*» Ακόμη, υπάρχει και κάτι άλλο που δεν γράφεται στην περιγραφή του προβλήματος ως αυτονόητο: *δεν υπάρχουν δύο στάσεις στην ίδια απόσταση από την αφετηρία*. Τα παραπάνω διατυπώνονται συμβολικώς ως εξής:

$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sName \neq \text{rAllStops}[k].sName)\;\&\&$$
$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)\;\&\&$$
$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[j].sDist > \text{rAllStops}[k].sDist) \Rightarrow$$
$$(\text{rAllStops}[j].sFare \geq \text{rAllStops}[k].sFare))$$

Εδώ πρέπει να λύσουμε ένα πρόβλημα: Αν πρέπει να προσθέσουμε μια νέα στάση πώς ελέγχουμε τη συμμόρφωσή της με την αναλλοίωτη; Θα πρέπει

- να βρούμε τη στάση με τη μέγιστη απόσταση από την αφετηρία που δεν είναι μεγαλύτερη από την απόσταση της νέας στάσης (αν υπάρχει),
- να βρούμε τη στάση με τη ελάχιστη απόσταση από την αφετηρία που δεν είναι μικρότερη από την απόσταση της νέας στάσης (αν υπάρχει),
- να ελέγξουμε αν η ισχύει η συνθήκη για τις τρεις αυτές στάσεις.

Αυτά μπορεί να γίνουν πολύ εύκολα αν έχουμε τον πίνακα ταξινομημένο ως προς την απόσταση από την αφετηρία:

$$\forall\, k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{-}1].sDist < \text{rAllStops}[k].sDist$$

Αν πρέπει να εισαγάγουμε την *aStop* και βρούμε ότι

$$\text{rAllStops}[k\text{-}1].sDist < aStop.sDist < \text{rAllStops}[k].sDist$$

η εισαγωγή πρέπει να γίνει στη θέση k και η αναλλοίωτη μας λέει ότι θα πρέπει να ισχύει και η:

$$\text{rAllStops}[k\text{-}1].sFare \leq aStop.sFare \leq \text{rAllStops}[k].sFare$$

Εδώ τώρα πρόσεξε ένα πρόβλημα:

- Αν το k έχει τιμή 0 (η εισαγωγή πρέπει να γίνει *πριν από το πρώτο* στοιχείο) πως θα ελέγξουμε την ανισότητα στα αριστερά;
- Αν το k έχει τιμή *rNoOfStops* (η εισαγωγή πρέπει να γίνει *μετά το τελευταίο* στοιχείο) πώς θα ελέγξουμε την ανισότητα στα δεξιά;

Μια λύση είναι να βάλουμε ξεχωριστούς ελέγχους για εισαγωγή στην πρώτη και στην τελευταία θέση. Μια άλλη λύση είναι οι φρουροί:

- Βάζουμε τα στοιχεία του πίνακα στις θέσεις $1\,..\,rNoOfStops$.
- Στη θέση 0 βάζουμε φρουρό

$$\text{rAllStops}[0].sDist == 0 \;\&\&\; \text{rAllStops}[0].sFare == 0$$

- Στη θέση *rNoOfStops*+1 βάζουμε φρουρό:

$$\text{rAllStops}[rNoOfStops\text{+}1].sDist == FLT\_MAX \;\&\&$$
$$\text{rAllStops}[rNoOfStops\text{+}1].sFare == FLT\_MAX$$

Αυτά μας αλλάζουν και τη σχέση μεταξύ *rNoOfStops* και *rReserved*:

$$(0 \leq rNoOfStops \leq rReserved\text{-}2) \;\&\&\; (rReserved \;\%\; rIncr == 0)$$

Τελικώς, μπορούμε να πούμε ότι η αναλλοίωτη της κλάσης είναι:

$$I \equiv I_D \;\&\&\; I_I$$

όπου (προσοχή στις αλλαγές των δεικτών!)

$$I_D \equiv (rCode \geq 0)\;\&\&$$
$$(\forall\, j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sName \neq \text{rAllStops}[k].sName)\;\&\&$$
$$(\forall\, j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)\;\&\&$$

$(\forall\ j, k: 1\,..\,rNoOfStops \bullet (\text{rAllStops}[j].sDist > \text{rAllStops}[k].sDist) \Rightarrow$
$(\text{rAllStops}[j].sFare \geq \text{rAllStops}[k].sFare))$

είναι το κομμάτι που εξαρτάται από τους **επιχειρησιακούς κανόνες** (business rules) που έχουν σχέση με το πρόβλημα:

- *Ο κωδικός είναι θετικός ακέραιος* (το "0" το προσθέτουμε εμείς και μόνο για δήλώσεις όπως: "`Route oneStop`").
- *Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία.*
- Οι στάσεις της διαδρομής είναι ένα *σύνολο*. Τα στοιχεία του συνόλου διαφέρουν ανά δύο ως προς το όνομα (*sName*) και ως προς την απόσταση από την αφετηρία (*sDist*).

$$I_I \equiv (0 \leq rNoOfStops \leq rReserved\text{-}2)\ \&\&\ (rReserved\ \%\ rIncr == 0)\ \&\&$$
$$(\forall\ k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{+}1].sDist > \text{rAllStops}[k].sDist)\ \&\&$$
$$(\text{rAllStops}[0].sDist == 0\ \&\&\ \text{rAllStops}[0].sFare == 0)\ \&\&$$
$$(\text{rAllStops}[rNoOfStops\text{+}1].sDist == FLT\_MAX\ \&\&$$
$$\text{rAllStops}[rNoOfStops\text{+}1].sFare == FLT\_MAX)$$

είναι το κομμάτι που εξαρτάται από τις επιλογές που κάναμε για την υλοποίηση. Αυτό το κομμάτι θα αλλάξει αν εγκαταλείψουμε τον δυναμικό πίνακα και επιλέξουμε άλλη δομή δεδομένων για την παράσταση των στάσεων. Σχετικώς να επισημάνουμε και το εξής: Αν ισχύει το τμήμα της I_I

$$(\forall\ k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{+}1].sDist > \text{rAllStops}[k].sDist)$$

προφανώς ισχύει και το τμήμα της I_D

$$(\forall\ j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)$$

αλλά αυτό δεν είναι λόγος να διαγράψουμε τη δεύτερη αφού αυτή ισχύει ακόμη και αν αλλάξουμε την υλοποίηση που επιλέγουμε για τις στάσεις.

20.7.2.2 Ένας Δημιουργός

Το πρώτο που θα κάνουμε είναι να γράψουμε έναν ερήμην δημιουργό. Τον δηλώνουμε:

```
   Route( int rc=0 );
```

και τον ορίζουμε ως εξής:

```
Route::Route( int rc )
{
   if ( rc < 0 )
      throw RouteXptn( "Route", RouteXptn::negCode, rc );
   rCode = rc;
   try
   {
      rReserved = rIncr;
      rAllStops = new RouteStop[ rReserved ];
      rNoOfStops = 0;
      rAllStops[0] = RouteStop( "guard0", 0.0, 0.0 );                  // φρουροί
      rAllStops[rNoOfStops+1] = RouteStop( "guardPte", FLT_MAX, FLT_MAX );
      rFrom = "";
      rTo = "";
      rInBetween = "";
   }
   catch( bad_alloc& )
   {
      throw RouteXptn( "Route", RouteXptn::allocFailed );
   }
} // Route::Route
```

Όπως βλέπεις, κατά βάση κάνουμε αυτά που ξέρουμε από τη *BString* και δεν αλλάζουν πολύ από το ότι εδώ έχουμε πίνακα αντικειμένων και όχι τιμών τύπου `char`. Η σημαντική διαφορά είναι οι δύο φρουροί.

20.7.2.3 Μέθοδοι

Θα χρειαστούμε μεθόδους για να δώσουμε τιμές στα μέλη *rFrom*, *rTo* και *rInBetween* (ή να αλλάξουμε τις τιμές τους). Είναι πολύ απλές και θα τις ορίσουμε **inline**:

```
   void setFrom( string aName )  {  rFrom = aName;  }
   void setTo( string aName )  {  rTo = aName;  }
   void setInBetween( string aName )  {  rInBetween = aName;  }
```

Πριν προχωρήσουμε στον χειρισμό των στοιχείων του πίνακα να πούμε ότι σε τέτοιες περιπτώσεις είναι χρήσιμη μια μέθοδος που «καθαρίζει» τον πίνακα ώστε το ίδιο αντικείμενο να ετοιμαστεί για να φιλοξενήσει άλλη διαδρομή:

```
void Route::clearRouteStops()
{  rAllStops[1] = rAllStops[rNoOfStops+1];
   rNoOfStops = 0;  }
```

Όπως βλέπεις δεν αναλυκλώνει τη μνήμη που έχει το αντικείμενο και επομενως δεν αλλάζει την τιμή του *rReserved*.

20.7.2.4 Χειρισμός Στοιχείων Πίνακα

Διαγραφή: Κατ' αρχάς ας ανταποκριθούμε στην απαίτηση «*να διαγράψουμε στάση που υπάρχει*», μια και η διαγραφή που είναι πιο απλή από την εισαγωγή στάσης.[6] Θα γράψουμε μια μέθοδο με όνομα *deleteRouteStop()* και προδιαγραφές:

```
// I
   aRoute.deleteRouteStop( stopName );
// I && (η διαδρομή aRoute δεν έχει στάση με όνομα stopName)
```

Αν δεν είχαμε την απαίτηση της ταξινόμησης τα πράγματα θα ήταν απλά:

```
void Route::deleteRouteStop( string stopName )
{
   if ( υπάρχει στάση με όνομα stopName στη θέση ndx )
   {
      αντίγραψε στη θέση ndx το τελευταίο στοιχείο του πίνακα
      --rNoOfStops;
   }
}
```

Τώρα, τα πράγματα είναι πιο πολύπλοκα:

```
   if ( υπάρχει στάση με όνομα stopName στη θέση ndx )
   {
      μετακίνησε τα στοιχεία
         rAllStops[ndx+1] ... rAllStops[rNoOfStops+1]
      στα
         rAllStops[ndx] ... rAllStops[rNoOfStops];
      --rNoOfStops;
   }
```

Αν γράψουμε μια

```
int Route::findNdx( const string& aName ) const
{
   rAllStops[rNoOfStops+1].setName( aName );
   int ndx( 1 );
   while ( rAllStops[ndx].getName() != aName )  ++ndx;
   if ( ndx > rNoOfStops ) ndx = -1;
   return ndx;
} // Route::findNdx
```

–που μας επιστρέφει τον δείκτη του στοιχείου που έχει στο *sName* τιμή *aName* ή –1 αν δεν βρει τέτοιο στοιχείο– μπορούμε να υλοποιήσουμε τη *deleteRouteStop()* ως εξής:

---

[6] Με όρους θεωρίας συνόλων –αν ονομάσουμε *A* το σύνολο στάσεων και *x* τη στάση που θέλουμε να διαγράψουμε– η πράξη που θέλουμε να υλοποιήσουμε είναι η: $A = A \setminus \{ x \}$ (το "=" με την έννοια της εκχώρησης).

```
void Route::deleteRouteStop( string stopName )
{
   int ndx( findNdx(stopName) );
   if ( ndx >= 0 )
   {
      try {  erase1RouteStop( ndx );  }
      catch( bad_alloc& )
      {  throw RouteXptn( "deleteRouteStop",
                          RouteXptn::allocFailed );  }
   }
} // Route::deleteRouteStop
```

όπου

```
void Route::erase1RouteStop( int ndx )
{
   for ( int k(ndx); k <= rNoOfStops; ++k )
      rAllStops[k] = rAllStops[k+1];
   --rNoOfStops;
} // Route::erase1RouteStop
```

Η δεύτερη συνάρτηση «σβήνει» το στοιχείο που βρίσκεται στη θέση *ndx* σε έναν ταξινομημένο πίνακα. Η πρώτη παίρνει την απόφαση για το αν θα πρέπει να σβηστεί κάποιο στοιχείο. Κάθε πρόγραμμα-πελάτης της κλάσης θα έχει πρόσβαση στην πρώτη συνάρτηση αλλά όχι στη δεύτερη που θα δηλωθεί σε περιοχή **private**.

Και εκείνα τα **try**/**catch** γιατί τα βάλαμε; Για όλα εκείνα τα «κατ' αρχήν» που λέγαμε πιο πάνω: κάποια από τις "`rAllStops[k] = rAllStops[k+1]`" της *erase1RouteStop()* μπορεί να ρίξει *bad_alloc*!

Η *deleteRouteStop()* είναι σύμφωνη με τις προδιαγραφές που βάλαμε αλλά μπορεί να σου φαίνεται περίεργο το ότι δεν αντιδρά στην περίπτωση που δεν υπάρχει στάση με το όνομα που μας ενδιαφέρει. Η αλήθεια είναι ότι σε μερικές περιπτώσεις το πρόγραμμα-πελάτης θα θέλει να το ξέρει. Για να λύσουμε αυτό το πρόβλημα θα εφοδιάσουμε την κλάση μας με την εξής μέθοδο:[7]

```
bool Route::find1RouteStop( string stopName )
{
   return ( findNdx(stopName) >= 0 );
} // Route::find1RouteStop
```

Και γιατί να μην χρησιμοποιήσουμε τη *findNdx()*; Η *findNdx()* βγάζει ως αποτέλεσμα τον δείκτη του στοιχείου στον πίνακα. Αυτός όμως έχει σχέση με την εσωτερική παράσταση του συνόλου των στάσεων. Αν τη δηλώσουμε "`public`" παραβιάζουμε την *απόκρυψη πληροφορίας* (§19.3.1). Αν αργότερα αλλάξουμε την εσωτερική παράσταση –π.χ. σε δένδρο– θα πρέπει να διορθώσουμε όλες τις εφαρμογές που χρησιμοποιούν την κλάση και τη *findNdx()*. Για τον λόγο αυτόν:

- Δηλώνουμε "`public`" τη *find1RouteStop()*.
- Δηλώνουμε "`private`" τη *findNdx()* που είναι *κρυμμένη μέθοδος* και όχι βοηθητική συνάρτηση (π.χ. σαν τη *lastDay()*).

Παρατηρήσεις: ▶

Πριν προχωρήσουμε θα κάνουμε δύο παρατηρήσεις σχετικώς με τη *findNdx()*.

1. Η *findNdx()* θέτει απαιτήσεις για την κλάση *RouteStop*: Πρέπει να έχει μεθόδους *setName()* και *getName()*.

2. Μιλούσαμε στην αρχή αυτού του κεφαλαίου για μηνύματα «*που μπορεί να δέχεται, ένα αντικείμενο ... είτε από άλλα αντικείμενα που "ζούν" στο [ίδιο] πρόγραμμα*». Εδώ δώσαμε τη δυνατότητα σε ένα αντικείμενο *aRoute* κλάσης *Route* να στέλνει

- Στο αντικείμενο –κλάσης *RouteStop*– *aRoute*`.rAllStops[`*rNoOfStops*+1`]` μήνυμα *setName()*.

[7] Με όρους θεωρίας συνόλων η *find1RouteStop()* υλοποιεί την $x \in A$.

- Σε ένα ή περισσότερα στοιχεία του πίνακα *aRoute*.rAllStops –που είναι αντικείμενα κλάσης *RouteStop*– μήνυμα *getName()*.◀

Εισαγωγή: Για να ανταποκριθούμε στην απαίτηση «*να προσθέσουμε νέα σταση*», θα γράψουμε μια μέθοδο με όνομα *addRouteStop()* και προδιαγραφές:

```
// I
   aRoute.addRouteStop( aStop );
// I && (η διαδρομή aRoute έχει τη στάση aStop)
```

Για να κάνουμε εισαγωγή της νέας στάσης θα πρέπει:[8]

- Να μην υπάρχει άλλη στάση με το ίδιο όνομα (*aStop*.*sName*) και φυσικά
- να μην υπάρχει άλλη σταση rAllStops[*ndx*] στην ίδια απόσταση από την αφετηρία (*aStop*.*sDist*). «Ίδια»; Τι θα πει ίδια; Προς το παρόν ας πούμε ότι σημαίνει «ίση» με την έννοια *aStop*.*sDist* == rAllStops[*ndx*].*sDist*.

Ας πούμε λοιπόν ότι δίνουμε:

```
   int nmNdx( findNdx(aStop.getName()) );
```

Τι θα κάνουμε αν πάρουμε *nmNdx* ≥ 1, αν δηλαδή βρούμε στάση με το ίδιο όνομα; Υπάρχουν δύο περιπτώσεις:

- Αν έχουμε επιπλέον *aStop*.*sDist* == rAllStops[*nmNdx*].*sDist* και *aStop*.*sFare* == rAllStops[*nmNdx*].*sFare* τότε «*η διαδρομή aRoute έχει τη στάση aStop*» χωρίς να κάνουμε οτιδήποτε! (θυμίσου: δεν κάναμε οτιδήποτε και στην περίπτωση της διαγραφής, όταν δεν βρήκαμε στάση με το ίδιο όνομα.)
- Αν όμως έχουμε διαφορές, δηλαδή *aStop*.*sDist* != rAllStops[*nmNdx*].*sDist* είτε *aStop*.*sFare* != rAllStops[*nmNdx*].*sFare* τότε έχουμε πρόβλημα και θα πρέπει να ρίξουμε εξαίρεση.

Δηλαδή:

```
   int nmNdx( findNdx(aStop.getName()) );
   if ( nmNdx > 0 )  // name found
   {
      if ( rAllStops[nmNdx].getDist() != aStop.getDist() ||
           rAllStops[nmNdx].getFare() != aStop.getFare() )
         throw RouteXptn( "addRouteStop",
                          RouteXptn::invalidArgument,
                          rAllStops[nmNdx], aStop );
   }
```

Ας πούμε τώρα ότι δεν έχουμε βρει στάση με το ίδιο όνομα· ψάχνουμε να βρούμε στάση στην ίδια απόσταση (*aStop*.*sDist*) από την αφετηρία. Αν υπάρχει θα έχει διαφορετικό όνομα· θα πρέπει λοιπόν να ρίξουμε εξαίρεση. Αφού έχουμε πίνακα ταξινομημένο στο *sDist* και φρουρούς στην αρχή και στο τέλος μπορούμε να χρησιμοποιήσουμε τη *binSearch()* της §9.6 αφού πρώτα

- τη μετατρέψουμε σε περίγραμμα

```
template < class T >
unsigned int binSearch( const T v[], int last, const T& x )
```

- επιφορτώσουμε τον τελεστή "<" για τη *RouteStop* ως εξής:

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < 0 );
} // operator<( const RouteStop
```

Αν δεν υπάρχει στάση στην ίδια απόσταση η *binSearch()* θα μας επιστρέψει τη θέση στην οποία θα πρέπει να εισαχθεί η νέα στάση ώστε ο πίνακας να είναι ταξινομημένος:

```
   else  // name not found
   {
```

[8] Με όρους θεωρίας συνόλων η *insert1RouteStop()* υλοποιεί την: $A = A \cup \{ x \}$.

```
      int distNdx( binSearch(rAllStops, rNoOfStops, aStop) );
      if ( rAllStops[distNdx].getDist() == aStop.getDist() )
         throw RouteXptn( "addRouteStop", RouteXptn::diffName,
                          rAllStops[distNdx], aStop );
```

αλλιώς θα πρέπει να έχουμε:

rAllStops[*distNdx*-1].*getDist*() < *aStop*.*getDist*() ≤ rAllStops[*distNdx*].*getDist*()

Πριν κάνουμε την εισαγωγή θα πρέπει να ελέγξουμε και τον περιορισμό για το κόμιστρο. Η αναλλοίωτη μας λέει ότι θα πρέπει να ισχύει και η:

rAllStops[*distNdx*-1].*getFare*()≤ *aStop*.*getFare*() ≤ rAllStops[*distNdx*].*getFare*()

```
      if ( rAllStops[distNdx-1].getFare() > aStop.getFare() )
         throw RouteXptn( "addRouteStop", RouteXptn::fareErr,
                          rAllStops[distNdx-1], aStop );
      if ( aStop.getFare() > rAllStops[distNdx].getFare() )
         throw RouteXptn( "addRouteStop", RouteXptn::fareErr,
                          aStop, rAllStops[distNdx] );
```

Αν περάσουμε και αυτούς τους ελέγχους είμαστε σχεδόν έτοιμοι για την εισαγωγή. Θα πρέπει μόνο να σιγουρέψουμε ότι έχουμε την απαραίτητη μνήμη:

```
      if ( rReserved <= rNoOfStops+2 )
      {
         try {  renew( rAllStops, rNoOfStops+2, rReserved+rIncr );
                rReserved += rIncr;  }
         catch( MyTmpltLibXptn& )
         {  throw RouteXptn( "addRouteStop",
                             RouteXptn::allocFailed );  }
      }
```

Πρόσεξε τη συνθήκη της **if**: Είναι η !*S* όπου

$$S \equiv rReserved > rNoOfStops+2$$

η συνθήκη: «*υπάρχουν διαθέσιμες θέσεις στον πίνακα*».

Πρόσεξε ακόμη ότι στην κλήση της *renew()* ζητούμε να αντιγραφούν στον νέο πίνακα τα πρώτα *rNoOfStops*+2 στοιχεία του παλιού.

Μετά από αυτό μπορούμε να κάνουμε την εισαγωγή:

```
      μετακίνησε τα στοιχεία
         rAllStops[distNdx] ... rAllStops[rNoOfStops+1]
      στα
         rAllStops[distNdx+1] ... rAllStops[rNoOfStops+2];
      rAllStops[distNdx] = aStop;
      ++rNoOfStops;
```

Οι μετακινήσεις στοιχείων «ανοίγουν κενό»[9] στη θέση *distNdx*. *Οι μετακινήσεις θα πρέπει να γίνουν από το τέλος προς την αρχή*:

```
      for ( int k(rNoOfStops+1); k >= distNdx; --k )
         rAllStops[k+1] = rAllStops[k];
      rAllStops[distNdx] = aStop;
      ++rNoOfStops;
```

Να ολόκληρη η μέθοδος, επιτέλους!

```
void Route::addRouteStop( const RouteStop& aStop )
{
   int nmNdx( findNdx(aStop.getName()) );
   if ( nmNdx > 0 )  // name found
   {
      if ( rAllStops[nmNdx].getDist() != aStop.getDist() ||
           rAllStops[nmNdx].getFare() != aStop.getFare() )
         throw RouteXptn( "addRouteStop",
                          RouteXptn::invalidArgument,
                          rAllStops[nmNdx], aStop );
   }
   else // name not found
```

[9] Φυσικά δεν υπάρχει «κενό»! Έτσι; Τρόπος του λέγειν...

```
   {
      int distNdx( binSearch(rAllStops, rNoOfStops, aStop) );
      if ( rAllStops[distNdx].getDist() == aStop.getDist() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::diffName,
                          rAllStops[distNdx], aStop );
      if ( rAllStops[distNdx-1].getFare() > aStop.getFare() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::fareErr,
                          rAllStops[distNdx-1], aStop );
      if ( aStop.getFare() > rAllStops[distNdx].getFare() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::fareErr,
                          aStop, rAllStops[distNdx] );
      insert1RouteStop( aStop, distNdx );
   }
} // Route::addRouteStop
```

όπου

```
void Route::insert1RouteStop( const RouteStop& aStop,
                              int distNdx )
{
   if ( rReserved <= rNoOfStops+2 )
   {
      try {  renew( rAllStops, rNoOfStops+2, rReserved+rIncr );
             rReserved += rIncr;  }
      catch( MyTmpltLibXptn& )
      {  throw RouteXptn( "insert1RouteStop", RouteXptn::allocFailed );  }
   }
   for ( int k(rNoOfStops+1); k >= distNdx; --k )
   {
      try {  rAllStops[k+1] = rAllStops[k];  }
      catch( bad_alloc& )
      {  throw RouteXptn( "insert1RouteStop", RouteXptn::allocFailed );  }
   }
   rAllStops[distNdx] = aStop;
   ++rNoOfStops;
} // Route::insert1RouteStop
```

Η *insert1RouteStop()*, όπως και η *erase1RouteStop()*, δηλώνεται ως **private**.

Έτσι, η εισαγωγή της στάσης *"MAPMAPA"* θα γίνει ως εξής:

```
      if ( oneRoute.find1RouteStop("MAPMAPA") )
         cout << "Στάση MAPMAPA υπάρχει" << endl;
      else
         oneRoute.addRouteStop( RouteStop("MAPMAPA", 25.5, 3.7) );
```

Τέλος, να σημειώσουμε ότι η υλοποίηση της *addRouteStop()* απαιτεί δύο ακόμη μεθόδους για τη *RouteStop*: τις *getDist()* και *getFare()*.

Τροποποίηση – Ανάκτηση: Στην περιγραφή του προβλήματος υπάρχει η απαίτηση για το πρόγραμμα να «*αλλάζει το όνομα της στάσης "ΣΤΑΥΡΟΣ" –αν υπάρχει– σε "ΔΙΑΣΤΑΥΡΩΣΗ"*.» Μήπως θα πρέπει να γράψουμε μια μέθοδο *modify1RouteStop()* για να αντιμετωπίζουμε τέτοιες απαιτήσεις; Μετά από αυτά που είδαμε για την *addRouteStop()*, ούτε να το σκεφθούμε. Τι κάνουμε λοιπόν;

```
   if ( υπάρχει στάση με όνομα "ΣΤΑΥΡΟΣ" )
   {
      αντίγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ" στο tmp
      βάλε στο tmp όνομα στάσης "ΔΙΑΣΤΑΥΡΩΣΗ"
      διάγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ"
      κάνε εισαγωγή του tmp
   }
```

Μπορούμε να υλοποιήσουμε όλα τα παραπάνω εκτός από την «*αντίγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ" στο tmp*». Για αυτήν την αντιγραφή θα μας χρειαστεί η

```
const RouteStop& Route::get1RouteStop( string stopName ) const
{
   int ndx( findNdx(stopName) );
   if ( ndx < 0 )
      throw RouteXptn( "get1RouteStop", RouteXptn::notFound,
```

```
                        stopName.c_str() );
   return rAllStops[ndx];
} // Route::get1RouteStop
```

Τώρα μπορούμε να γράψουμε:

```
      if ( !oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ") )
         cout << "Στάση ΣΤΑΥΡΟΣ δεν υπάρχει" << endl;
      else if ( oneRoute.find1RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ") )
         cout << "Στάση ΔΙΑΣΤΑΥΡΩΣΗ υπάρχει" << endl;
      else
      {
         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
```

Πρόσεξε δύο πράγματα:

- Αν βάζαμε τον τύπο της *get1RouteStop* "`RouteStop&`" και όχι "`const RouteStop&`" θα μπορούσαμε να τροποποιήσουμε το στοιχείο του πίνακα επι τόπου με τις συνέπειες που μπορείς να φανταστείς.
- Όταν γίνεται η κλήση `oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ")` έχουμε διασφαλίσει με τη `oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ")` ότι υπάρχει τέτοια στάση. Γενικώς, αν καλέσεις την *get1RouteStop* χωρίς να εξασφαλιστείς προηγουμένως με τη *find1RouteStop* μπορεί να ριχτεί εξαίρεση.

20.7.3 Οι Κλάσεις Τελικώς

Μετά από όσα είδαμε το περιεχόμενο του `Route.h` είναι:

```
#ifndef _ROUTE_H
#define _ROUTE_H

#include <string>
#include <new>

#include "RouteStop.h"

using namespace std;

class Route
{
public:
   Route( int rc=0 );
   ~Route() {  delete[] rAllStops;  }
   void setFrom( string aName )  {  rFrom = aName;  }
   void setTo( string aName )  {  rTo = aName;  }
   void setInBetween( string aName )  {  rInBetween = aName;  }
   bool find1RouteStop( string stopName ) const;
   const RouteStop& get1RouteStop( string stopName ) const;
   void addRouteStop( const RouteStop& aStop );
   void deleteRouteStop( string stopName );
   void writeToText( ostream& tout ) const;
private:
   enum {  rIncr = 5  };    // το "κβάντο" αύξησης δυναμικής
                            // μνήμης για τον πίνακα rAllStops
   unsigned int rCode;      // κωδικός διαδρομής
   string       rFrom;      // αφετηρία
   string       rTo;        // τέρμα
   string       rInBetween; // ενδιάμεσος
   RouteStop*   rAllStops;  // πίνακας στάσεων διαδρομής
   unsigned int rNoOfStops; // πλήθος στάσεων
   unsigned int rReserved;  // πλήθος στοιχείων που παραχωρήθηκαν
```

```
                              // πάντα πολλαπλάσιο του 5
   int findNdx( const string& aName ) const;
   void insert1RouteStop( const RouteStop& aStop, int distNdx );
   void erase1RouteStop( int ndx );
}; // Route

struct RouteXptn
{
   enum { negCode, allocFailed, invalidArgument, notFound,
          diffName, fareErr };
   char         funcName[100];
   unsigned int errCode;
   int          errIntVal;
   RouteStop      errStop1, errStop2;
   char         errStrVal[100];
   RouteXptn( const char* fn, int ec, int ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errIntVal = ev;  }
   RouteXptn( const char* fn, int ec, const char* astr )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, astr, 99 );  errStrVal[99] = '\0';  }
   RouteXptn( const char* fn, int ec,
              const RouteStop& aStop1, const RouteStop& aStop2 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errStop1 = aStop1;  errStop2 = aStop2;  }
}; // RouteXptn

#endif // _ROUTE_H
```

Η *writeToText()* ανταποκρίνεται στην τελευταία απαίτηση και είναι:

```
void Route::writeToText( ostream& tout ) const
{
   tout << rCode << endl;
   tout << rFrom << '\t' << rTo << '\t' << rInBetween << endl;
   for ( int k(1); k <= rNoOfStops; ++k )
   {  rAllStops[k].writeToText( tout );  tout << endl;  }
} // Route::writeToText
```

Όπως βλέπεις, θα πρέπει να γράψουμε μια *writeToText()* και για τη *RouteStop*.

Το περιεχόμενο του **RouteStop.h** είναι:

```
#ifndef _ROUTESTOP_H
#define _ROUTESTOP_H

#include <string>
#include <new>

using namespace std;

class RouteStop
{
public:
   RouteStop( string aName="", float aDist=0, float aFare=0 );
   string getName() const {  return sName;  }
   float getDist() const {  return sDist;  }
   float getFare() const {  return sFare;  }
   void setName( string aName );
   void writeToText( ostream& tout ) const;
private:
   string sName;  // όνομα στάσης
   float  sDist;  // απόσταση από αφετηρία σε km
   float  sFare;  // τιμή εισιτηρίου από την αφετηρία
}; // RouteStop

struct RouteStopXptn
{
```

```
   enum { noName, negDist, negFare };
   char         funcName[100];
   unsigned int errCode;
   double       errVal;
   RouteStopXptn( const char* fn, int ec, double ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal = ev;  }
}; // RouteStopXptn

bool operator<( const RouteStop& lhs, const RouteStop& rhs );
bool operator>=( const RouteStop& lhs, const RouteStop& rhs );

#endif //_ROUTESTOP_H
```

Η *writeToText()* είναι πολύ απλή:

```
void RouteStop::writeToText( ostream& tout ) const
{
   tout << sName << '\t' << sDist << '\t' << sFare;
} // RouteStop::writeToText
```

Πολύ απλός είναι και ο `operator<` (και ο "`>=`" που είναι ο "`!<`")

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < 0 );
} // operator<( const RouteStop

bool operator>=( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( !(lhs < rhs) );
} // operator>=( const RouteStop
```

20.7.4 Και το Πρόγραμμα

Το πρόγραμμα είναι απλούστατο:

```
#include <iostream>
#include <fstream>
#include <string>

#include "Route.cpp"

using namespace std;

void readFromText( RouteStop& aStop, istream& tin );

int main()
{
   try
   {
      ifstream tin( "kvadra.txt" );
      string   name, buf;
      double   dist, fare;

      getline( tin, buf, '\n' );

      Route oneRoute( atoi(buf.c_str()) );

      getline( tin, name, '\t' );  oneRoute.setFrom( name );
      getline( tin, name, '\t' );  oneRoute.setTo( name );
      getline( tin, name, '\n' );  oneRoute.setInBetween( name );

      RouteStop oneStop;
      readFromText( oneStop, tin );
      while ( !tin.eof() )
      {
         oneRoute.addRouteStop( oneStop );
         readFromText( oneStop, tin );
```

```
      } // while
      tin.close();

      if ( oneRoute.find1RouteStop("ΜΑΡΜΑΡΑ") )
         cout << "Στάση ΜΑΡΜΑΡΑ υπάρχει" << endl;
      else
         oneRoute.addRouteStop( RouteStop("ΜΑΡΜΑΡΑ", 25.5, 3.7) );

      if ( !oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ") )
         cout << "Στάση ΣΤΑΥΡΟΣ δεν υπάρχει" << endl;
      else if ( oneRoute.find1RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ") )
         cout << "Στάση ΔΙΑΣΤΑΥΡΩΣΗ υπάρχει" << endl;
      else
      {
         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
      ofstream tout( "kvadranew.txt" );
      oneRoute.writeToText( tout );
      tout.close();
   }
   catch( RouteStopXptn& x )
   {
      switch ( x.errCode )
      {
         case RouteStopXptn::noName:
            cout << "bus stop name cannot be empty in "
                 << x.funcName << endl;
            break;
         case RouteStopXptn::negDist:
            cout << "bus stop distance from start (" << x.errVal
                 << ") cannot be negative in " << x.funcName
                 << endl;
            break;
         case RouteStopXptn::negFare:
            cout << "fare from start to bus stop (" << x.errVal
                 << ") cannot be negative in " << x.funcName
                 << endl;
            break;
         default:
            cout << "unexpected RouteStopXptn exception from "
                 << x.funcName << endl;
      }
   }
   catch( RouteXptn& x )
   {
      switch ( x.errCode )
      {
         case RouteXptn::negCode:
            cout << "negative Route code (" << x.errIntVal
                 << ") in " << x.funcName << endl;
            break;
         case RouteXptn::allocFailed:
            cout << "out of memory in " << x.funcName << endl;
            break;
         case RouteXptn::invalidArgument:
         case RouteXptn::diffName:
            cout << "trying to insert bus stop (";
            x.errStop2.writeToText( cout );
            cout << ") over existing (";
            x.errStop1.writeToText( cout );
            cout << endl;
            break;
         case RouteXptn::notFound:
```

```
            cout << "no stop " << x.errStrVal << " requested by "
                 << x.funcName << endl;
            break;
         case RouteXptn::fareErr:
            cout << "inserting (";
            x.errStop2.writeToText( cout );
            cout << ") violates increasing fare principle (";
            x.errStop1.writeToText( cout );
            cout << endl;
            break;
         default:
            cout << "unexpected RouteXptn exception from "
                 << x.funcName << endl;
      }
   }
} // main
```

όπου:

```
void readFromText( RouteStop& aStop, istream& tin )
{
   string name;
   getline( tin, name, '\t' );
   if ( !tin.eof() )
   {
      string buf;
      getline( tin, buf, '\t' );
      float dist( atof(buf.c_str()) );
      getline( tin, buf, '\n' );
      float fare( atof(buf.c_str()) );
      aStop = RouteStop( name, dist, fare );
   }
} // readFromText
```

Σχόλια; Στη συνέχεια...

20.7.5 Σχόλια, Παρατηρήσεις κλπ

1. Κλάση Μέσα σε Κλάση

Η *RouteStop* θα μπορούσε να ορισθεί και μέσα στη *Route*:

```
class Route
{
public:
   class RouteStop
   {
   public:
   // . . .
   private:
   // . . .
   }; // RouteStop

   struct RouteStopXptn
   {
   // . . .
   }; // RouteStopXptn

   // . . .
private:
   // . . .
}; // Route

bool operator<( const Route::RouteStop& lhs,
                const Route::RouteStop& rhs );
bool operator>=( const Route::RouteStop& lhs,
                 const Route::RouteStop& rhs );
```

Στην περίπτωση αυτή έχουμε γενικώς την εξής αλλαγή:

όπου είχαμε "`RouteStop`" θα βάζουμε "`Route::RouteStop`" και
όπου είχαμε "`RouteStopXptn`" θα βάζουμε "`Route::RouteStopXptn`"

Αν σου φαίνονται περίεργα ξαναδιάβασε αυτά που λέγαμε για ονοματοχώρους στην §18.5.

Έτσι, για παράδειγμα, ο ορισμός του δημιουργού γίνεται:

```
Route::RouteStop::RouteStop( string aName,
                             float aDist, float aFare )
{
   if ( aName.empty() && (aDist > 0 || aFare > 0) )
      throw Route::RouteStopXptn( "RouteStop",
                                  Route::RouteStopXptn::noName );
   if ( aDist < 0 )
      throw Route::RouteStopXptn( "RouteStop",
                            Route::RouteStopXptn::negDist, aDist );
   if ( aFare < 0 )
      throw Route::RouteStopXptn( "RouteStop",
                            Route::RouteStopXptn::negFare, aFare );
   sName = aName;
   sDist = aDist;
   sFare = aFare;
} // Route::RouteStop::RouteStop
```

Πάντως, στις **throw** μπορούμε να απλουστεύσουμε το γράψιμο, π.χ.:

```
      throw RouteStopXptn( "RouteStop",
                           Route::RouteStopXptn::noName );
```

Στη `main` θα έχουμε:

```
     Route::RouteStop oneStop;
// . . .
        oneRoute.addRouteStop( Route::RouteStop("ΜΑΡΜΑΡΑ",
                                                25.5, 3.7) );
// . . .
        Route::RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
// . . .
        oneRoute.addRouteStop(
               Route::RouteStop( "ΔΙΑΣΤΑΥΡΩΣΗ",
                                 tmp.getDist(), tmp.getFare()) );
// . . .
```

Φυσικά, για να μπορούμε να δώσουμε αυτές τις εντολές θα πρέπει να έχουμε ορίσει τη *RouteStop* σε περιοχή **public**.

Πώς θα επιλέξουμε τη θέση ορισμού μιας τέτοιας κλάσης; Γενικώς: *Αν μια κλάση χρησιμοποιείται από μια μόνον άλλη κλάση αποκλειστικώς είναι προτιμότερο να ορίζεται μέσα στη δεύτερη.*

Στην περίπτωσή μας: η *RouteStop* έπρεπε να ορισθεί μέσα στη *Route* ή έξω από αυτήν; Όπως μπορείς να μαντέψεις, το πρόβλημα αυτού του παραδείγματος και οι κλάσεις που γράψαμε για τη λύση του είναι κομμάτι ενός πιο μεγάλου προβλήματος του οποίου η λύση απαιτεί και άλλες κλάσεις (και αρκετά προγράμματα). Για να απαντήσουμε το ερώτημα θα πρέπει να εφαρμόσουμε τον κανόνα που δώσαμε αλλά σε όλες τις κλάσεις.

2. Μέθοδος *readFromText*

Η *readFromText()* θα έπρεπε να είναι μέθοδος της *RouteStop* και όχι καθολική συνάρτηση. Γιατί τη γράψαμε έτσι; Διότι είναι προχειρογραμμένη σε απαράδεκτο βαθμό. Ως άσκηση (Άσκ. 20-5) σου αφήνουμε τη μετατροπή της *readFromText()* σε μέθοδο και το γράψιμο μιας *readFromText()* για τη *Route*. Για να τη λύσεις θα πρέπει να ξαναδείς τη λύση της Άσκ. 10-8.[10]

3. Χρησιμοποιώντας Εξαιρέσεις

Ένας άλλος τρόπος να γράψουμε το κομμάτι προγράμματος με την εισαγωγή και την αλλαγή ονόματος είναι ο εξής:

```
      try
```

[10] Την έλυσες;

```
      {
         oneRoute.addRouteStop( RouteStop("ΜΑΡΜΑΡΑ", 25.5, 3.7) );

         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
      catch( RouteXptn& x )
      {
         switch ( x.errCode )
         {
            case RouteXptn::invalidArgument:
               cout << "Στάση " << x.errStop2.getName()
                    << " υπάρχει" << endl;
               break;
            case RouteXptn::diffName:
               cout << "Υπάρχει στάση στα "
                    << x.errStop2.getDist() << " km με όνομα "
                    << x.errStop2.getName() << endl;
               break;
            case RouteXptn::notFound:
               cout << "Στάση " << x.errStrVal << " δεν υπάρχει"
                    << endl;
               break;
            default:
               throw;
         } // switch
      } // catch
```

Μια **try-catch** μέσα στην **try**! Ναι! Στην περιοχή του **try** έχουμε μόνο τις εντολές που μας ενδιαφέρουν χωρίς τους αμυντικούς ελέγχους με τις **if**. Στην **catch** ασχολούμαστε μόνο με *RouteXptn* και μάλιστα τρία συγκεκριμένα είδη: *invalidArgument*, *diffName* και *notFound*. Για τα υπόλοιπα είδη τι κάνουμε: ξαναρίχνουμε (στη **default**) την εξαίρεση για να την πιάσει η εξωτερική **catch**.

4. Συγκρίσεις Πραγματικών

Να επισημάνουμε τώρα και κάτι που έχει σχέση με προηγούμενα κεφάλαια. Ο τρόπος που επιφορτώσαμε τον τελεστή "`<`" για τη *RouteStop* ορίζει στην πραγματικότητα και την ισότητα για την κλάση αυτή:[11]

$$(a == b) \equiv ((a.sDist - b.sDist) == 0)$$

Δηλαδή δεχόμαστε ως κλειδί το μέλος *sDist*. Θα μπορούσαμε όμως να δεχτούμε και το *sName*. Επιλέξαμε το *sDist* λόγω της ταξινόμησης του πίνακα.

Τώρα πρόσεξε το εξής: Αν προσπαθήσουμε να εισαγάγουμε μια νέα στάση

`RouteStop( "ΣΚΑΡΗ", 25.6, 3.7 )`

είναι φανερό ότι αναφερόμαστε –με άλλο όνομα– στη στάση

`RouteStop( "ΜΑΡΜΑΡΑ", 25.5, 3.7 )`

Η διαφορά των 100 *m* (0.1 *km*) στην απόσταση από την αφετηρία είναι μέσα στα όρια σφάλματος για μια τέτοια μέτρηση. Έτσι, θα ήταν πιο σωστό αν επιφορτώναμε τον "`==`" ως

```
bool operator==( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( fabs(lhs.getDist()-rhs.getDist()) < 0.15 );
} // operator==( const RouteStop
```

(0.15 ή 0.1 ή 0.2, ό,τι διαλέξεις.)

Ο "`<`" –για να είναι συμβατός– θα πρέπει να επιφορτωθεί ως:

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < -0.15 );
```

[11] Πώς; Έτσι: (*a* `==` *b*) ≡ (`!`(*a* `<` *b*) `&&` `!`(*b* `<` *a*)).

```
} // operator<( const RouteStop
```

Ακόμη, η σύγκριση:

```
rAllStops[nmNdx].getDist() != aStop.getDist()
```

–στην *addRouteStop()*– θα πρέπει να γραφεί:

```
fabs(rAllStops[nmNdx].getDist()-aStop.getDist()) >= 0.15
```

Παρομοίως, η σύγκριση:

```
rAllStops[nmNdx].getFare() != aStop.getFare()
```

θα πρέπει να γραφεί:

```
fabs(rAllStops[nmNdx].getFare()-aStop.getFare()) >= 0.10
```

ή κάτι παρόμοιο.[12]

5. Μέθοδος *getAllStops()*

Ας πούμε ότι γράφεις τις *Route* και *RouteStop* για κάποιον πελάτη χωρίς να παραδώσεις την αρχική μορφή σε C++. Είναι σίγουρο ότι ο πελάτης σου θα σου ζητήσει τη δυνατότητα να παίρνει τον πίνακα των στάσεων ενός αντικειμένου *Route* για να τον επεξεργαστεί με τα δικά του προγράμματα. Θα χρειαστείς λοιπόν δύο μεθόδους *getAllStops()* και *getNoOfStops()*.

Η δεύτερη είναι απλή:

```
   unsigned int getNoOfStops() const {  return rNoOfStops;  }
```

Η πρώτη χρειάζεται λίγη προσοχή. Το απλούστερο που μπορούμε να κάνουμε είναι να μιμηθούμε τη *c_str()*:

```
const RouteStop* Route::getAllStops() const
{  return rAllStops+1;  }
```

Αυτή η μορφή έχει προβλήματα:

- Θα βάλεις ιδέες στον προγραμματιστή-χρήστη των κλάσεων να αρχίσει να σκαλίζει να ανακαλύψει τα μυστικά μας με τους φρουρούς και –πιθανότατα– να καταστρέψει τα αντικείμενα.
- Αργότερα, που θα αλλάξουμε τη δομή αποθήκευσης σε `set<RouteStop>` τι θα κάνεις;[13]

Μια άλλη ιδέα είναι η εξής: Γράφουμε μια μέθοδο

```
const RouteStop* Route::getAllStops() const
{
   RouteStop* fv( 0 );
   try
   {  fv = new RouteStop[rNoOfStops];  }
   catch( bad_alloc& )
   {  throw RouteXptn( "getAllStops", RouteXptn::allocFailed );  }
   for ( int k(0); k < rNoOfStops; ++k )  fv[k] = rAllStops[k+1];
   return fv;
} // Route::getAllStops
```

που επιστρέφει *αντίγραφο* του πίνακα στάσεων. Αλλά το αντίγραφο είναι δυναμικός πίνακας για τον οποίον δεν φαίνεται η εντολή που παίρνει τη μνήμη και το πιθανότερο είναι ότι ο προγραμματιστής που θα χρησιμοποιήσει τη μεθοδο θα ξεχάσει να βάλει την αντίστοιχη "`delete[]…`".

Μπορούμε να κάνουμε κάτι καλύτερο; Αν σκεφτούμε ότι οι πεπειραμένοι προγραμματιστές μόλις γράψουν μια "`new`" γράφουν και την αντίστοιχη "`delete`" γράφουμε μια μέθοδο

```
void Route::getAllStops( RouteStop*& aStops ) const
{
   try {  renew( aStops, 0, rNoOfStops );  }
   catch( MyTmpltLibXptn& )
   {  throw RouteXptn( "getAllStops", RouteXptn::allocFailed );  }
   for ( int k(0); k < rNoOfStops; ++k )
```

---

[12] Αυτά τα "`0.15`" και "`0.10`" δεν είναι «μαγικές σταθερές»; Είναι και παραείναι! Θα ασχοληθούμε με αυτές αργότερα.

[13] `set<RouteStop>`!!! Τι είναι αυτά; Θα τα μάθεις αργότερα…

```
      aStops[k] = rAllStops[k+1];
} // Route::getAllStops
```

Τώρα, ο πεπειραμένος προγραμματιστής που λέγαμε, θέλοντας να χρησιμοποιήσει τη *getAllStops()* θα γράψει αρχικώς:

```
   RouteStop* allStops( new RouteStop[1] );

   delete[] allStops;
```

και στη συνέχεια θα συμπληρώσει:

```
   RouteStop* allStops( new RouteStop[1] );
// . . .
   oneRoute.getAllStops( allStops );
// . . .
   delete[] allStops;
```

Πάντως θα πρέπει να σημειώσουμε ότι αυτή η «καλύτερη λύση» κοστίζει σε υπολογιστικό χρόνο· αυτά έχουν οι αντιγραφές πινάκων.

6. Γιατί όχι *setCode()*;

Γιατί –ενώ γράψαμε *setFrom()*, *setTo()* και *setInBetween()*– δεν γράψαμε και μια *setCode()* για τη *Route*; Διότι το *rCode* είναι (υποκατάστατο) κλειδί και από τις ΒΔ ξέρουμε ότι δεν αλλάζουμε το κλειδί.[14]

Βέβαια, μπορεί το *rCode* να είναι υποκατάστατο κλειδί αλλά το { *rFrom*, *rTo*, *rInBetween* } είναι το «φυσικό» κλειδί. Γιατί γράψαμε τις *setFrom()*, *setTo()* και *setInBetween()*;

Μπορούμε να μην τις γράψουμε και απλώς να αλλάξουμε τον δημιουργό:

```
Route( int rc=0,
       string aFrom="", string aTo="", string aInBetween="" );
```

Έτσι, στο πρόγραμμά μας θα έχουμε:

```
// . . .
      string   nameF, nameT, nameIB, buf;
      double   dist, fare;

      getline( tin, buf, '\n' );
      getline( tin, nameF, '\t' );
      getline( tin, nameT, '\t' );
      getline( tin, nameIB, '\n' );

      Route oneRoute( atoi(buf.c_str()), nameF, nameT, nameIB );
// . . .
```

Τώρα έγινε χειρότερο! Ας πούμε ότι έβαλα στο *rTo* τιμή `"Δράννα"` αντί για `"Δράμα"`. Δεν μπορώ να το διορθώσω; Στο επόμενο κεφάλαιο θα επιφορτώσουμε τον "`operator=`" για τη *Route*. Θα μπορείς να το διορθώσεις ως εξής:

```
oneRoute = Route( oneRoute.getCode(), oneRoute.getFrom(),
                  "Δράμα", oneRoute.getInBetween() );
```

Φυσικά, κάτι θα πρέπει να κάνεις και για τις στάσεις.

Πάντως, οι *setFrom()*, *setTo()* και *setInBetween()* «νομιμοποιούνται» να υπάρχουν διότι ένας από τους λόγους που εισάγουμε υποκατάστατα κλειδιά είναι ακριβώς αυτός: αν το κλειδί αποτελείται από πολλές (στην περίπτωσή μας: τρεις) λέξεις θα γίνονται λάθη και θα πρέπει να μπορούμε να τα διορθώνουμε.

20.8 Συσχετίσεις Κλάσεων

Η κλάση *Route* –μέρος της λύσης ένός πιο «πραγματικού» προβλήματος– είναι σαφώς πιο πολύπλοκη από τις *Date* και *BString*. Κάθε αντικείμενο κλάσης *Route* περιέχει –δύο ή περισ-

[14] Και για όποιον δεν το κατάλαβε ακόμη: αυτό το παράδειγμα το έχουμε πάρει από μια βάση δεδομένων.

σότερα– αντικείμενα κλάσης *RouteStop*. Έχουμε λοιπόν, στην περίπτωση αυτή, μια **συσχέτιση** (relationship) δύο κλάσεων. Πιο συγκεκριμένα λέμε ότι σε ένα αντικείμενο *Route* έχουμε **σύνθεση** (composition) αντικειμένων *RouteStop*. Κάθε αντικείμενο κλάσης *RouteStop* ανήκει σε ένα αντικείμενο κλάσης *Route*. Όταν καταστρέφεται το αντικείμενο *Route* κατάστρέφονται και τα αντικείμενα κλάσης *RouteStop* που περιέχει.

Η **πολλαπλότητα** (multiplicity) της συσχέτισης είναι **ένα-προς-πολλά** (one-to-many) και τη γράφουμε συμβολικώς 1:*N* ή 1:*. Με μεγαλύτερη ακρίβεια μπορούμε να πούμε:

- Η πολλαπλότητα ως προς τη *RouteStop* ("*N*"): Ένα αντικείμενο κλάσης *Route* έχει κατ' ελάχιστο 2 αντικείμενα κλάσης *RouteStop* (αφετηρία και τέρμα) ενώ δεν υπάρχει περιορισμός στο μέγιστο. Γράφουμε: `2..*`.
- Η πολλαπλότητα ως προς τη *Route* ("1"): Ένα αντικείμενο κλάσης *RouteStop* ανήκει σε ένα ακριβώς αντικείμενο κλάσης *Route* (σε μια διαδρομή και μόνο), με την έννοια που είπαμε πιο πάνω. Γράφουμε: `1..1`.

Για να δουμε και άλλα είδη συσχετίσεων δίνουμε μερικές ακόμη πτυχές του προβλήματος διαδρομών λεωφορείων:

> *Ένα δρομολόγιο γίνεται σε μια διαδρομή, σε συγκεκριμένη ημερομηνία και με συγκεκριμένο χρόνο αναχώρησης (π.χ. 21.06.2015 18:00), από ένα μόνο λεωφορείο και με συγκεκριμένο οδηγό. Σε μερικά δρομολόγια μπορεί να υπάρχει και εισπράκτορας. Τα λεωφορεία έχουν ως ταυτότητα τον αριθμό κυκλοφορίας, ενώ ένα χαρακτηριστικό τους, που μας ενδιαφέρει ιδιαιτέρως, είναι η χωρητικότητά τους (αριθμός θέσεων).*

Μεγάλο μέρος της λύσης του προβλήματος διαχείρισης των δρομολογίων είναι η σχεδίαση και η υλοποίηση μιας Βάσης Δεδομένων (ΒΔ), όπου θα βρίσκονται όλα τα δεδομένα που μας ενδιαφέρουν. Εδώ θα ασχοληθούμε με τα προγράμματα για τη διαχείριση των στοιχείων που βρίσκονται στη ΒΔ και πιο συγκεκριμένα με τις κλάσεις που θα έχουμε σε αυτά τα προγράμματα. Στα προγράμματα αυτά υπάρχουν *συλλογές* με αντικείμενα των κλάσεων οι οποίες επικοινωνούν με τη ΒΔ. Όταν εκτελείται κάποιο από αυτά τα προγράμματα φορτώνει από τη ΒΔ στη μνήμη –στις συλλογές– τα δεδομένα που χρειάζεται. Στο τέλος της εκτέλεσης κάποιου προγράμματος, οι οποιεσδήποτε ενημερώσεις των περιεχομένων των συλλογών έχουν μεταφέρθεί και στη ΒΔ.

Παρατηρήσεις: ▶

1. Κατά τη διάρκεια της εκτέλεσης έχουμε φορτωμένο στη μνήμη ένα τμήμα της ΒΔ. Πόσο μεγάλο τμήμα; Από μερικά αντικείμενα μέχρι ολόκληρη τη ΒΔ.
2. Παραπάνω λέμε ότι τα αντικείμενα υπάρχουν σε *συλλογές*. Τι είδους συλλογές; πίνακες ή δένδρα ή λίστες κλπ.
3. Η απάντηση στα παραπάνω ερωτήματα καθορίζεται από την πολιτική διαχείρισης της ΒΔ από το πρόγραμμα. Η επιλογή πολιτικής δεν γίνεται στην τύχη. Χρειάζεται «μέτρημα»!

◀

Από την περιγραφή που μας δίνεται μπορούμε να δούμε τις εξής κλάσεις:

- Για τις στάσεις:

```
class RouteStop
```

- Για τις διαδρομές:

```
class Route
```

- Για τους εργαζόμενους:

```
class Employee
{
public:
// . . .
private:
   unsigned int eIdNum;      // αριθμός μητρώου εργαζομένου
   string       eSurname;    // επώνυμο
```

```
   string       eFirstName;  // όνομα
   char         eOccupation; // εργασία
}; // Employee
```

- Για τα λεωφορεία:

```
class Bus
{
public:
// . . .
private:
   string        bRegNum;      // αριθμός κυκλοφορίας
   unsigned int  bNoOfSeats;   // αριθμός θέσεων
   unsigned int* bAllServices; // πίνακας δρομολογίων
}; // Bus
```

- Για τα δρομολόγια:

```
class BusService
{
public:
// . . .
private:
   unsigned int bseCode;      // κωδικός δρομολογίου
   Route        bseRoute;     // διαδρομή
   DateTime     bseDT;        // ημερομηνία/ώρα αναχώρησης
   Bus          bseBus;       // λεωφορείο
   Employee     bseDriver;    // οδηγός
   Employee     bseConductor; // εισπράκτορας
}; // BusService
```

Αυτά για μια πρώτη καταγραφή. Τώρα ας τα σκεφτούμε λίγο παραπάνω.

- Ας πάρουμε ως διαδρομή το παράδειγμα που δίνεται στην περιγραφή του προβλήματος: Αθήνα – Σούνιο από τα Μεσόγεια. Πόσα τέτοια δρομολόγια γίνονται κάθε μέρα; Πολλά! Μπορεί και ένα κάθε δυο ώρες. Προφανώς, το να έχουμε σε κάθε ένα από αυτά τα δρομολόγια όλα τα στοιχεία και κυρίως όλες τις στάσεις της διαδρομής είναι σπατάλη μνήμης. Εκτός από αυτό όμως, εδώ έχουμε μια περίπτωση **πλεονασμού** (redundancy) που δημιουργεί τις λεγόμενες **ανωμαλίες ενημέρωσης** (update anomalies). Θα πρέπει να διορθώνεις *όλα τα δρομολόγια* που έχουν τη συγκεκριμένη διαδρομή κάθε φορά που θέλεις:
  - Να *τροποποιήσεις* κάποιο στοιχείο της διαδρομής· αυτή είναι η **ανωμαλία τροποποίησης** (modification anomaly).
  - Να *διαγράψεις* μια στάση· αυτή είναι η **ανωμαλία διαγραφής** (deletion anomaly).
  - Να *εισαγάγεις* μια (νέα) στάση· αυτή είναι η **ανωμαλία εισαγωγής** (insertion anomaly).
- Παρόμοια προβλήματα υπάρχουν με τα στοιχεία του λεωφορείου, του οδηγού και του εισπράκτορα.

Πώς αντιμετωπίζονται αυτά τα προβλήματα; Με το να γράψουμε την κλάση για τα δρομολόγια ως εξής:

```
class BusService
{
public:
// . . .
private:
   unsigned int bseRouteCode;      // κωδικός διαδρομής
   DateTime     bseDT;             // ημερομηνία/ώρα αναχώρησης
   unsigned int bseBusRegNum;      // αρ. κυκλοφορίας λεωφορείου
   unsigned int bseDriverIdNum;    // αρ. μητρ. οδηγού
   unsigned int bseConductorIdNum; // αρ. μητρ. εισπράκτορα
}; // BusService
```

Δηλαδή αντί να βάλουμε μέσα σε ένα αντικείμενο κλάσης *BusService* ολόκληρα τα αντικείμενα των άλλων κλάσεων βάλαμε μόνον τα κλειδιά των αντικειμένων.[15]

Γιατί δεν κάναμε το ίδιο και για τις στάσεις; Εκεί τα πράγματα είναι διαφορετικά: Φυσικά μια στάση μπορεί να ανήκει σε πολλές διαδρομές αλλά δεν είναι απαραίτητο να έχει σε όλες τις διαδρομές την ίδια απόσταση από την αφετηρία –που μπορεί να μην είναι ίδια για όλες τις διαδρομές– και επομένως τό ίδιο κόμιστρο.

Και τώρα να δούμε και λίγη ορολογία για όσα είδαμε:

- Η σχέση της *BusService* με τις *Route*, *Bus* και *Employee* είναι σχέση **συνάθροισης** (aggregation). Κάθε αντικείμενο κλάσης *Route*, *Bus* και *Employee* μπορεί να ανήκει σε *ένα ή περισσότερα* αντικείμενα κλάσης *BusService*. Όταν καταστρέφεται το αντικείμενο *BusService δεν* καταστρέφονται τα αντικείμενα *Route*, *Bus* και *Employee* που συναθροίζει.
  - *BusService–Route*. Πολλαπλότητα ως προς τη *BusService*: Σε μια διαδρομή εκτελούνται από κανένα (ανενεργή διαδρομή) μέχρι πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς τη *Route*: Ένα δρομολόγιο εκτελείται σε μια ακριβώς διαδρομή: `1..1`. (Συσχέτιση **πολλά-προς-ένα**, many-to-one.)
  - *BusService–Bus*. Πολλαπλότητα ως προς τη *BusService*: Ένα λεωφορείο εκτελεί από κανένα μέχρι πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς τη *Bus*: Ένα δρομολόγιο εκτελείται «*από ένα μόνο λεωφορείο*»: `1..1`. (Συσχέτιση πολλά-προς-ένα.)
  - *BusService–Employee*. Οι κλάσεις αυτές συσχετίζονται με δύο τροπους που ξεχωρίζουν από τον **ρόλο** (role) του εργαζόμενου. Πολλαπλότητα –και στις δύο περιπτώσεις– ως προς τη *BusService*: Ένας εργαζόμενος εργάζεται σε κανένα, σε ένα ή σε πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς την *Employee*: α) αν ο ρόλος του εργαζόμενου είναι οδηγός (συσχέτιση με την τιμή του μέλους *bseDriverIdNum*): ένα δρομολόγιο εκτελείται από έναν οδηγό: `1..1` β) αν ο ρόλος είναι εισπράκτορας (συσχέτιση με την τιμή του μέλους *bseConductorIdNum*, μπορεί να μην υπάρχει): `0..1`. (Συσχετίσεις πολλά-προς-ένα.)

Η *σύνθεση* και η *συνάθροιση* είναι δύο περιπτώσεις της σχέσης "`has_a`". Στη βιβλιογραφία μπορεί να συναντήσεις και την αντίστροφη σχέση της "`has_a`", την "`is_part_of`". Γράφουμε δηλαδή:

Route `has_a` *RouteStop* ή *RouteStop* `is_part_of` *Route*
BusService `has_a` *Bus* ή *Bus* `is_part_of` *BusService*

Οι κλάσεις που είδαμε παραπάνω είναι όλες κλάσεις στη **φάση της υλοποίησης** (implementation phase). Η σχεδίαση των κλάσεων όμως ξεκινάει πολύ νωρίς στη φάση της **σχεδίασης** (design) και περνάει από –συνήθως αρκετές– αναθεωρήσεις μέχρι να φτάσει σε τελική μορφή.

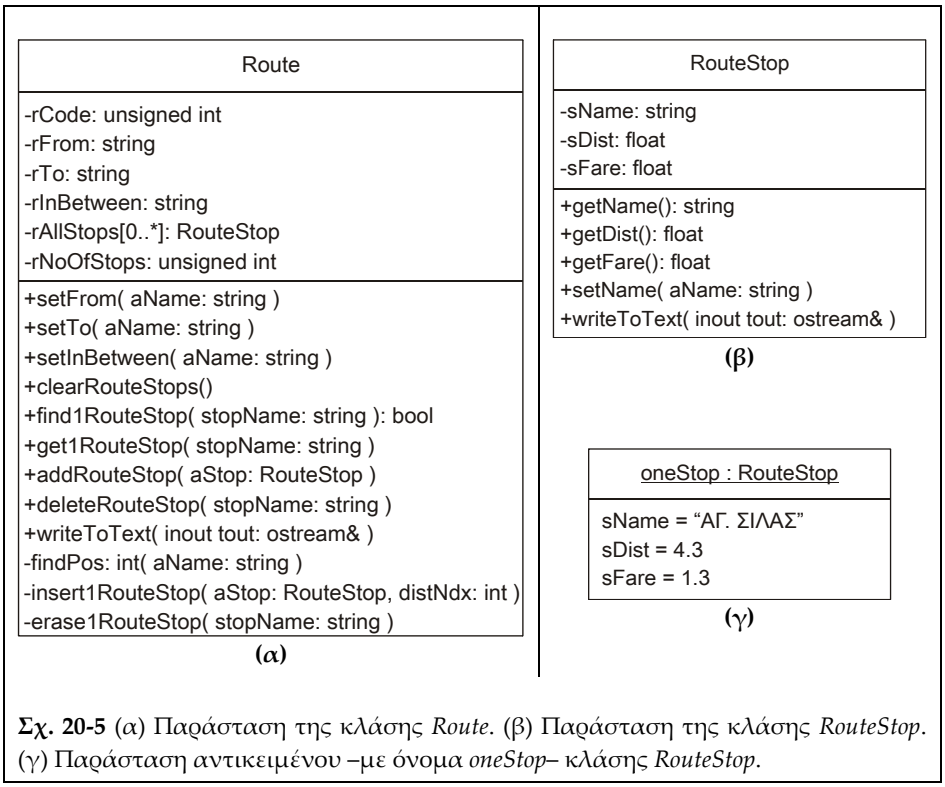
Οι συσχετίσεις που είδαμε παραπάνω είναι διμελείς και απλές. Στη φάση της σχεδίασης οι συσχετίσεις μπορεί να είναι πολυμελείς και πιο πολύπλοκες. Μια τέτοια συσχέτιση μπορεί να συνοδεύεται και από αντίστοιχη **κλάση συσχέτισης** (association class).

Η κλάση *BusService* είναι στην πραγματικότητα η κλάση μιας *τετραμελούς συσχέτισης* των κλάσεων *Route*, *Bus*, *Employee* και *Employee*. Έτσι, έχει ως μέλη τα χαρακτηριστικά (κλειδιά) των κλάσεων που συσχετίζει: *bseRouteCode*, *bseBusRegNum*, *bseDriverIdNum* και *bseConductorIdNum*. Έχει όμως και ένα επιπλέον χαρακτηριστικό το *bseDT*.

Για να δούμε άλλη μια κλάση συσχέτισης ξεκινούμε με άλλη μια πτυχή του προβλήματός μας:

> *Για κάθε στάση κάποιας από τις διαδρομές που εξυπηρετούν τα λεωφορεία του ΚΤΕΛ καταγράφουμε: το όνομα (που είναι μοναδικό), περιγραφή που καθορίζει*

[15] Λέμε ότι αυτά τα χαρακτηριστικά της *BusService* είναι **ξένα κλειδιά** (foreign keys).



Σχ. 20-5 (α) Παράσταση της κλάσης *Route*. (β) Παράσταση της κλάσης *RouteStop*. (γ) Παράσταση αντικειμένου –με όνομα *oneStop*– κλάσης *RouteStop*.

με ακρίβεια τη θέση και το επίπεδο υποδομών για την εξυπηρέτηση των επιβατών (σταθμός, οικίσκος, στέγαστρο, σήμα στάσης).

Κάθε στάση παριστάνεται με ένα αντικείμενο κλάσης

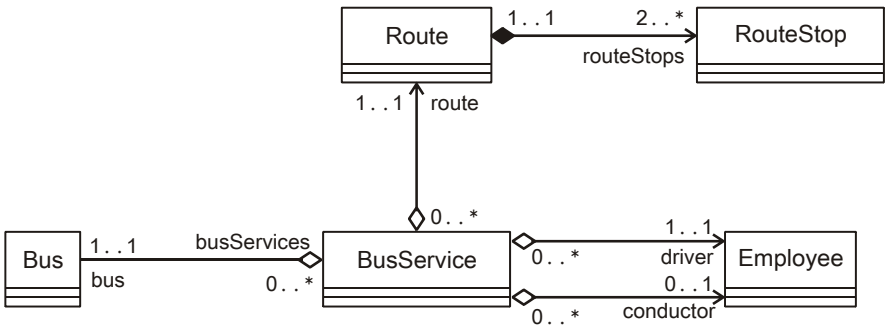
```
class BusStop
{
public:
// . . .
private:
   string        bsName;      // όνομα στάσης
   string        bsLocation;  // θέση
   unsigned char bsFaciLevel; // επίπεδο εξυπηρέτησης
}; // BusStop
```

Η *BusStop* συσχετίζεται με τη *Route*. Κάθε διαδρομή (αντικείμενο κλάσης *Route*) έχει 2 ή περισσότερες στάσεις (αντικείμενα κλάσης *BusStop*)· πολλαπλότητα 2..*. Κάθε στάση περιλαμβάνεται σε μια τουλάχιστον διαδρομή: πολλαπλότητα 1..*. Η πολλαπλότητα της συσχέτισης είναι πολλα-προς-πολλά (*M*:*N*). Για κάθε στάση που συσχετίζουμε με μια διαδρομή υπολογίζουμε την απόστασή της και το κόμιστρο από την αφετηρία. Δηλαδή η συσχέτιση έχει δύο χαρακτηριστικά. Έχουμε λοιπόν μια κλάση συσχέτισης, τη *RouteStop*, που έχει ως μέλη τα κλειδιά των κλάσεων που συσχετίζονται (*rCode*, *bsName*) και επιπλέον τα *sDist* και *sFare*. Στη φάση της υλοποίησης, κάνοντας την επιλογή να βάλουμε μέσα στο αντικείμενο *Route* όλα τα αντικείμενα *RouteStop* που το αφορούν, ο κωδικός διαδρομής είναι άχρηστος.

Αν έχεις ασχοληθεί με τη σχεδίαση Βάσεων Δεδομένων τα παραπάνω θα σου είναι οικεία. Εκεί, στα μοντέλα Οντοτήτων-Συσχετίσεων, έχεις *οντότητες* (entities)· οι κλάσεις είναι ένας τρόπος υλοποίησης των οντοτήτων.

20.8.1 Διαγραμματικές Παραστάσεις

Στην Τεχνολογία Λογισμικού χρησιμοποιούνται πολλά είδη διαγραμματικών παραστασεων που βοηθούν τον μηχανικό λογισμικού στις διάφορες φάσεις της ανάλυσης, της σχεδία-



Σχ. 20-6 Διαγραμματική παράσταση των κλάσεων και των συσχετίσεών τους.

σης, της ανάπτυξης κλπ. Εδώ θα πάρουμε μια ιδέα ενός είδους διαγραμμάτων. Δεν είναι κατ' ανάγκην το πιο χρήσιμο· το επιλέγουμε επειδή παριστάνει με διαγράμματα αυτά που είπαμε παραπάνω. Ακολουθούμε (όχι με «ευλάβεια») τούς κανόνες της UML (OMG 2010).[16]

Να ξεκινήσουμε με την παράσταση κλάσεων. Αυτή γίνεται με ένα ορθογώνιο με τρία τμήματα (Σχ. 20-5): Στο πρώτο γράφουμε το όνομα της κλάσης, στο δεύτερο τα μέλη και στο τρίτο τις μεθόδους. Σε ορισμένες περιπτώσεις μπορεί να υπάρχει και τέταρτο τμήμα με τις εξαιρέσεις που ρίχνουν οι μέθοδοι της κλάσης.

Κάθε μέλος γράφεται σε ξεχωριστή σειρά με το εξής συντακτικό:

ορατότητα, *όνομα*, "**:**", *τύπος*

όπου:

ορατότητα = "**+**" | "**-**" | "**#**";

Το "**+**" σημαίνει ότι δηλώνεται σε περιοχή "**public**", το "**-**" σε περιοχή "**private**" και το "**#**" σε περιοχή "**protected**" (αυτή θα τη μάθουμε αργότερα).

Για τις μεθόδους γράφουμε:

ορατότητα, *όνομα*, "**(**", [*λίστα παραμέτρων*]. "**)**", "**:**", *τύπος*

όπου:

λίστα παραμέτρων = *παράμετρος* | *λίστα παραμέτρων*, *παράμετρος*;

παράμετρος = [*είδος*], *όνομα*, [*πολλαπλότητα*]. "**:**", *τύπος*;

είδος = "**in**" | "**out**" | "**inout**";

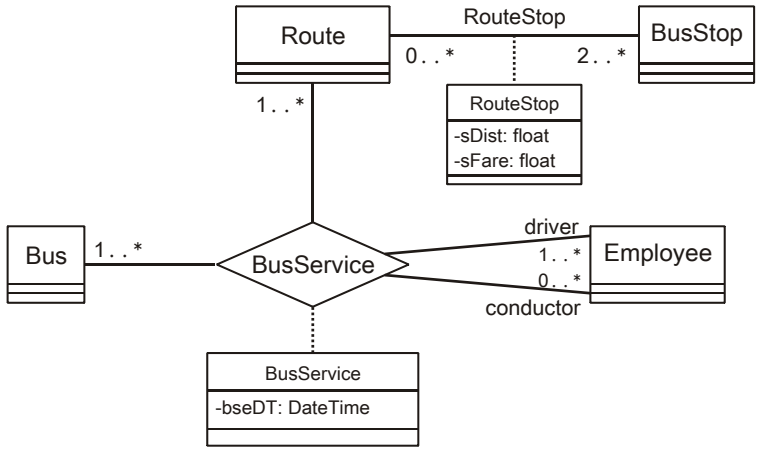
Αν δεν υπάρχει το *είδος* εννοείται "**in**".

Στα διαγράμματα δεν είναι απαραίτητο να βάζουμε όλα τα στοιχεία μιας κλάσης· βάζουμε αυτά που απαιτούνται για να περιγράψουμε ό,τι θέλουμε. Μπορείς να παραλείψεις και ολόκληρο τμήμα. Στο Σχ. 20-6 βλέπεις ότι από τις κλάσεις που παριστάνονται φαίνεται μόνον το τμήμα του ονόματος ενώ στο Σχ. 20-7 βλέπεις τις κλάσεις συσχέτισης με τα τμήματα ονόματος και μελών.

Να σχολιάσουμε τώρα αυτά που βλέπουμε στο Σχ. 20-6:

- Όπως είπαμε, στη συσχέτιση *Route-RouteStop* έχουμε *σύνθεση* αντικειμένων κλάσης *RouteStop* σε ένα αντικείμενο κλάσης *Route*. Αυτό παριστάνεται με τον μαύρο ρόμβο στο άκρο προς τη *Route*. Στο ίδιο άκρο βλέπουμε την πληθικότητα της κλάσης στη συσχέτιση και τον ρόλο της. Στο άλλο άκρο βλέπουμε ένα βέλος προς τη *RouteStop*. Αυτό δείχνει μια **μονόδρομη** (uni-directional) συσχέτιση: Στην τιμή ενός αντικειμένου *Route* υπάρχει η πληροφορία για τα αντικείμενα *RouteStop* με τα οποία συσχετίζεται (πίνακας rAllStops) ενώ στην τιμή ενός αντικειμένου *RouteStop* δεν υπάρχει πληροφορία για σχετιζόμενο αντικείμενο *Route*. Για τον λόγο αυτόν δεν υπάρχει και ρόλος για το αντικείμενο *RouteStop*.

[16] UML: Unified Modeling Language, OMG: Object Management Group.



Σχ. 20-7 Το διάγραμμα οντοτήτων-συσχετίσεων από το οποίο προήλθε το διάγραμμα του Σχ. 20-6. Εδώ η *RouteStop* είναι κλάση (διμελούς) συσχέτισης. Κλάση (τετραμελούς) συσχέτισης είναι και η *BusService*. Οι πληθικότητες διαφέρουν από αυτές του Σχ. 20-6.

- Στη συσχέτιση *BusService–Bus* έχουμε –στο άκρο προς τη *BusService*– έναν λευκό ρόμβο. Αυτός δείχνει ότι ένα αντικείμενο κλάσης *Bus συναθροίζεται* σε ένα αντικείμενο κλάσης *BusService*. Στη γραμμή συσχέτισης δεν υπάρχουν βέλη· αυτό δείχνει μια **αμφίδρομη** (bi-directional) συσχέτιση. Πράγματι, από ένα αντικείμενο κλάσης *BusService* μπορούμε να βρούμε το λεωφορείο που την εξυπηρετεί και από ένα αντικείμενο *Bus* μπορούμε να βρούμε τα δρομολόγια που εξυπηρετεί (πίνακας bAllServices). Αυτό φαίνεται και από τους ρόλους που βλέπουμε και στα δύο άκρα.
- Στις παραπάνω συσχετίσεις οι αναγραφές των ρόλων μπορεί να μην είναι απαραίτητες. Αυτό όμως δεν ισχύει για τις δύο συσχετίσεις *BusService–Employee*. Χωρίς τις αναγραφές των ρόλων δεν είναι φανερές οι αντιστοιχίες των (διαφορετικών) πληθικοτήτων.

Στο Σχ. 20-7 βλέπεις το διάγραμμα κλάσεων από το οποίο προήλθε αυτό του Σχ. 20-6. Πρόκειται για ένα **διάγραμμα οντοτήτων-συσχετίσεων** (entity-relationship –ER– diagram). Εδώ βλέπεις:

- Μια επιπλέον κλάση, τη *BusStop*, που δεν υπάρχει στο Σχ. 20-6.
- Η *RouteStop* είναι μια κλάση συσχέτισης που «κρέμεται» από τη γραμμή της (διμελούς) συσχέτισης με το ίδιο όνομα. Παρομοίως, η *BusService* είναι η κλάση της τετραμελούς συσχέτισης με το ίδιο όνομα. Ο ρόμβος χρησιμοποιείται για την παράσταση συσχετίσεων με τρία ή περισσότερα μέλη.[17]
- Οι πληθικότητες έχουν αλλάξει. Πώς υπολογίζονται αυτές της *RouteStop*; Πληθικότητα της *Route* είναι η απάντηση στην ερώτηση: «*Πόσες στάσεις έχει μια διαδρομή;*» ενώ πληθικότητα της *BusStop* είναι η απάντηση στην ερώτηση: «*Σε πόσες διαδρομές συμμετέχει μια στάση;*» Για τις πληθικότητες της *BusService* οι ερωτήσεις είναι πιο πολύπλοκες. Για παράδειγμα για την πληθικότητα της *Route* ρωτάμε: «*Ένα λεωφορείο, με έναν όδηγό και (πιθανόν) με έναν εισπράκτορα πόσες διαδρομές εξυπηρετεί;*» Με παρόμοιες ερωτήσεις βρίσκουμε τις πληθικότητες των άλλων κλάσεων *Bus*, *Employee* (*driver*) και *Employee* (*conductor*)

Το **διάγραμμα κλάσεων** (class diagram) είναι ένα είδος στατικού **διαγραμμάτος δομής** (structure diagram) από τα πολλά που υπάρχουν. Πέρα από αυτά υπάρχουν και τα

[17] Για τις συσχετίσεις θα δεις να χρησιμοποιούνται συνήθως *ρήματα* και όχι ουσιαστικά.

διαγράμματα συμπεριφοράς (behavior diagrams) που περιγράφουν τα μηνύματα που μπορεί να δεχθεί κάποιο αντικείμενο και την ανταπόκρισή του σε αυτά.

Για να καταλάβεις τη χρησιμότητα αυτών των διαγραμμάτων θα πρέπει να πάρεις υπόψη σου ότι πρώτα θα γίνει το διάγραμμα οντοτήτων-συσχετίσεων, από αυτό θα προέλθει –με χρήση κάποιων κανόνων– το διάγραμμα κλάσεων του Σχ. 20-6 και από αυτό (και από άλλα) θα σχεδιαστούν και θα αναπτυχθούν οι κλάσεις. Έτσι, τα διαγράμματα μπορεί να χρησιμοποιηθούν για τον έλεγχο ορθότητας της σχεδίασης: κάποιοι, που είναι ειδικοί στο πρόβλημα που λύνει το πληροφοριακό σύστημα που σχεδιάζεις, θα καταλάβουν πολύ πιο εύκολα ένα τέτοιο διάγραμμα παρά το πρόγραμμα που γράφεις και έτσι θα μπορείς να έχεις τις διορθώσεις τους και τις παρατηρήσεις τους. Στη συνέχεια, είναι δική σου ευθύνη να απεικονίσεις με σωστό τρόπο αυτά που δείχνει το διάγραμμα στο πρόγραμμα που γράφεις.

20.9 Για να Γράψουμε μια Κλάση...

Όταν σχεδιάζουμε μια κλάση το εύκολο μέρος είναι να βρούμε τα χαρακτηριστικά των αντικειμένων της. Το πολύ-πολύ να κάνουμε μερικές υπερβολές και να βάλουμε ως χαρακτηριστικά κάποιες υπολογιζόμενες ποσότητες. Για παράδειγμα στη *Date* αν βάζαμε ως μέλος:

```
WeekDay dDayOfWeek;
```

θα ήταν λάθος αφού αυτή μπορεί να υπολογιστεί.

Πάντως τέτοια λάθη εύκολα εντοπίζονται και διορθώνονται.

Η σχεδίαση συμπεριφοράς των αντικειμένων, δηλαδή το πώς θα ανταποκρίνονται στα διάφορα μηνύματα, θέλει περισσότερη προσοχή. Στις περισσότερες περιπτώσεις, όπως είναι φυσικό, τα πιο σημαντικά στοιχεία της συμπεριφοράς έχουν να κάνουν με τη συγκεκριμένη κλάση και τις εφαρμογές που αυτή χρησιμοποιείται. Υπάρχουν όμως και μερικά στοιχεία τα οποία, αν δεν είναι υποχρεωτικό να υπάρχουν, θα πρέπει να εξετασθεί η ανάγκη ύπαρξής τους.

Αυτά θα τα τυποποιήσουμε αργότερα σε μια «συνταγή». Προς το παρόν όμως ας σκεφθούμε μερικά από αυτά.

Ο **ερήμην δημιουργός**, με ή χωρίς αρχικές τιμές, είναι απαραίτητος για τη δήλωση μεταβλητών-αντικειμένων της κλάσης (και όχι μόνον). Τον γράφουμε έτσι ώστε αρχικώς το αντικείμενο να συμμορφώνεται με την αναλλοίωτη της κλάσης.

Αν τα αντικείμενα της κλάσης χρειάζονται δυναμική μνήμη τότε πρέπει να ορίσουμε εμείς **δημιουργό αντιγραφής**, **τελεστή εκχώρησης** και **καταστροφέα**.[18]

Τέλος, χρειαζόμαστε μεθόδους για να παίρνουμε (*get*) και να αλλάζουμε (*set*) τις τιμές των μελών ενός αντικειμένου. Όλων; Όχι κατ' ανάγκη. Για παράδειγμα, στη *BString*, έχει νόημα να γράψουμε μια *getReserved()*; Όχι βέβαια! Η ύπαρξη του μέλους *bsReserved*

- *έχει σχέση* με τον τρόπο που αποφασίσαμε εμείς να χειριστούμε την παραχώρηση δυναμικής μνήμης στα αντικείμενά μας και
- *δεν έχει σχέση* με το «φυσικό νόημα» των αντικειμένων κλάσης *BString*.

Πολύ περισσότερο, δεν έχει νόημα να γράψουμε μια *setReserved*(. . .).

Σε πολλές περιπτώσεις οι απαιτήσεις της εφαρμογής μας οδηγούν να γράψουμε μεθόδους που είναι πολύ προτιμότερες από τις "*get*" και "*set*". Αν γυρίσουμε στη *Battery*, του προηγούμενου κεφαλαίου, οι *powerDevice()* και *maxTime()* είναι πολύ πιο ουσιαστικές από μια *getEnergy()* και η *recharge()* πιο ουσιαστική από μια *setEnergy()*. Η *Τεχνολογία Λογισμικού* –και πιο συγκεκριμένα η *Αντικειμενοστρεφής Σχεδίαση*– μας δίνει μια τεχνική, τις **περι-**

[18] Αυτά είναι τα μέλη του «κανόνα των τριών» που θα δούμε στη συνέχεια.

πτώσεις χρήσης (use cases), που –μεταξύ άλλων– μας βοηθάει να βρούμε τέτοιες «καλές» μεθόδους.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

20-1 Στέλνουμε μήνυμα σε ένα αντικείμενο:

- Γράφοντας το όνομα της κλάσης και το όνομα της αντίστοιχης μεθόδου με τα κατάλληλα ορίσματα
- Γράφοντας το όνομα του αντικειμένου και το όνομα της αντίστοιχης μεθόδου με τα κατάλληλα ορίσματα
- Με eMail

20-2 Είναι απαραίτητο να ορίσουμε τελεστή εκχώρησης για μια κλάση αν

- Αν κάθε αντικείμενό της έχει έναν τουλάχιστον δυναμικό πίνακα
- Αν κάθε αντικείμενό της έχει έναν πίνακα τουλάχιστον
- Αν κάθε αντικείμενό της έχει περισσότερα από ένα μέλη

20-3 Χρειάζεται να ορίσουμε εμείς τελεστή εκχώρησης για την κλάση *RouteStop*;

- Ναι (δικαιολόγησε την απάντησή σου και όρισέ τον)
- Όχι (δικαιολόγησε την απάντησή σου)

Το ίδιο για την κλάση:

```
class RouteStopCStr
{
public:
// . . .
private:
   char  sName[20]; // όνομα στάσης
   float sDist;     // απόσταση από αφετηρία σε km
   float sFare;     // τιμή εισιτηρίου από την αφετηρία
}; // RouteStopCStr
```

Β Ομάδα

20-4 Συμπληρώνοντας αυτά που λέμε στην πρώτη παρατήρηση της §20.7.5, ξαναγράψε ολόκληρο το πρόγραμμα των διαδρομών λεωφορείου ορίζοντας τη *RouteStop* μέσα στη *Route*.

20-5 Μετάτρεψε τη *readFromText()* σε μέθοδο της *RouteStop*. Όπως έχουμε τονίσει, ένα αρχείο text μπορεί να έχει πολλά λάθη (αφού ο χρήστης μπορεί να προσπαθήσει να το αλλάξει με έναν απλό κειμενογράφο). Βάλε άμυνες για όσο περισσότερες «κακοτοπιές» μπορείς. Πριν ξεκινήσεις, ξαναδές τη λύση της Άσκ. 10-8.

Το ίδιο προσεκτικά γράψε μια *readFromText* για τη *Route*.

Γ Ομάδα

20-6 Μια βιομηχανία κατασκευάζει έπιπλα. Για λόγους προγραμματισμού της παραγωγής και προμήθειας πρώτων υλών η βιομηχανία κρατάει τη σύνθεση του κάθε προϊόντος της σε ένα αντικείμενο κλάσης:

```
class Product
```

```
{
public:
// . . .
private:
   string       prCode;      // Κωδικός προϊόντος
   string       prDescr;     // Περιγραφή
   Component*   prComp;      // Συνιστώσες
   unsigned int prNoOfComp;  // Πλήθος συνιστωσών
}; // Person
```

Ο πίνακας των συνιστωσών (πρώτων υλών) έχει στοιχεία τύπου:

```
class Component
{
public:
// . . .
private:
   char   cCode[10];  // Κωδικός πρώτης ύλης
   double cQty;       // ποσότητα
}; // Product
```

1. Υλοποίησε την κλάση με τις μεθόδους που θεωρείς απαραίτητες.

2. Ας πούμε ότι θέλουμε να αλλάξουμε την ποσότητα κάποιας πρώτης ύλης (ξέρουμε τον κωδικό της) που υπάρχει στη σύνθεση κάποιου προϊόντος. Γράψε μέθοδο που θα κάνει αυτήν τη δουλειά.

3. Πώς θα μπορούσαμε να γράψουμε ένα αντικείμενο κλάσης *Product* σε ένα μη μορφοποιημένο αρχείο ώστε να έχουμε τη δυνατότητα να το ξαναδιαβάσουμε στη συνέχεια. Γράψε μεθόδους *save()* και *load()* που λύνουν το πρόβλημα αυτό.

Αν θέλεις, μπορείς να εισαγάγεις και άλλα μέλη στην κλάση· δεν μπορείς όμως να αφαιρέσεις ή να αλλάξεις αυτά που υπάρχουν.

κεφάλαιο

21

Ειδικές Συναρτήσεις και Άλλα

Ο στόχος μας σε αυτό το κεφάλαιο:
Να γνωρίσουμε καλύτερα μερικά βασικά εργαλεία που υπάρχουν σε κάθε κλάση (είτε το θέλουμε είτε όχι). Τελικώς θα καταλήξουμε σε μια «συνταγή» που θα πρέπει να ακολουθούμε όταν σχεδιάζουμε οποιαδήποτε κλάση: Τι πρέπει να σκεφθούμε και να αποφασίσουμε ασχέτως από τις εφαρμογές όπου θα χρησιμοποιηθεί η κλάση μας.

Προσδοκώμενα αποτελέσματα:
Όταν θα έχεις μελετήσει αυτό το τεύχος θα μπορείς να σχεδιάσεις και να υλοποιήσεις, κατ' αρχήν, οποιαδήποτε κλάση.

Έννοιες κλειδιά:

- *δημιουργός*
- *ερήμην δημιουργός*
- *δημιουργός αντιγραφής*
- *λίστα εκκίνησης*
- *καταστροφέας*
- *τελεστής εκχώρησης*
- *RAII*
- *στατικά μέλη*
- *σταθερά μέλη*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Όπως λέγαμε στο εισαγωγικό σημείωμα του προηγούμενου κεφαλαίου, στον ορισμό μιας κλάσης «*Περιγράφεται επίσης και η λειτουργία* ***δημιουργίας στιγμιοτύπου*** *(instance creation).*» Στην πραγματικότητα η λειτουργία δημιουργίας στιγμιοτύπου είναι το εργαλείο που μας χρειάζεται για να διασφαλίσουμε ότι η αρχική κατάσταση του αντικειμένου συμμορφώνεται με την αναλλοίωτη. Στη C++ μας δίνεται η δυνατότητα να καθορίσουμε εμείς το πώς μπορεί να γίνονται οι δηλώσεις των αντικειμένων μιας κλάσης, π.χ. μεταβλητών τύπου *Date*. Αυτό γίνεται με τον ορισμό ειδικών μεθόδων που λέγονται **δημιουργοί** (creators) ή **κατασκευαστές** (constructors).

Μέχρι τώρα είδαμε δημιουργούς για τις κλάσεις *Date*, *BString*, *Battery*, *Route* και *RouteStop*. Στη *BString* και στη *Route* παρουσιάστηκε και άλλη ανάγκη: κάθε αντικείμενό τους δεσμεύει πόρους του συστήματος (μνήμη) που πρέπει να αποδεσμευθεί όταν το αντικείμενο καταστρέφεται. Έτσι, υποχρεωθήκαμε να ορίσουμε και **καταστροφέα** (destructor) όπου περιγράφουμε πώς αποδεσμεύονται οι πόροι (μνήμη) που δεσμεύθηκαν.

Ενώ όμως για κάθε κλάση έχουμε έναν καταστροφέα, συχνά χρειαζόμαστε περισσότερους από έναν δημιουργούς για διαφορετικές χρήσεις. Στη συνέχεια θα δούμε τις διάφορες περιπτώσεις όπου καλούνται οι δημιουργοί.

Τέσσερα από αυτά τα εργαλεία ονομάζονται **ειδικές συναρτήσεις** (special functions): ο *ερήμην δημιουργός*, ο *δημιουργός αντιγραφής*, ο *καταστροφέας* και ο *τελεστής εκχώρησης* και μοιράζονται μια άλλη πολύ ενδιαφέρουσα ιδιότητα: αν δεν τα ορίσουμε εμείς ορίζει αυτόματως (και ερήμην μας) ο μεταγλωττιστής **συναγόμενες** (implicit) αντίστοιχες συναρτήσεις. Είναι σωστές; Πολύ συχνά όχι. Είδαμε ήδη παραδείγματα για την κλάση *BString*.

Πολλοί περιλαμβάνουν στις ειδικές συναρτήσεις όλους τους δημιουργούς καθώς και τις **συναρτήσεις μετατροπής τύπου** (conversion functions) που θα δούμε επίσης.

Δύο προβλήματα που αντιμετωπίζουμε συχνά γράφοντας δημιουργούς −και άλλες μεθόδους− είναι:

- Η ανάγκη για σταθερές με όνομα, για να αποφύγουμε τις λεγόμενες «μαγικές σταθερές».
- Η ανάγκη για βοηθητικές συναρτήσεις.

  Εδώ θα τα δούμε πληρέστερα.

Τέλος, θα δούμε τις περιπλοκές από τη χρήση εξαιρέσεων στους δημιουργούς και τους καταστροφείς και που χρειάζονται ιδιαίτερη προσοχή.

Στη τέλος του κεφαλαίου θα βρεις τη «συνταγή»-ανακεφαλαίωση.

21.1 Ερήμην Δημιουργός

Ο **ερήμην δημιουργός** (default constructor) είναι αυτός που δημιουργεί ένα αντικείμενο και καθορίζει την κατάστασή του (τιμές των μελών) αυτομάτως, ερήμην του προγραμματιστή, δηλαδή *ο δημιουργός που μπορεί να κληθεί χωρίς παραμέτρους*. Η πιο απλή περίπτωση τέτοιου δημιουργού είναι αυτός που δεν έχει παραμέτρους.

Για να τον δούμε καλύτερα, αλλάζουμε λιγάκι τη *Date*:

```
class Date
{ // I: (dYear > 0) && (0 < dMonth <= 12) &&
  //    (0 < dDay <= lastDay(dYear, dMonth))
public:
//   Date( int yp = 1, int mp = 1, int dp = 1 );
   Date();
   Date( int yp, int mp = 1, int dp = 1 );
// . . .
```

δηλαδή, απομονώνουμε την περίπτωση:

```
Date::Date()
{
   cout << "in Date default constructor" << endl;
   dYear = 1;  dMonth = 1;  dDay = 1;
                                // 1η Ιανουαρίου του έτους 1 μ.Χ.
} // Date::Date
```

Οι εντολές:

```
   cout << "απλή δήλωση" << endl;
   Date d;
   cout << "δήλωση πίνακα" << endl;
   Date da[3];
   cout << "δυναμική μεταβλητή" << endl;
   Date* pd( new Date );
   cout << "δυναμικός πίνακας" << endl;
   Date* pda( new Date[2] );
```

θα μας δώσουν:

```
απλή δήλωση
in Date default constructor
δήλωση πίνακα
in Date default constructor
in Date default constructor
in Date default constructor
δυναμική μεταβλητή
in Date default constructor
δυναμικός πίνακας
in Date default constructor
in Date default constructor
```

Έτσι, εμπειρικώς, βλέπουμε ότι ο ερήμην δημιουργός καλείται όταν:

- Δηλώνουμε μια μεταβλητή χωρίς αρχική τιμή.
- Δηλώνουμε σταθερό πίνακα (μια φορά για κάθε στοιχείο).
- Παίρνουμε δυναμική μνήμη για μια μεταβλητή χωρίς αρχική τιμή.
- Παίρνουμε δυναμική μνήμη για πίνακα (μια φορά για κάθε στοιχείο).

Καταλαβαίνεις ότι σπανίως θα γράψεις κλάση χωρίς ερήμην δημιουργό.

Παρατηρήσεις: ▶

1. Γιατί στον δημιουργό με αρχική τιμή γράψαμε:

```
   Date( int yp, int mp = 1, int dp = 1 );
```

και όχι

```
   Date( int yp = 1, int mp = 1, int dp = 1 );
```

Διότι ο μεταγλωττιστής δεν θα ήξερε ποιον από τους δύο δημιουργούς να διαλέξει σε όλες τις δηλώσεις του παραδείγματος, αφού και οι δύο θα ήταν κατάλληλοι.

2. Δες έναν τρόπο αντιμετώπισης των «μαγικών σταθερών»:

```
   dYear = 1;  dMonth = 1;  dDay = 1;        // 1η Ιανουαρίου του έτους 1 μ.Χ.
```

Καλά, με σχόλιο; Ε, με σχόλιο, τι να κάνουμε; . . .◀

21.1.1 Δημιουργός με Αρχική Τιμή

Όπως είπαμε παραπάνω «*Ο ερήμην δημιουργός (default constructor) είναι ... ο δημιουργός που μπορεί να κληθεί χωρίς παραμέτρους.*» Τέτοιος δημιουργός είναι και αυτός που ήδη ονομάσαμε «2 σε 1» (ή «πολλοί σε 1»): *αυτός που έχει παραμέτρους αλλά όλες έχουν ερήμην καθοριζόμενες τιμές.* Αυτός είναι συνήθως πιο πολύπλοκος από τον ερήμην δημιουργό χωρίς παραμέτρους αφού θα πρέπει να κάνουμε ελέγχους συμμόρφωσης με την αναλλοίωτη. Και τι γίνεται αν βρούμε παρανομίες; Ρίχνουμε εξαιρέσεις!

Για παράδειγμα στη *Date,* δηλώνουμε (§19.1):

```
   Date( int yp = 1, int mp = 1, int dp = 1 );
```

ορίζουμε:

```
Date::Date( int yp, int mp, int dp )
{
cout << "in Date default constructor: ";
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   dYear = yp;
   if ( mp <= 0 || 12 < mp )
      throw DateXptn( "Date", DateXptn::monthErr, mp );
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
cout << dDay << '.' << dMonth << '.' << dYear << endl;
} // Date
```

και οι εντολές:

```
   cout << "απλή δήλωση" << endl;
   Date d;
   cout << "δήλωση πίνακα" << endl;
   Date da[3];
   cout << "δυναμική μεταβλητή" << endl;
   Date* pd( new Date );
   cout << "δυναμικός πίνακας" << endl;
   Date* pda( new Date[2] );
```

θα μα ς δώσουν:

```
απλή δήλωση
in Date default constructor: 1.1.1
δήλωση πίνακα
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
δυναμική μεταβλητή
in Date default constructor: 1.1.1
δυναμικός πίνακας
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
```

Αυτός ο δημιουργός μπορεί να λειτουργεί ως ερήμην δημιουργός αλλά και ως **δημιουργός με αρχική τιμή** (initializing constructor). Δοκιμάζουμε τις εντολές:

```
   cout << "δήλωση με αρχική τιμή" << endl;
   Date d( 2006, 8, 26 );
   cout << "δήμιουργία σταθεράς" << endl;
   Date( 2007, 10, 5 );
   cout << "δυναμική μεταβλητή με αρχική τιμή" << endl;
   Date* pd( new Date(2001, 1, 11) );
   cout << "μετατροπή τύπου ορίσματος" << endl;
   bool before( d < 2008 );
   cout << before << endl;
```

Αποτέλεσμα:

```
δήλωση με αρχική τιμή
in Date default constructor: 26.8.2006
```

```
δήμιουργία σταθεράς
in Date default constructor: 5.10.2007
δυναμική μεταβλητή με αρχική τιμή
in Date default constructor: 11.1.2001
μετατροπή τύπου ορίσματος
in Date default constructor: 1.1.2008
1
```

Βλέπουμε λοιπόν ότι ο δημιουργός με αρχική τιμή καλείται όταν:

- Δηλώνουμε μια μεταβλητή *με αρχική τιμή*.
- Δημιουργούμε μια «σταθερά της κλάσης».
- Παίρνουμε δυναμική μνήμη για μια μεταβλητή *με αρχική τιμή*.
- Περνούμε σε παράμετρο μια τιμή (2008) που πρέπει να *μετατραπεί* σε τιμή της κλάσης μας (*Date*).

Ερήμην δημιουργό με αρχική τιμή έχουμε και στην κλάση *BString*, όπου δηλώνουμε:

```
   BString( const char* rhs="" );
```

και ορίζουμε:

```
BString::BString( const char* rhs )
{
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   try {  bsData = new char [bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BString
```

Υπάρχει περίπτωση να έχουμε δημιουργό που χρειάζεται οπωσδήποτε κάποια αρχική τιμή στη δήλωση; Δεν είναι τόσο συχνή περίπτωση αλλά υπάρχει πιθανότητα να χρειαστείς κάτι τέτοιο. Ο δημιουργός της *Date* μπορεί να γίνει έτσι αν τον δηλώσεις:

```
   Date( int yp, int mp = 1, int dp = 1 );
```

και ο δημιουργός της *BString* αν τον δηλώσεις:

```
   BString( const char* rhs );
```

21.2 Δημιουργός Αντιγραφής

Μια ειδική κατηγορία δημιουργών είναι οι **δημιουργοί αντιγραφής** (copy constructors). Αυτοί καλούνται όταν κατά τη δήλωση ενός αντικειμένου του δίνουμε ως αρχική τιμή την τιμή ενός άλλου της ίδιας κλάσης, π.χ.:

```
   Date d1( 2002, 7, 2 ), d2( d1 );  // d1 == d2 == 02.07.2002
```

ή

```
   Date d1( 2002, 7, 2 ), d2 = d1;   // d1 == d2 == 02.07.2002
```

Στην περίπτωση αυτή καλείται ο ερήμην καθορισμένος δημιουργός αντιγραφής της κλάσης. Αυτός δουλεύει ως εξής: αντιγράφει, μια προς μια, τις τιμές όλων των μελών του *d1* στα μέλη του *d2*.

Αυτό δεν είναι πάντοτε ικανοποιητικό και συχνά πρέπει να γράψουμε δικό μας δημιουργό αντιγραφής. Στην §20.2 είδαμε ένα τέτοιο παράδειγμα στην κλάση *BString*, για την οποίαν γράψαμε τον δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
   try {  bsData = new char[rhs.bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
```

```
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BString
```

Πότε γράφουμε δημιουργό αντιγραφής;

♦ ***Γράφουμε δημιουργό αντιγραφής όταν η αντιγραφή μέλος προς μέλος που θα κάνει αυτομάτως η C++ δεν κάνει αυτό που θέλουμε.***

Αυτό είναι πολύ γενικό· γιατί να μην πούμε «όταν τα αντικείμενά της κλάσης παίρνουν δυναμική μνήμη»; Αυτή είναι μια ειδική περίπτωση *δέσμευσης πόρων του συστήματος από αντικείμενα της κλάσης*. Και αυτή η διατύπωση όμως δεν καλύπτει τα πάντα.

- Αργότερα θα δούμε παράδειγμα όπου ο δημιουργός αντιγραφής δεν θα κάνει αντιγραφή αλλά *μεταβίβαση*.
- Υπάρχουν περιπτώσεις που θέλουμε απλώς να «αχρηστεύσουμε» τον δημιουργό αντιγραφής που γράφεται αυτομάτως.

Από τις κλάσεις που είδαμε μέχρι τώρα:

- Για τις *Date*, *GrElmn* και *RouteStop* δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής.
- Χρειάζεται όμως για τη *BString* και τη *Route* διότι οι τιμές των αντικειμένων αυτών τών κλάσεων δεν υπάρχουν εξ ολοκλήρου στις τιμές των μελών αλλά σε δυναμική μνήμη που δεν αντιγράφεται στην αντιγραφή μέλος προς μέλος.

Τώρα πρόσεξε το εξής:

♦ ***Η παράμετρος του δημιουργού αντιγραφής πρέπει να είναι παράμετρος αναφοράς. Κάθε δημιουργός αντιγραφής μιας κλάσης K θα δηλώνεται με επικεφαλίδα: "`K( const K& rhs )`".***

Στη συνέχεια θα καταλάβεις το γιατί.

Μπορεί ο δημιουργός αντιγραφής να να έχει και άλλες παραμέτρους; Ναι, αλλά θα πρέπει να εμφανίζονται μετά από την "`const K& rhs`" και *να έχουν όλες ερήμην καθορισμένες τιμές*.

Με τον δημιουργό αντιγραφής η C++ κάνει δύο ακόμη πολύ σημαντικές δουλειές:

- Πέρασμα παραμέτρων τιμής σε συνάρτηση.
- Επιστροφή τιμής (εκτέλεση της `return`) από συνάρτηση με τύπο.

Ας πούμε ότι έχουμε μια συνάρτηση:[1]

```
BString appendW( BString s )
{
   BString fv;
   char* sv( new char[s.length()+2] );
   strcpy( sv, s.c_str() );
   sv[s.length()] = 'W';  sv[s.length()+1] = '\0';
   fv = BString( sv );
cout << "returning from appendW" << endl;
   return fv;
} // appendW
```

Αλλάζουμε λίγο τους δημιουργούς της *BString*. Στον ερήμην δημιουργό βάζουμε:

```
BString::BString( const char* rhs )
{
cout << "In default constructor; rhs: " << rhs << endl;
   bsLen = cStrLen( rhs );
// . . .
```

και στον δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
cout << "In copy constructor; rhs: " << rhs.c_str() << endl;
   try {  bsData = new char [bsReserved];  }
```

[1] Τι κάνει αυτή η (άχρηστη!) συνάρτηση;

```
// . . .
```

Σε ένα πρόγραμμα καλούμε την *appendW()* ως εξής:

```
      BString q( "abc" ), t;
cout << "going to call appendW" << endl;
      t = appendW( q );
      cout << "q: " << q.c_str() << "   t: " << t.c_str() << endl;
```

και παίρνουμε:[2]

```
In default constructor; rhs: abc
In default constructor; rhs:
going to call appendW
In copy constructor; rhs: abc
In default constructor; rhs:
In default constructor; rhs: abcW
returning from appendW
In copy constructor; rhs: abcW
q: abc   t: abcW
```

Αμέσως μετά το μήνυμα «`going to call appendW`» βλέπεις το «`In copy constructor; rhs: abc`». Παρομοίως, αμέσως μετά το «`returning from appendW`» βλέπεις το «`In copy constructor; rhs: abcW`».

Στην §7.3 μάθαμε για τη κλήση συνάρτησης με παραμέτρους τιμής: «*Κατ' αρχήν παραχωρείται στη συνάρτηση ... μνήμη για όλα τα τοπικά αντικείμενα: [τυπικές παραμέτρους τιμής και τοπικές μεταβλητές]. Στη συνέχεια αντιγράφονται οι τιμές των πραγματικών παραμετρων στις αντίστοιχες τυπικές.*» Αυτό το «*στη συνέχεια αντιγράφονται οι τιμές*» μάλλον σε κάνει να σκεφτείς τον τελεστή εκχώρησης αλλά δεν είναι έτσι. Το μήνυμα «`In copy constructor; rhs: abc`» αμέσως μετά την κλήση της συνάρτησης μας δείχνει ότι *δημιουργείται ένα τοπικό αντικείμενο s με αρχική τιμή την τιμή της q.* Δηλαδή, όταν καλείται η συνάρτηση είναι σαν να εκτελείται η δήλωση:

```
BString s( q );
```

Στη συνέχεια, λέγαμε στην §7.3 «*Η τιμή της παράστασης που υπάρχει στη* `return` *... αντικαθιστά την κλήση της συνάρτησης.*» Στην περίπτωσή μας η *fv* που υπάρχει στη `return` είναι μια μεταβλητή τοπική της *appendW()*, που θα χαθεί όταν τελειώσει η εκτέλεση της συνάρτησης. Έτσι, δημιουργείται μια προσωρινή θέση στη μνήμη, ας την πούμε *tmp*, όπου αντιγράφεται η τιμή της *fv*. Και αυτό γίνεται με τον δημιουργό αντιγραφής· είναι σαν να εκτελείται η δήλωση:

```
BString tmp( fv );
```

και από αυτήν παίρνουμε το μήνυμα «`In copy constructor; rhs: abcW`».

Σημείωση: ▶
Αν αντί για "`return fv`" γράψεις "`return BString(sv)`" ή "`return sv`"το πιο πιθανό είναι να κληθεί μόνον ο ερήμην δημιουργός (με αρχική τιμή). Παρομοίως, θα κληθεί ο δημιουργός με αρχική τιμή αν καλέσεις την *appendW* με όρισμα `"abc"`. ◀

Θα πρέπει λοιπόν να θυμάσαι ότι:

- ***Το πέρασμα της τιμής της πραγματικής παραμέτρου στην τυπική παράμετρο τιμής, στη C++, γίνεται με τον τρόπο που δίνουμε αρχική τιμή σε μια μεταβλητή –δηλαδή με τον δημιουργό αντιγραφής– και όχι με εκχώρηση.***
- ***Η επιστροφή τιμής μιας συνάρτησης –με την εντολή* `return`*– στη C++, γίνεται με τον τρόπο που δίνουμε αρχική τιμή σε μια μεταβλητή: με τον δημιουργό αντιγραφής.***

Και τώρα ας ξανασκεφτούμε γιατί στον δημιουργό αντιγραφής περνούμε το προς αντιγραφή αντικείμενο *υποχρεωτικώς* με παράμετρο αναφοράς και όχι τιμής. Αν βάζαμε παράμετρο τιμής *θα χρησιμοποιούσαμε στον ορισμό του δημιουργού αντιγραφής τον δημιουργό*

[2] Borland C++, v.5.5

αντιγραφής (που θα ενεργοποιηθεί για το πέρασμα παραμέτρου τιμής), θα είχαμε δηλαδή μια αέναη αναδρομή!

21.3 Σειρά Δημιουργίας – Λίστα Εκκίνησης

Ας δούμε τώρα με ποια σειρά γίνεται η δημιουργία ενός αντικειμένου[3].

- Δημιουργούνται τα μέλη με τη σειρά που δηλώνονται.
- Εκτελείται το σώμα του δημιουργού.

  Αυτό φαίνεται και στο παρακάτω

Παράδειγμα 1

Το πρόγραμμα:

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class A
 4: {
 5:    int aM;
 6: public:
 7:    A( int ap = 0 )    // ερήμην δημιουργός
 8:    {  aM = ap;
 9:       cout << "A object created, aM = " << aM << endl;  }
10:    A( const A& rhs )  // δημιουργός αντιγραφής
11:    {  aM = rhs.aM;
12:       cout << "A object copy created, aM = " << aM << endl; }
13:    A& operator=( const A& rhs ) // τελεστής εκχώρησης
14:    {  aM = rhs.aM;
15:       cout << "A assignment, aM = " << aM << endl;
16:       return *this;  }
17:    int getM() const {  return aM;  }
18: }; // A
19:
20: class C
21: {
22:    int cM;
23: public:
24:    C( int cp = 0 )    // ερήμην δημιουργός
25:    {  cM = cp;
26:       cout << "C object created, cM = " << cM << endl;  }
27:    C( const C& rhs )  // δημιουργός αντιγραφής
28:    {  cM = rhs.cM;
29:      cout << "C object copy created, cM = " << cM << endl; }
30:    C& operator=( const C& rhs ) // τελεστής εκχώρησης
31:    {  cM = rhs.cM;
32:       cout << "C assignment, cM = " << cM << endl;
33:       return *this;  }
34:    int getM() const {  return cM;  }
35: }; // C
36:
37: class D
38: {
39:    A dA;
40:    C dC;
41: public:
42:    D( A ap=A(0), C cp=C(0) )  // ερήμην δημιουργός
43:    {  dA = A(ap.getM()+1);  dC = C(cp.getM()+1);
44:       cout << "D object created" << endl;  }
45:    D( const D& rhs )          // δημιουργός αντιγραφής
46:    {  dA = rhs.dA;  dC = rhs.dC;
47:       cout << "D object copy created" << endl;  }
```

[3] Ο κανόνας δεν είναι πλήρης. Θα επανέλθουμε...

```
48: }; // D
49:
50: int main()
51: {
52:    D d;
53: }
```

θα δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 0
C object created, cM = 0
A object created, aM = 1
A assignment, aM = 1
C object created, cM = 1
C assignment, cM = 1
D object created
```

Πριν δημιουργηθεί το αντικείμενο *d* (γρ. 52 στη **main**): Πρώτα (γρ. 42) δημιουργούνται τα αντικείμενα-παράμετροι *cp* και *ap* (με αυτήν τη σειρά: πρόσεξε ότι η λίστα ορισμάτων σαρώνεται από το τέλος προς την αρχή πρώτα το *cp* και μετά το *ap*) του δημιουργού και μας δίνουν τις δύο πρώτες γραμμές από τους ερήμην δημιουργούς (με αρχική τιμή) των *A* και *C*.

Μετά αρχίζει η δημιουργία του *d*:

- Πρώτα θα δημιουργηθούν τα μέλη *da* και *dc* από τους ερήμην δημιουργούς των κλάσεων *A* και *C* (μηνύματα: «**A object created**», «**C object created**»), με τη σειρά που είναι δηλωμένα (γρ. 39, 40).
- Στη συνέχεια θα εκτελεσθεί το σώμα του δημιουργού της *D*: Καλείται ο δημιουργός (με αρχική τιμή) της *A* για να δημιουργήσει ένα αντικείμενο "**A(ap.getM()+1)**" (γρ. 43, μήνυμα: «**A object created, aM = 1**») που το εκχωρεί στο *da* («**A assignment, aM = 1**»). Παρόμοια συμβαίνουν και για το *dc* (μηνύματα: «**C object created, cM = 1**», «**C assignment, cM = 1**»). Τελικώς, θα μας δοθεί το μήνυμα: «**D object created**».

Αλλάζουμε τον δημιουργό της *D* (γρ. 42-44) ως εξής:

```
42:    D( A ap=A(0), C cp=C(0) )  // ερήμην δημιουργός
43:    : dA( A(ap.getM()+1) ),  dC( C(cp.getM()+1) )
44:    {  cout << "D object created" << endl;  }
```

δηλαδή: δίνουμε τιμές στα μέλη *da* και *dc* με **λίστα εκκίνησης** (initialization list) και όχι με τις εντολές εκχώρησης της γρ. 43. Τώρα η εκτέλεση θα δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 1
C object created, cM = 1
D object created
```

Ποια είναι η διαφορά από πριν; Τώρα τα δύο μέλη του *d* δημιουργούνται από τους δημιουργούς αντιγραφής με τις τιμές που πρέπει να έχουν. Έτσι *δεν χρειάζεται να γίνουν οι δύο εκχωρήσεις μέσα στο σώμα του δημιουργού* και ο δημιουργός γίνεται ταχύτερος.[4]

Παράδειγμα 2

Χρησιμοποιώντας λίστα εκκίνησης θα μπορούσαμε να γράψουμε τον δημιουργό της *Date* ως εξής:

```
Date::Date( int yp, int mp, int dp )
   : year( yp ),  month( mp ),  day( dp )
{
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   if ( mp <= 0 || 12 < mp )
```

[4] Στη συνέχεια θα δεις ότι δεν είναι μόνο θέμα ταχύτητας.

```
        throw DateXptn( "Date", DateXptn::monthErr, mp );
    if ( dp <= 0 || lastDay(year, month) < dp )
        throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
} // Date
```

Πρόσεξε ότι τώρα:

- Πρώτα δίνουμε τις τιμές στα μέλη –με τη δημιουργία τους– και μετά κάνουμε τους ελέγχους.
- *Οι έλεγχοι γίνονται πάντοτε στις τιμές των παραμέτρων και όχι των μελών.* (γιατί;)

Οι αρχικές τιμές που βάζουμε στη λίστα εκκίνησης είναι, γενικώς, σταθερές παραστάσεις των τυπικών παραμέτρων.

Παράδειγμα 3

Για την *complex* (§19.5) θα μπορούσαμε να γράψουμε:

```
complex::complex( double rp = 0.0, double ip = 0.0 )
    :  re( rp ), im( ip ) { };
```

Θα μπορούσαμε να γράψουμε έτσι και τον δημιουργό της *BString;* Όχι! Με τη "`bsData( rhs )`" *θα αντιγραφεί η τιμή του βέλους rhs στο βέλος bsData.*

Έτσι, οι δημιουργοί των κλάσεων εξαιρέσεων θα γραφούν ως εξής:

```
DateXptn( const char* mn, int ec, int ev1 = 0,
          int ev2 = 0, int ev3 = 0 )
    : errorCode( ec ),
      errVal1( ev1 ),  errVal2( ev2 ),  errVal3( ev3 )
{  strncpy( funcName, mn );  funcName[99] = '\0';  }
```

και

```
BStringXptn( char* mn, int ec, int ev = 0 )
    :  errorCode( ec ),  errorValue( ev )
{  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
```

21.4 Εξαιρέσεις από τον Δημιουργό

Ο δημιουργός είναι μια συνάρτηση που πρέπει τελικώς να δημιουργήσει ένα αντικείμενο που συμμορφώνεται με την αναλλοίωτη.[5]

♦ ***Το (κάθε) αντικείμενο δημιουργείται όταν ολοκληρωθεί η εκτέλεση του δημιουργού.***

Απο εκεί και μετά μπορείς να το χειριστείς (και να το καταστρέψεις με τον καταστροφέα.) Η ζωή του αντικειμένου τελειώνει όταν αρχίσει η εκτέλεση του καταστροφέα του.

Αν δεν είναι δυνατό να έχουμε αντικείμενο που συμμορφώνεται με την αναλλοίωτη το καλύτερο που έχουμε να κάνουμε είναι να ρίξουμε εξαίρεση. Τι γίνεται όμως σε μια τέτοια περίπτωση; Έχουμε ένα «μισοδημιουργημένο» αντικείμενο; Όχι ή, εν πάση περιπτώσει, δεν έχουμε κάτι το οποίο μπορούμε να χειριστούμε.

Και τι γίνεται αν, πριν ριχτεί η εξαίρεση, έχουμε δεσμεύσει ήδη κάποιους πόρους του συστήματος; Θα πρέπει να τους απελευθερώσουμε.

Πριν δούμε σχετικά παραδείγματα ας πούμε τα καλά νέα: Αν κατά τη δημιουργία κάποιου μέλους έχουμε αποτυχία και ριχτεί εξαίρεση θα κληθούν οι καταστροφείς των μελών που ήδη δημιουργήθηκαν για να τα καταστρέψουν.

Πάμε τώρα στα δύσκολα με άλλο ένα παράδειγμα δημιουργού αντιγραφής: θα γράψουμε δημιουργό αντιγραφής για τη *Route*. Αντιγράφοντας αυτόν που γράψαμε για τη *BString* παίρνουμε:

---

[5] Η αναλλοίωτη της κλάσης είναι η *απαίτηση* για τον (κάθε) δημιουργό. Και ποια είναι η *προϋπόθεση;* Εξαρτάται από το είδος του δημιουργού.

```
Route::Route( const Route& rhs )
{
   try {  rAllStops = new RouteStop[rhs.rReserved];  }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rReserved = rhs.rReserved;
   for ( int k(0); k < rhs.rNoOfStops; ++k )
      rAllStops[k] = rhs.rAllStops[k];
   rNoOfStops = rhs.rNoOfStops;
   rCode = rhs.rCode;
   rFrom = rhs.rFrom;
   rTo = rhs.rTo;
   rInBetween = rhs.rInBetween;
} // Route::Route
```

Αυτός όμως δεν είναι σωστός. Κάθε `rAllStops[k]`, τύπου *RouteStop*, έχει ένα μέλος τύπου (*std*::)*string*. Η αντιγραφή του γίνεται όπως περίπου είδαμε στην επιφόρτωση του "`operator=`" για τη *BString* (§20.4)· δηλαδή για κάθε στοιχείο δεσμεύεται δυναμική μνήμη. Επομένως είναι δυνατόν να ριχτεί *bad_alloc* και όταν εκτελείται κάποια από αυτές τις αντιγραφές. Θα πρέπει λοιπόν, προβλέποντας μια τέτοια περίπτωση, να γράψουμε:

```
   try
   {
      for ( int k(0); k < rhs.rNoOfStops; ++k )
         rAllStops[k] = rhs.rAllStops[k];
   }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
```

Αλλά και αυτό είναι λάθος! Γιατί; Διότι έτσι θα έχουμε διαρροή μνήμης: η δυναμική μνήμη που πήραμε για τον `rAllStops` δεν έχει ανακυκλωθεί. Αυτό θα πρέπει να γίνει ως εξής:

```
   catch( bad_alloc )
   {  delete[] rAllStops;
      throw RouteXptn( "Route", RouteXptn::allocFailed );  }
```

και τελικώς θα έχουμε:

```
Route::Route( const Route& rhs )
{
   try {  rAllStops = new RouteStop[rhs.rReserved];  }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rReserved = rhs.rReserved;
   try
   {
      for ( int k(0); k < rhs.rNoOfStops; ++k )
         rAllStops[k] = rhs.rAllStops[k];
      rFrom = rhs.rFrom;
      rTo = rhs.rTo;
      rInBetween = rhs.rInBetween;
   }
   catch( bad_alloc )
   {  delete[] rAllStops;
      throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rNoOfStops = rhs.rNoOfStops;
   rCode = rhs.rCode;
} // Route::Route
```

Πρόσεξε ότι μέσα στην περιοχή της (δεύτερης) **try** βάλαμε και τις αντιγραφές των *rFrom*, *rTo* και *rInBetween* αφού και αυτές μπορεί να δώσουν το ίδιο πρόβλημα. Αυτό θα πρέπει να το πάρουμε υπόψη μας και στις *setFrom*, *setTo* και *setInBetween*. Να τις ξαναγράψεις σωστά!

Γενικώς λοιπόν:

♦ ***Αν ρίξεις εξαίρεση από κάποιον δημιουργό φρόντισε πρώτα να αποδεσμεύσεις τους πόρους του συστήματος που έχεις δεσμεύσει για το αντικείμενο που δημιουργείς.***

Και αν η εξαίρεση ριχτεί από κάποια συνάρτηση που καλεί ο δημιουργός; Φροντίζουμε:

- να πιάσουμε την εξαίρεση στον δημιουργό,
- να αποδεσμεύσουμε τους πόρους που έχουμε δεσμεύσει και μετά
- ξαναρίχνουμε την εξαίρεση.

Αν θέλεις να λύσεις –τώρα ή αργότερα– ένα πρόβλημα τέτοιου είδους σκέψου την εξής περίπτωση: Στη *RouteStop* δηλώνουμε:

```
BString sName;  // όνομα στάσης
```

και έχουμε την εξής αλλαγή: Αν κάτι δεν πάει καλά σε κάποια από τις εκχωρήσεις "`rAllStops[k] = rhs.rAllStops[k]`" η εξαίρεση που θα πάρεις (δες την παρατήρηση που ακολουθεί) δεν θα είναι *bad_alloc* αλλά *BStringXptn*(με κωδικό *allocFailed*). Πώς θα τη χειριστείς;

Παρατήρησεις: ▶

1. Είπαμε ότι για τη *RouteStop* δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής. Όπως είδαμε όμως, *αυτό δεν σημαίνει ότι όταν αντιγράφουμε (με εκχώρηση) αντικείμενα αυτής της κλάσης δεν μπορεί, κατ' αρχήν, να ριχτεί εξαίρεση*. Προφανώς το ίδιο ισχύει και για αντιγραφή με τον δημιουργό αντιγραφής.
2. Τα πράγματα με την πρώτη **try** είναι πιο πολύπλοκα: Τι γίνεται αν μας δοθεί μνήμη για να δημιουργηθούν *m* (< *rhs.rReserved*) αντικείμενα και μετά εμφανισθεί το πρόβλημα; Έχουμε διαρροή μνήμης που δεσμεύτηκε για τα αντικείμενα αυτά! Η C++ έχει τρόπο (ριζικής) αντιμετώπισης όλων αυτών των προβλημάτων και θα τον μάθουμε αργότερα.◀

21.5 Ο Καταστροφέας

Ας ξεκινήσουμε με το πρόβλημα: έστω ότι έχουμε μια ομάδα μέσα στην οποία δηλώνουμε ένα αντικείμενο:

```
{
    BString a;
// . . .
} // T: εδώ πρέπει να καταστραφεί το a.
```

Το *a* είναι τοπικό στην ομάδα και θα πρέπει να καταστραφεί όταν η εκτέλεση φθάσει στο σημείο *T*, στο τέλος της ομάδας. Η C++ μας εγγυάται ότι θα επιστρέψει στη στοίβα τη μνήμη που κρατάμε για τα μέλη *bsData*, *bsLen*, *bsReserved* αλλά η δυναμική μνήμη που πήραμε από τον σωρό (και τη δείχνει το *bsData*) θα μείνει δεσμευμένη (και άχρηστη αφού το βέλος προς αυτήν ήταν το *bsData*). Όπως είδαμε, η C++ μας επιτρέπει να γράψουμε μια ειδική συνάρτηση-μέλος, που λέγεται **καταστροφέας** (destructor) που θα τον καλέσει όταν καταστρέφεται ένα αντικείμενο. Με τη συνάρτηση αυτή θα πρέπει να φροντίσουμε να επιστρέφεται η δυναμική μνήμη στον σωρό.

Για τη *BString* χρειάζεται να ορίσουμε *εμείς* καταστροφέα:

```
BString::~BString()
{
   delete[] bsStart;
} // BString::~BString
```

αλλά για τη *Date* δεν χρειάζεται. Για αυτήν, ο μεταγλωττιστής θα ορίσει αυτομάτως έναν *εννοούμενο* καταστροφέα· μπορείς να τον σκέφτεσαι κάπως έτσι:

```
   ~Date() { };
```

και είναι δεκτός. Γενικώς:

♦ ***Πρέπει να γράφουμε καταστροφέα για μια κλάση που τα αντικείμενά της δεσμεύουν πόρους του συστήματος, ώστε με την καταστροφή του αντικειμένου να αποδεσμεύονται οι πόροι.***

Αν έχουμε μια κλάση *K*:

- μπορούμε να γράψουμε *έναν καταστροφέα το πολύ* (ενώ μπορούμε να έχουμε περισσότερους από έναν δημιουργούς),
- ο καταστροφέας έχει όνομα "`~K`",
- ο καταστροφέας *δεν έχει παραμέτρους*.

Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα

```
~K() { };
```

στις κλάσεις που δεν χρειάζονται. Αργότερα θα δούμε ότι σε μερικές περιπτώσεις αυτό είναι απαραίτητο.

Με ποιες προδιαγραφές γράφουμε τον καταστροφέα;

Προϋπόθεση: Η αναλλοίωτη της κλάσης· το αντικείμενο που καταστρέφεται –όπως και κάθε αντικείμενο– συμμορφώνεται με την αναλλοίωτη της κλάσης. Για παράδειγμα, για να εκτελεσθεί η "`delete[] bsData`" η τιμή του *bsData* θα πρέπει να είναι είτε 0 (μηδέν) είτε μια διεύθυνση στη δυναμική μνήμη από όπου ξεκινάει η αποθήκευση πίνακα. Η αναλλοίωτη της *BString* –και πιο συγκεκριμένα η "0 < *bsReserved*"– μας εγγυάται την ύπαρξη του δυναμικού πίνακα.

Απαίτηση: Οι πόροι –που ήταν δεσμευμένοι από το αντικείμενο που καταστράφηκε– έχουν αποδεσμευθεί.

Τώρα θα κάνουμε ένα πείραμα –με ένα προγραμματάκι– για να δούμε πότε καλείται ένας καταστροφέας. Αλλάζουμε τον καταστροφέα της *BString* σε:

```
BString::~BString()
{
cout << "destructing BString " << c_str() << endl;
   delete[] bsData;
} // BString::~BString
```

και δοκιμάζουμε το

```
#include <iostream>
#include "BString.cpp"
using namespace std;

struct S
{
   BString sName;
   float   sDist;
}; // S

int main()
{
   {
      BString a( "aaa" );
// . . .
      cout << "going to destroy a" << endl;
   } // T: εδώ πρέπει να καταστραφεί το a.

   BString* b( new BString("bbb") );
// . . .
   cout << "going to delete b" << endl;
   delete b;

   BString* c( new BString[2] );
   c[0] = "c0c0";  c[1] = "c1c1";
// . . .
   cout << "going to delete c" << endl;
   delete[] c;
```

```
   {
      S s;
      s.sName = "sssss";  s.sDist = 7.33;
// . . .
      cout << "going to destroy s" << endl;
   } // T: εδώ πρέπει να καταστραφεί το s.

   S* s( new S );
   s->sName = "psps";  s->sDist = 7.33;
// . . .
   cout << "going to delete s" << endl;
   delete s;
}
```

Αποτέλεσμα:

```
going to destroy a
destructing BString aaa
going to delete b
destructing BString bbb
going to delete c
destructing BString c1c1
destructing BString c0c0
going to destroy s
destructing BString sssss
going to delete s
destructing BString psps
```

Τι βλέπουμε εδώ;

- Κατ' αρχάς δίνουμε το παράδειγμα με το οποίο ξεκινάει αυτή η παράγραφος: «*Το a είναι τοπικό στην ομάδα και θα πρέπει να καταστραφεί όταν η εκτέλεση φθάσει στο σημείο T, στο τέλος της ομάδας.*» Εδώ το βλέπεις.
- Στα δύο επόμενα βλέπεις το εξής: αφού για να δημιουργηθούν δυναμικές μεταβλητές ή πίνακες καλείται ο δημιουργός, για να ανακυκλωθούν καλείται ο καταστροφέας.
- Τέλος, στα δύο τελευταία παραδείγματα βλέπεις το εξής: Ο καταστροφέας των αντικειμενων μιας κλάσης –εδώ ο εννοούμενος της κλάσης *S*– καλεί τους καταστροφείς των μελών των αντικειμένων.

Πώς γίνεται η καταστροφή του αντικειμένου;

♦ ***Η καταστροφή ενός αντικειμένου γίνεται με την αντίστροφη σειρά από αυτήν που γίνεται η δημιουργία του.***

Με αυτά που ξέρουμε μέχρι τώρα:

- Εκτελείται το σώμα του καταστροφέα.
- Καταστρέφονται τα μέλη με αντίστροφη σειρά από αυτήν που δηλώνονται.

Μπορείς να δεις ένα παράδειγμα πολύ εύκολα. Εφοδίασε τις κλάσεις του Παραδ. 1 της §21.3 με τους καταστροφείς:

```
   ~A()
   {  cout << "destructing an A object aM = " << aM << endl; }

   ~C()
   {  cout << "destructing a C object aM = " << cM << endl; }

   ~D()
   {  cout << "destructing a D object" << endl; }
```

και η εκτέλεση του προγράμματος θα μας δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 1
C object created, cM = 1
D object created
```

```
destructing an A object aM = 0
destructing a C object aM = 0
destructing a D object
destructing a C object aM = 1
destructing an A object aM = 1
```

Οι δύο πρώτες καταστροφές προέρχονται από τις καταστροφές των "`A(0)`" και "`C(0)`" και δεν έχουν σχέση με την καταστροφή του *d*· αυτή ξεκινάει με την εκτέλεση του σώματος του καταστροφέα που μας δίνει το μήνυμα "`destructing a D object`". Στη συνέχεια βλέπεις ότι καταστρέφονται πρώτα το *dc* και μετά το *da*.

21.5.1 Ο Καταστροφέας *δεν* Ρίχνει Εξαιρέσεις

Ένα θέμα που χρειάζεται προσοχή είναι οι εξαιρέσεις του καταστροφέα:

- ***Ένας καταστροφέας όχι μόνο δεν ρίχνει εξαιρέσεις αλλά δεν επιτρέπει να βγαίνουν από αυτόν εξαιρέσεις από συναρτήσεις που καλεί.***

Γιατί; Σκέψου την εξής περίπτωση: Μέσα σε μια συνάρτηση ρίχνεται μια εξαίρεση και αρχίζει το «ξετύλιγμα της στοίβας». Πρώτο βήμα: σταματάει η εκτέλεση της συνάρτησης και καταστρέφονται όλα τα τοπικά αντικείμενά της. Τι θα γίνει αν κατά την καταστροφή κάποιου από αυτά τα αντικείμενα πέσει μέσα στον καταστροφέα μια εξαίρεση; Σταματάει η εκτέλεση του καταστροφέα, δηλαδή η καταστροφή του αντικειμένου. Όποιο σενάριο διαχείρισης εξαιρέσεων και να έχεις στήσει καταρρέει (μαζί και το πρόγραμμα).

Τώρα θα πεις: τι εξαίρεση από τον καταστροφέα, από μισή γραμμή συνάρτηση;! Ε, έτσι είναι στη *BString*. Υπάρχουν και πιο πολύπλοκοι καταστροφείς· στη συνέχεια θα δούμε ένα παράδειγμα.

21.5.2 Καλούμε τον Καταστροφέα;

Σε όλα τα παραδείγματα που δώσαμε παραπάνω ο καταστροφέας καλείται αυτομάτως χωρίς εμείς να γράψουμε κάτι στο πρόγραμμά μας. Υπάρχει περίπτωση να χρειαστεί να τον καλέσουμε με δική μας εντολή όπου να βλέπουμε κάτι σαν "`~BString()`" ή "`~Date()`"; Σπανίως! Δες ένα παράδειγμα, αφού πρώτα κάνεις μια επανάληψη στην «τρίτη μορφή του "`new`"» (§16.5).

Αλλάζοντας λίγο το παράδειγμα που δίναμε εκεί, γράφουμε:

```
   char* buf( new char[100] );
   BString* bs( new (buf) BString );
// . . .
   delete[] buf;
```

Τα μέλη της `*`*bs* θα υλοποιηθούν «πάνω» στα στοιχεία του buf. Για παράδειγμα, τα μέλη *bsData*, *bsLen*, *bsReserved* θα μπορούσαν να υλοποιηθούν ξεκινώντας από buf`[0]`, buf`[4]`, buf`[8]` αντιστοίχως. Αλλά η (δυναμική) μνήμη που δείχνει το *bsData* θα παραχωρηθεί αλλού (όχι πάνω στον buf). Όταν εκτελεσθεί η "`delete[] buf`" θα ανακυκλωθούν όλα τα μέλη του `*`*bs*, μεταξύ αυτών και το βέλος *bs*`->`*bsData*, αλλά θα μας μείνει αυτό που δείχνει. Έχουμε δηλαδή διαρροή μνήμης. Πώς το αποφεύγουμε; Βάζοντας:

```
// . . .
   bs->~BString();
   delete[] buf;
```

Το ίδιο θα πρέπει να κάνουμε και αν θέλουμε να αλλάξουμε τη χρήση της buf:

```
   char* buf( new char[100] );
   BString* bs( new (buf) BString );
// . . .
   bs->~BString();
   double* d( new (buf) double );
// . . .
```

```
   delete[] buf;
```

Παρατήρηση: ▶
Μετά την κλήση "`bs->~BString()`"ο δυναμικός πίνακας που δείχνει το βέλος *bs*->*bsData* δεν υπάρχει πια· έχει ανακυκλωθεί. Το *bs*->*bsReserved* έχει θετική τιμή αλλά είναι ψεύτικη.◀

21.6 Ο Τελεστής Εκχώρησης

Στο προηγούμενο κεφάλαιο (§20.4) είδαμε ότι το πρόβλημα που μας υποχρέωνε να γράψουμε δημιουργό αντιγραφής για τη *BString* μας υποχρέωνε να επιφορτώσουμε και τον τελεστή εκχώρησης που τον ορίσαμε ως εξής:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )  // if ( !αυτοεκχώρηση )
   {
   // Πάρε τη μνήμη που χρειάζεσαι
      char* tmp;
      try {  tmp = new char[rhs.bsReserved];  }
      catch( bad_alloc& )
      { throw BStringXptn( "operator=", BStringXptn::allocFailed ); }
   // "Καθάρισε" την παλιά τιμή
      delete[] bsData;
   // Αντίγραψε την τιμή του rhs
      bsData = tmp;
      bsReserved = rhs.bsReserved;
      for ( int k(0); k < rhs.bsLen; ++k )
         bsData[k] = rhs.bsData[k];
      bsLen = rhs.bsLen;
   }
   return *this;
} // BString::operator=
```

Κάναμε μάλιστα την επισήμανση: «*Εδώ πρόσεξε ότι παίρνουμε τη μνήμη με ένα βοηθητικό βέλος (tmp). Ακόμη, κατά τη συνήθειά μας, βάλαμε εντολές που πιάνουν πιθανή εξαίρεση bad_alloc και ρίχνουν δική μας BStringXptn (allocFailed). Οι εντολές που ακολουθούν δεν θα εκτελεσθούν αν δεν παραχωρηθεί η μνήμη που ζητούμε αφού στην περίπτωση αυτή θα ριχτεί εξαίρεση.*» Συνεπώς, αν κατά την εκτέλεση της "`a = b`" «μείνουμε» από μνήμη θα ριχτεί εξαίρεση αλλά δεν θα καταστραφεί η παλιά τιμή της *a*. Όπως καταλαβαίνεις, έτσι αφήνουμε αρκετές επιλογές –στο πρόγραμμα που χρησιμοποιεί την κλάση– για τη διαχείριση της κατάστασης.

Θα τηρούμε λοιπόν γενικώς τον εξής προγραμματιστικό κανόνα:

- ***Κάθε μέθοδος που αποπειράται να αλλάξει την τιμή ενός αντικειμένου (εκχώρηση, ανάγνωση από αρχείο) αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.***

Λέμε ότι η μέθοδος αυτή έχει **ασφάλεια προς τις εξαιρέσεις** (exception safety) επιπέδου **ισχυρής εγγύησης** (strong guarantee). Αργότερα θα δούμε την ασφάλεια προς τις εξαιρέσεις πιο εκτεταμένα.

21.6.1 Η Ασφαλής *swap()*

Εδώ αξίζει να αναφέρουμε μια κομψή μορφή του τελεστή εκχώρησης που προτείνεται από τον (Stepanov[6], 2007) και έχει ασφάλεια ισχυρής εγγύησης. Αυτή η μορφή βασίζεται στον δημιουργό αντιγραφής και σε μια ασφαλή μέθοδο *swap()*.

---

[6] A.A. Stepanof, ρώσος προγραμματιστής (και όχι μόνον), «πατέρας» της STL.

Ας ξεκινήσουμε με τη *BString*. Αν θέλαμε να γράψουμε μια μέθοδο *swap()*, με αυτά που ξέρουμε μέχρι τώρα, θα εξειδικεύαμε («με το χέρι») το περίγραμμα της §14.7.1:

```
void BString::swap( BString& rhs )
{
   BString s( *this );
   *this = rhs;  rhs = s;
} // BString::swap
```

Εδώ όμως χρησιμοποιούμε τον δημιουργό αντιγραφής και τον τελεστή εκχώρησης που μπορεί να ρίξουν εξαιρέσεις. Επομένως δεν είναι ασφαλής. Το πιο βασικό πρόβλημα είναι ότι θέλουμε να τη χρησιμοποιήσουμε *για να ορίσουμε τον τελεστή εκχώρησης* και αυτή η μορφή τον χρησιμοποιεί!

Μπορούμε να γράψουμε μια ασφαλή *swap()* αν κάνουμε τις ανταλλαγές τιμών μέλος-προς-μέλος, δηλαδή:

```
void BString::swap( BString& other )
{
   char* pSave( bsData );
   bsData = other.bsData;  other.bsData = pSave;

   size_t save( bsLen );
   bsLen = other.bsLen;  other.bsLen = save;

   save = bsReserved;
   bsReserved = other.bsReserved;  other.bsReserved = save;
} // BString::swap
```

Πρόσεξε τα εξής:

- Στην πρώτη αντιμετάθεση έχουμε ανταλλαγή τιμών των *βελών bsData* και *rhs.bsData* και όχι των ορμαθών που δείχνουν.
- Η ασφάλεια της μεθόδου στηρίζεται στο ότι ανταλλάσσονται απλές τιμές τύπου `char*` ή `size_t`. Δεν καλεί δημιουργό ή τον τελεστή εκχώρησης της *BString*.[7]

Τώρα, μπορούμε να γράψουμε:[8]

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      BString tmp( rhs );
      swap( tmp );
   }
   return *this;
} // BString::operator=
```

Αν ρίξει εξαίρεση ο δημιουργός αντιγραφής, η εκτέλεση της "`BString tmp(rhs)`" δεν θα ολοκληρωθεί, η *swap()* δεν θα κληθεί και το `*this` δεν θα πειραχτεί. Αν το *tmp* δημιουργηθεί κανονικώς τότε όλα θα πάνε καλώς αφού η *swap()* αποκλείεται να ρίξει εξαίρεση.

Σύγκρινε αυτήν τη μορφή με αυτήν της προηγούμενης παραγράφου: στην πρώτη μορφή ήταν υποχρέωση δική μας να δεσμεύσουμε πόρους (δυναμική μνήμη) και να τη απελευθερώσουμε. Αυτά πρέπει να γίνουν με προσοχή, όπως ήδη είδαμε. Στη δεύτερη μορφή:

- Η δέσμευση πόρων γίνεται με τη δημιουργία ενός αντικειμένου (*tmp*).
- Η απελευθέρωση των πόρων γίνεται με την καταστροφή του αντικειμένου.

---

[7] Λέμε ότι η *swap()* είναι ασφαλής προς τις εξαιρέσεις σε επίπεδο **εγγύησης μη-ρίψης** (no-throw guarantee).

[8] Αυτή η μορφή επιφόρτωσης του "`operator=`" θα δεις να αναφέρεται και ως *κανονική μορφή* (canonical form).

Χρησιμοποιούμε δηλαδή την τεχνική RAII με πολύ πιο φυσικό τρόπο από αυτόν που είδαμε στο παράδειγμα της §16.7.1.

Θα πρέπει να επισημάνουμε ακόμη την τυποποιημένη μορφή· αν θελήσεις να τον γράψεις για τη *Route* θα έχεις:

```
Route& Route::operator=( const Route& rhs )
{
   if ( &rhs != this )
   {
      Route tmp( rhs );
      swap( tmp );
   }
   return *this;
} // Route::operator=
```

Βέβαια, θα πρέπει να γράψουμε τη *swap()*· αυτό σου το αφήνουμε ως άσκηση.

Θα δηλώσουμε τη *swap()* ως "`private`" ή "`public`"; Δεν έχουμε λόγο να τη δηλώσουμε "`private`"· πέρα από τη χρήση της στις άλλες μεθόδους, την παρέχουμε ως ένα επιπλέον εργαλείο προς τον προγραμματιστή-χρήστη της κλάσης. Θυμίσου άλλωστε ότι η *string*`::`*swap()* είναι "`public`".

Παρατηρήσεις: ▶

1. Αν χρησιμοποιήσεις το περίγραμμα `std::swap()` μπορείς να γράψεις πολύ πιο απλά:

```
void BString::swap( BString& other )
{
   std::swap( bsData, other.bsData );
   std::swap( bsLen, other.bsLen );
   std::swap( bsReserved, other.bsReserved );
} // BString::swap
```

Στη συνέχεια θα χρησιμοποιούμε αυτήν τη μορφή.

2. Όποιος διάβασε με προσοχή τα προηγούμενα μπορεί να έχει σκεφτεί ήδη μια βελτίωση. Αν αντί για "`const BString& rhs`" βάλουμε παράμετρο τιμής, δηλαδή "`BString rhs`" όταν εκτελείται η "`a = b`" θα κληθεί ο δημιουργός αντιγραφής για να δημιουργήσει ένα τοπικό αντίγραφο της *b* στην *rhs* που θα καταστραφεί όταν τελειώσει η εκτέλεση της συνάρτησης. Έτσι, η *tmp* μας είναι άχρηστη!

```
BString& BString::operator=( BString rhs )
{
   if ( &rhs != this )
   {
      swap( rhs );
   }
   return *this;
} // BString::operator=
```

Στην περίπτωση αυτήν *θα γίνεται πάντοτε αντιγραφή στο τοπικό αντικείμενο* ενώ στην προηγούμενη μορφή δεν γίνεται στην περίπτωση αυτοεκχώρησης. Σωστό! Αλλά πόσες αυτόεκχωρήσεις θα ζητηθούν σε κάποιο πρόγραμμα;

3. Αν υπάρξει κάποιο πρόβλημα, κατά τη χρήση αυτού του τελεστή, η εξαίρεση που θα ριχτεί θα δείχνει προέλευση τον δημιουργό (αντιγραφής). Αυτό φυσικά είναι παραπλανητικό· ας το διορθώσουμε:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
```

```
} // BString::operator=
```

Δηλαδή: πιάνουμε την όποια εξαίρεση *BStringXptn*, διορθώνουμε το *funcName* και την ξαναρίχνουμε.◀

Η χρήση της ασφαλούς *swap()* μπορεί να μας δώσει ισχυρή εγγύηση ασφάλειας προς τις εξαιρέσεις όχι μόνον για τον τελεστή εκχώρησης αλλά και για άλλες μεθόδους που αλλάζουν την τιμή του αντικειμένου, π.χ. μεθόδους ανάγνωσης από αρχείο. Θα δώσουμε τώρα ένα τέτοιο παράδειγμα: Θα λύσουμε μερικώς την Άσκ. 20-5. Θα γράψουμε μια μέθοδο

```
void RouteStop::readFromText( istream& tin )
```

που θα διαβάζει τιμή αντικειμένου *RouteStop* από αρχείο και θα είναι ασφαλής προς τις εξαιρέσεις: το αντικείμενο-ιδιοκτήτης της θα κρατάει την τιμή που έχει αν ριχτεί εξαίρεση. Θα χρησιμοποιήσουμε τη:

```
void RouteStop::swap( RouteStop& other )
{
   sName.swap( other.sName );
   std::swap( sDist, other.sDist );
   std::swap( sFare, other.sFare );
} // RouteStop::swap
```

Να σημειώσουμε ότι η `string::swap()` είναι ασφαλής.

Το *tin* είναι ρεύμα από ένα αρχείο text που είναι ήδη ανοικτό για διάβασμα. Σε κάθε γραμμή του αρχείου αναμένουμε τα στοιχεία μιας στάσης:

```
 0: void RouteStop::readFromText( istream& tin )
 1: {
 2:    try
 3:    {
 4:       string line;
 5:       getline( tin, line, '\n' );
 6:       if ( tin.fail() )
 7:          throw RouteStopXptn( "readFromText",
 8:                               RouteStopXptn::cannotRead );
 9:       if ( line.empty() )
10:          throw RouteStopXptn( "readFromText",
11:                               RouteStopXptn::lineEmpty );
12:       size_t t1Ndx( line.find('\t', 0) );
13:       if ( t1Ndx == string::npos )
14:          throw RouteStopXptn( "readFromText",
15:                               RouteStopXptn::incomplete );
16:       size_t t2Ndx( line.find('\t', t1Ndx+1) );
17:       if ( t1Ndx == string::npos )
18:          throw RouteStopXptn( "readFromText",
19:                               RouteStopXptn::incomplete );
20:       RouteStop tmp;
21:       if ( t1Ndx == 0 )
22:          throw RouteStopXptn( "readFromText",
23:                               RouteStopXptn::noName );
24:       tmp.sName = line.substr( 0, t1Ndx );
25:       string buf( line.substr(t1Ndx+1, t2Ndx-t1Ndx-1) );
26:       tmp.sDist = atof( buf.c_str() );
27:       if ( tmp.sDist < 0 )
28:          throw RouteStopXptn( "readFromText",
29:                               RouteStopXptn::negDist,
30:                               tmp.sDist );
31:       buf = line.substr( t2Ndx+1 );
32:       tmp.sFare = atof( buf.c_str() );
33:       if ( tmp.sFare < 0 )
34:          throw RouteStopXptn( "readFromText",
35:                               RouteStopXptn::negFare,
36:                               tmp.sFare );
37:       this->swap( tmp );
38:    }
39:    catch( bad_alloc )
40:    {  throw RouteStopXptn( "readFromText",
```

```
41:                              RouteStopXptn::allocFailed );  }
42: } // RouteStop::readFromText
```

Μετά τη δήλωση της γρ. 4, διαβάζουμε με τη *getline()* (γρ. 5). Αν η *getline()* εκτελέστηκε κανονικώς θα προχωρήσουμε, αλλιώς... εξαίρεση (γρ. 6-8).

Αν διαβάσαμε άδεια γραμμή και πάλι ρίχνουμε εξαίρεση (γρ. 9-11).

Αν δεν είναι άδεια *θα πρέπει να έχει δύο φορές τον* `'\t'`:

- Χρησιμοποιώντας τη *find()* της *string* επιδιώκουμε να αποθηκεύσουμε τη θέση του πρώτου στην *t1Ndx* (γρ. 12).
- Και πάλι με τη *find()* αλλά ξεκινώντας από τη θέση *t1Ndx*+1 επιδιώκουμε να αποθηκεύσουμε τη θέση του δεύτερου στην *t2Ndx* (γρ. 16).

Αποτυχία οποιασδήποτε από τις δύο *find()* οδηγεί στη ρίψη εξαίρεσης (γρ. 13-15, 17-19).

Αφού ο πρώτος `'\t'` βρίσκεται στη θέση *t1Ndx* το όνομα της στάσης βρίσκεται στις θέσεις 0 `..` *t1Ndx*–1 (εκτός από την περίπτωση που *t1Ndx* `==` 0). Αφού ο δεύτερος `'\t'` βρίσκεται στη θέση *t2Ndx* στις θέσεις *t1Ndx*+1 `..` *t2Ndx*–1 υπαρχει η απόσταση και από *t2Ndx*+1 μέχρι το τέλος της γραμμής υπαρχει το κόμιστρο. Όλα αυτά θα αποθηκευτούν σε μια βοηθητική μεταβλητή (γρ. 20). Παραλλήλως θα γίνονται και οι σχετικοί έλεγχοι (γρ. 21-36).

Αν δεν χρειαστεί να ρίξουμε εξαίρεση, στο τέλος του δρόμου δίνουμε:

```
   this->swap( tmp );
```

ή απλώς: `swap( tmp )`.

Όλα αυτά γίνονται υπό έλεγχο για *bad_alloc* που μπορεί να έλθει –τουλάχιστον κατ' αρχήν– από τη διαχείριση δεδομένων τύπου *string*.

Παρατηρήσεις: ▶

1. Σχετικώς με τη *RouteStopXptn*`::`*lineEmpty*: Είναι τόσο τρομερό πρόβλημα που πρέπει να ρίξουμε εξαίρεση; Λοιπόν, πρόσεξε: Η μέθοδος έχει υποχρέωση να γνωστοποιήσει το γεγονός «*με έβαλες να διαβάσω μια γραμμή με στοιχεία στάσης και η γραμμή ήταν άδεια!*» Το «*ε, καλά, δεν χάθηκε ο κόσμος*» μπορεί να το πει το πρόγραμμα που καλεί τη *readFromText()*. Στην περίπτωσή μας αυτό μπορεί να γίνει αν αλλάξουμε τον τρόπο ανάγνωσης ως εξής:

```
      do {
         try {
             RouteStop oneStop;
             oneStop.readFromText( tin );
             oneRoute.insert1RouteStop( oneStop );
         }
         catch( RouteStopXptn& x )
         {
            if ( x.errCode != RouteStopXptn::lineEmpty &&
                 x.errCode != RouteStopXptn::cannotRead )
               throw;
         }
      } while ( !tin.eof() );
```

2. Γιατί να βάλουμε "`this->swap(tmp)`" και όχι "`*this = tmp`"; Διότι η δεύτερη θα εκτελέσει, μεταξύ άλλων, την "`sName = tmp.sName`" που, κατ' αρχήν, δεν είναι ασφαλής. Η *swap()* μας καλύπτει από την εξής περίπτωση: έχουμε περάσει επιτυχώς όλους τους ελέγχους και παίρνουμε εξαίρεση *bad_alloc* όταν προσπαθούμε να αντιγράψουμε το όνομα της στάσης.◀

21.7 * Προσωρινά Αντικείμενα

Στην §19.1 είδαμε την εντολή:

```
   d1 = Date( 2008, 7, 9 );
```

ενώ στην §20.4.2 είπαμε ότι η "`s2 = "what\'s this\?"`" είναι δεκτή διότι ο μεταγλωττιστής την καταλαβαίνει σαν:

```
s2 = BString( "what\'s this\?" );
```

Αυτά είναι δύο παραδείγματα όπου καλείται δημιουργός με αρχική τιμή να δημιουργήσει ένα **προσωρινό** (temporary) αντικείμενο *χωρίς όνομα*. Και στις δύο περιπτώσεις η τιμή του προσωρινού αντικειμένου εκχωρείται στο αντικείμενο που βρίσκεται στο αριστερό μέρος της εκχώρησης. Άλλη συνηθισμένη περίπτωση είναι η δημιουργία προσωρινού αντικειμενου σε κάποιο όρισμα όταν καλείται μια συνάρτηση.

Πόσο ζει ένα προσωρινό αντικείμενο; Μέχρι το τέλος του υπολογισμού της παράστασης μέσα στην οποία εμφανίζεται. Στα παραδείγματά μας, το κάθε ένα από τα προσωρινά αντικείμενα ζει μέχρι το τέλος της εκτέλεσης της αντίστοιχης εκχώρησης.

21.8 Ο «Κανόνας των Τριών»

Από όσα είπαμε μέχρι τώρα θα πρέπει να σου φαίνεται αυτονόητος ο «Κανόνας των Τριών» (rule of three ή Law of The Big Three) που λέει ότι

- ***Αν μια κλάση χρειάζεται κάποιο από τα: Δημιουργό Αντιγραφής, Τελεστή Εκχώρησης, Καταστροφέα, πιθανότατα χρειάζεται και τα άλλα δύο.***

Η *BString* και η *Route* είναι χαρακτηριστικά παραδείγματα εφαρμογής του κανόνα.

Με την ευκαιρία, να επανέλθουμε και στο ερώτημα: *πότε απαιτείται να ορίσουμε εμείς αυτά τα τρία εργαλεία;* Η απάντηση που δώσαμε είναι:

- ***Απαιτείται να ορίσουμε Δημιουργό Αντιγραφής, Τελεστή Εκχώρησης και Καταστροφέα, όταν τα αντικείμενα της κλάσης διαχειρίζονται πόρους του συστήματος που παίρνουν δυναμικώς.***

21.9 Μια Παρένθεση για τη *renew()*

Εδώ θα πρέπει να κάνουμε μια παρένθεση για να διορθώσουμε τη *renew()* που είδαμε στην §16.12 όπως είχαμε υποσχεθεί στη δεύτερη παρατήρηση της παραγράφου. Να θυμίσουμε ότι είχαμε:

```
try
{
   T* temp( new T[nf] );
   for ( int k(0); k < ni; ++k )  temp[k] = p[k];
   delete[] p;   p = temp;
}
catch( bad_alloc& )
{
   throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed );
}
```

και στην παρατήρηση λέγαμε «*Η bad_alloc μπορεί να προέλθει όχι μόνο από τη "*`new T[nf]`*" αλλά και από τις αντιγραφές "*`temp[k] = p[k]`*". Αυτό μπορεί να γίνει αν ο T έχει αντικείμενα που χρησιμοποιούν δυναμική μνήμη.*»

Ας ξαναγράψουμε τα παραπάνω πιο προσεκτικά:

```
T* temp;
try
{  temp = new T[nf];  }
catch( bad_alloc& )
{  throw MyTmpltLibXptn( "renew",
                         MyTmpltLibXptn::allocFailed ); }
try
{
   for ( int k(0); k < ni; ++k )  temp[k] = p[k];
```

```
   }
   catch( bad_alloc& )
   {  delete[] temp;
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::allocFailed ); }
   delete[] p;   p = temp;
```

Δηλαδή, αν πάρουμε μνήμη για τα *nf* αντικείμενα τύπου *T* που δείχνει το *temp* αλλά στη συνέχεια αποτύχουμε στην προσπάθεια να πάρουμε μνήμη για να κάνουμε τις αντιγραφές, φροντίζουμε –πριν ρίξουμε την εξαίρεση– να ανακυκλώσουμε τον πίνακα που δείχνει η *temp*.

Ας πούμε τώρα ότι θέλουμε να μεγαλώσουμε έναν δυναμικό πίνακα με στοιχεία κλάσης *BString*:

- Όπως γράψαμε τον τελεστή εκχώρησης της κλάσης (§20.4), όταν βρει πρόβλημα με τη δυναμική μνήμη, ρίχνει εξαίρεση κλάσης *BStringXptn* με κωδικό σφάλματος *BStringXptn ::allocFailed*.
- Παρόμοια εξαίρεση μπορεί να ριχτεί και από τον ερήμην δημιουργό που καλείται για να δημιουργήσει όλα τα στοιχεία του πίνακα που δείχνει το *temp*.

Τι πρέπει να κάνουμε; Και στις δύο **try** θα πρέπει να βάλουμε και δεύτερη **catch** που θα πιάνει τις *BStringXptn*.

Στην πρώτη περίπτωση θα έχουμε:

```
   catch( bad_alloc& )
   {  throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed ); }
   catch( BStringXptn& x )
   {
      if ( x.errorCode == BStringXptn::allocFailed )
         throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed);
      else
         throw;
   }
```

και στη δεύτερη:

```
   catch( bad_alloc& )
   {  delete[] temp;
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::allocFailed ); }
   catch( BStringXptn& x )
   {
      delete[] temp;
      if ( x.errorCode == BStringXptn::allocFailed )
         throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed);
      else
         throw;
   }
```

Για να πάρουμε υπόψη μας όλα αυτά θα πρέπει να επιφορτώσουμε ένα δεύτερο περίγραμμα

```
template < typename T, typename TXptn >
void renew( T*& p, int ni, int nf )
```

στο οποίο θα ενσωματώσουμε τα παραπάνω αντικαθιστώντας το *BStringXptn* με *TXptn*. Σε κάθε στιγμιότυπο, στη θέση αυτής της παραμέτρου, θα πρέπει να βάζουμε κλάση εξαιρέσεων, σαν αυτές που γράφουμε εδώ, που να έχει κωδικό σφάλματος με όνομα *allocFailed*.

Τώρα πρόσεξε: *ειδικώς για τα παραδείγματα που δίνουμε σε αυτό το βιβλίο μπορούμε να απλουστεύσουμε τη renew()* αν πάρουμε υπόψη μας τα εξής:

- Αν δεν μπορέσουμε να πάρουμε δυναμική μνήμη για τον δυναμικό πίνακα που δείχνει η *temp* θα πάρουμε *bad_alloc*.

- Ο ερήμην δημιουργός δημιουργεί τα στοιχεία του δυναμικού πίνακα με τις ερήμην καθορισμένες τιμές. Δεν θα μπορέσει να δημιουργήσει κάποια από τα στοιχεία του δυναμικού πίνακα αν δεν μπορέσει να δεσμεύσει κάποιον πόρο του συστήματος. Στα παραδείγματά μας ο μόνος πόρος που διαχειριζόμαστε είναι η δυναμική μνήμη· άρα θα πάρουμε και πάλι *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.
- Ο αντιγραφικός τελεστής εκχώρησης αντιγράφει το *έγκυρο* αντικείμενο `p[k]` στο `temp[k]`. Η μόνη περίπτωση αποτυχίας της αντιγραφής έχει να κάνει με αποτυχία δέσμευσης πόρων. Για τα παραδείγματά μας αυτό σημαίνει αποτυχία δέσμευσης δυναμικής μνήμης, δηλαδή *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.

  Μπορούμε λοιπόν να χρησιμοποιούμε της εξής μορφή:

```
template< typename T >
void renew( T*& p, int ni, int nf )
{
   if ( ni < 0 || nf < ni )
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::domainError,
                            ni, nf );
// 0 <= ni <= nf
   if ( p == 0 && ni > 0 )
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::noArray );
// (0 <= ni <= nf) && (p != 0 || ni == 0)
   T* temp;
   try
   {  temp = new T[nf];  }
   catch( ... )
   {  throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed ); }
   try
   {
      for ( int k(0); k < ni; ++k )  temp[k] = p[k];
   }
   catch( ... )
   {  delete[] temp;
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::allocFailed ); }
   delete[] p;   p = temp;
} // void renew
```

όπου θεωρούμε ότι το μόνο πράγμα που μπορεί να «πάει στραβά» είναι η αποτυχία παραχώρησης δυναμικής μνήμης.

Αν χρησιμοποιήσεις τη *renew()* όταν τα αντικείμενα της *T* δεσμεύουν άλλους πόρους θα πρέπει να χρησιμοποιήσεις την πιο πολύπλοκη μορφή.[9]

Να τονίσουμε ότι όλες οι παραπάνω μορφές της *renew()* είναι ασφαλείς ως προς τις εξαιρέσεις με την εξής έννοια: αν ριχτεί εξαίρεση

- δεν έχουμε διαρροή μνήμης και
- δεν αλλάζει ο αρχικός πίνακας.

21.10 Αυτά που Μάθαμε στην Πράξη: AN ΔΕ {ΔΑ ΤΕ ΚΑ} GE SE

Στην §20.8 παραθέσαμε μερικά πράγματα που πρέπει να σκεφθούμε όταν γράφουμε μια κλάση. Τώρα μπορούμε να τα δούμε πιο συστηματικά.

Όπως θα πρέπει να έγινε φανερό, από όσα είπαμε μέχρι τώρα, για κάθε κλάση, πέρα από οποιεσδήποτε άλλες ειδικές απαιτήσεις, θα πρέπει να εξετάζουμε –και να υλοποιούμε όταν χρειάζεται– τα εξής:

- αναλλοίωτη (**AN**, class invariant),
- ερήμην δημιουργό (**ΔΕ**, default constructor),

[9] που θα πρέπει να την ολοκληρώσεις.

- δημιουργό αντιγραφής (**ΔΑ**, copy constructor),
- τελεστή εκχώρησης (**ΤΕ**, "**=**"),
- καταστροφέα αντικειμένων (**ΚΑ**, destructor),
- συναρτήσεις "*get*" (**GE**) για όλα τα χαρακτηριστικά,
- συναρτήσεις "*set*" (**SE**) για όλα τα χαρακτηριστικά.

Πάντοτε ή όταν μας χρειάζονται στην εφαρμογή-πελάτη της κλάσης; Είναι πολύ λίγες οι περιπτώσεις που η χρήση μιας κλάσης περιορίζεται σε μια μόνον εφαρμογή. Η εφαρμογή που θα χρησιμοποιήσει για πρώτη φορά μια κλάση μπορεί απλώς να σε οδηγήσει στην υλοποίηση επιπλέον μεθόδων ή/και στην αχρήστευση μερικών μεθόδων "*get*" ή/και "*set*". Αυτό το τελευταίο το είδαμε ήδη στην κλάση *Battery*: δεν γράψαμε μεν *getEnergy*, *setEnergy*, *setMaxEnergy*, *setVoltage* αλλά γράψαμε μεθόδους που –μαζί με τον δημιουργό– δίνουν μεγαλύτερη λειτουργικότητα.

Ας τα πάρουμε ένα-ένα:

Αναλλοίωτη (ΑΝ, class invariant): Η αναλλοίωτη είναι μέρος

- της απαίτησης κάθε δημιουργού (αλλιώς: ισχύει αμέσως μετά τη δημιουργία του κάθε αντικειμένου),
- της προϋπόθεσης του καταστροφέα,
- της προϋπόθεσης και της απαίτησης κάθε μεθόδου ενός αντικειμένου που έχει δηλωθεί "`public`".

Έτσι, μεταξύ άλλων είναι απαραίτητη για να γράψουμε σωστά τις μεθόδους *set* και δημιουργούς με αρχικές τιμές. Θα πρέπει να γράφεται με μαθηματική διατύπωση; Όχι κατ' ανάγκη· μπορεί να καταγραφεί και σε μορφή κειμένου. Αλλά, όσο ακριβέστερα κατάγραφεί τόσο καλύτερα χρησιμοποιείται.

Ερήμην Δημιουργός (ΔΕ, default constructor): Ο ερήμην δημιουργός καλείται για τη δημιουργία: στατικών ή δυναμικών μεταβλητών χωρίς αρχική τιμή, και στατικών ή δυναμικών πινάκων. Σχεδόν απαραίτητος. Αν δεν τον γράψεις θα γράψει έναν εννοούμενο ο μεταγλωττιστής· τα αντικείμενά σου θα ξεκινούν με μη καθορισμένη τιμή που, πιθανότατα, δεν συμμορφώνεται με την αναλλοίωτη. Συνήθως, μπορεί να δηλωθεί μαζί με τον δημιουργό με αρχικές τιμές («2 σε 1»).

Δημιουργός Αντιγραφής (ΔΑ, copy constructor): Καλείται α) όταν κατά τη δήλωση ενός αντικειμένου του δίνουμε ως αρχική τιμή την τιμή ενός άλλου της ίδιας κλάσης, β) για πέρασμα παραμέτρου τιμής σε συνάρτηση γ) επιστροφή τιμής από συνάρτηση με τύπο. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά αρκεί η αυτόματη αντιγραφή; Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Τελεστής Εκχώρησης (ΤΕ, "="): Εκχωρεί την τιμή μιας παράστασης σε μια μεταβλητή (αντικείμενο) του ίδιου τύπου. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά δεν είναι σίγουρο ότι κάνει σωστά τη δουλειά. Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Καταστροφέας (ΚΑ, destructor): Ο καταστροφέας είναι απαραίτητος για αντικείμενα που παίρνουν δυναμικά πόρους τη διάρκεια της ζωής τους, π.χ. μνήμη για έναν δυναμικό πίνακα. Δεν χρειάζεται να τον γράψεις όταν η αυτόματη καταστροφή του αντικειμένου κάνει σωστά τη δουλειά.

Συναρτήσεις *get* (GE) για όλα τα χαρακτηριστικά: Εξετάζουμε την ανάγκη για όλα τα χαρακτηριστικά αλλά δεν γράφουμε για όλα οπωσδήποτε! Π.χ. για την *BString* δεν υπάρχει λόγος να γράψουμε μια *getReserved()* που θα επιστρέφει την τιμή της *bsReserved*: αυτή η μεταβλητή είναι ένα «μυστικό» της υλοποίησης που κάνουμε. Να σημειώσουμε ότι, μερικές φορές, αυτές οι μέθοδοι δεν έχουν όνομα *getΚάτι*· παράδειγμα η *BString*::*length()* που είναι στην πραγματικότητα η *getLen* και η *BString*::*c_str()* που είναι η *getData*.

Συναρτήσεις *set* (SE) για όλα τα χαρακτηριστικά: Για όλα; Όχι! Για παράδειγμα, στη *BString* δεν είναι δυνατόν να επιτρέψουμε τη αυθαίρετη μεταβολή των *bsLen* και *bsReserved*. Οι συναρτήσεις *set* μεταβάλλουν τις τιμές μελών και οι νέες τιμές πρέπει να συμμορφώνονται με την αναλλοίωτη. Οι τιμές ορισμένων μελών μπορεί να αλλάζουν από άλλες μεθόδους σύμφωνα με τη λειτουργία του αντικειμένου.

Αν κάθε αντικείμενο έχει **πίνακα(-ες)** –ή συλλογές αντικειμένων άλλου είδους– τότε εξέτασε την ανάγκη για ύπαρξη μεθόδων για:

- **αναζήτηση**,
- **ανάκτηση**,
- **ενημέρωση** (εισαγωγή, διαγραφή, τροποποίηση)

ενός στοιχείου της (κάθε) συλλογής.

Στη *BString* μπορούμε να κάνουμε ανάκτηση και τροποποίηση με την επιφόρτωση του "`[]`". Το ότι αυτό φαίνεται πολύ βολικό και φυσικό για αυτήν την περίπτωση δεν σημαίνει ότι μπορεί να εφαρμοσθεί σε κάθε πίνακα.

Παρατηρήσεις:

1. Μερικοί προγραμματιστές θεωρούν ότι όταν έχουμε να γράψουμε συναρτήσεις *get* και *set* καλύτερα να μη γράψουμε κλάση αλλά δομή.[10] Ο δικός μας σχετικός κανόνας (§19.5) είναι λίγο διαφορετικός και βασίζεται στην αναλλοίωτη.[11] Τον υπενθυμίζουμε: **Αν μια κλάση**
   - **έχει αναλλοίωτη `true` (τα πάντα δεκτά) και**
   - **δεν έχει «μυστικά» της υλοποίησης, δηλαδή μέλη στα οποία δεν θέλουμε να δώσουμε άμεση πρόσβαση,**

   θα προτιμούμε να ομαδοποιούμε τα στοιχεία μας με μια δομή.
2. Πολλοί προγραμματιστές γράφουν καταστροφέα πάντοτε, ακόμη και αν δεν κάνει οτιδήποτε. Π.χ. θα έγραφαν: "`~Date() { }`".[12] Αυτό θα κάνουμε και εμείς από εδώ και πέρα.
3. Φαίνεται ότι ο *Δημιουργός Αντιγραφής*, ο *Τελεστής Εκχώρησης* και ο *Καταστροφέας* πάνε μαζί: ή έχεις να γράψεις και τα τρία (αν παίρνουμε πόρους του συστήματος) ή τίποτε (κανόνας των τριών). Έτσι, έχουν τα πράγματα συνήθως αλλά όχι πάντοτε. Θα δούμε και περιπτώσεις που χρειαζόμαστε μόνον κάποια από αυτά.
4. Φυσικά, η κλάση δεν περιορίζεται στα παραπάνω. Οι πιο ενδιαφέρουσες μέθοδοι θα προκύψουν από τις προδιαγραφές του προβλήματος ή τα σχέδια των εφαρμογών που θα χρησιμοποιήσουν την κλάση.

Στη συνέχεια θα εφαρμόσουμε τους παραπάνω κανόνες και θα ξαναγράψουμε μια κλάση από τα παλιά.

21.10.1 * Επιστροφή στις *string* και *BString*: Μέθοδος *reserve()*

Εδώ –και πριν δώσουμε το παράδειγμά μας– θα πρέπει να κάνουμε μια παρένθεση διότι όποιος «ερευνά τας γραφάς» γίνεται καχύποπτος: «*να μη γράψουμε getReserved() και setReserved(); Και γιατί η string έχει τις μεθόδους capacity() και reserve();*»

Η C++ δίνει –στον προγραμματιστή που χρησιμοποιεί τη *string*– τη δυνατοτητα να ελαχιστοποιήσει τις αλλαγές του δυναμικού πίνακα και –επομένως– τις αντιγραφές που

---

[10] Π.χ. στο (Lockheed Martin 2005) ο *AV Rule 123* λέει: "*The number of accessor and mutator functions **should** be minimized*", δηλαδή: *ο αριθμός των συναρτήσεων get και set πρέπει να ελαχιστοποιείται.*

[11] Να υπενθυμίσουμε ότι και στο (Lockheed Martin 2005) ο *AV Rule 66* λέει: "*A class **should** be used to model an entity that maintains an invariant.*"

[12] Συνηθέστατα: "`virtual ~Date() { }`"· θα το καταλάβεις αργότερα.

γίνονται σε κάθε αλλαγή. Πώς; Με το να καθορίσει το μεγεθος του πίνακα καλώντας τη μέθοδο *reserve()*. Δες ένα παράδειγμα:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1( "test text" );

   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;

   s1.reserve( 40 );
   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;
}
```

Αποτέλεσμα:

```
test text  9  9
test text  9  40
```

Η "`s1.capacity()`" μας επιστρέφει ως τιμή (`size_t`)[13] το μέγιστο μήκος που μπορεί να φτάσει το κείμενο που είναι αποθηκευμένο στην *s1* χωρίς να χρειαστεί να αλλάξει ο δυναμικός πίνακας. Με την "`s1.reserve(40)`" αλλάζουμε σε "`40`" (αυτό) το μέγιστο μήκος.

Η "`s1.reserve(` *n* `)`"

- Δεν θα αλλάξει την τιμή της *s1*.
- Αν *n* > *s1*.*capacity()* θα αλλάξει το μέγιστο μήκος σε *n* (ή μεγαλύτερο) εκτός αν δεν βρει αρκετή μνήμη ή *n* > *s1*.*max_size()*.
- Αν *n* < *s1*.*capacity()* δεν είναι απαραίτητο να αλλάξει κάτι.

Ας γράψουμε τις δύο αυτές μεθόδους για τη *BString*. Να παραδεχτούμε ότι η *capacity()* είναι η *getReserved()*:

```
   size_t capacity() const {  return bsReserved;  }
```

Η *reserve()* είναι πιο περίπλοκη:

```
void BString::reserve( int newRes )
{
   if ( newRes < 0 )
      throw BStringXptn( "reserve",
                         BStringXptn::domainError, newRes );
   if ( newRes > bsReserved ||
        (newRes < bsReserved/2 && newRes > bsLen) )
   {
      char* tmp;
      newRes = (newRes/bsIncr+1)*bsIncr;
      try {  tmp = new char[newRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "reserve", BStringXptn::allocFailed ); }
      bsReserved = newRes;
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
} // BString::reserve
```

Πρόσεξε ότι αλλάζουμε την τιμή του *bsReserved* αν:

- *newRes* > *bsReserved* ή
- *bsLen* < *newRes* < *bsReserved*/2

Ακόμη, για να διατηρήσουμε την αναλλοίωτή μας αλλάζουμε την τιμή της *newRes* σε "`(newRes/bsIncr+1)*bsIncr`".

---

[13] Ακριβέστερα: `string::size_type`.

Τέλος, πρόσεξε ότι αν η *reserve()* κληθεί με αρνητικό όρισμα ρίχνουμε εξαίρεση με κωδικό λάθους *BStringXptn*::*domainError* ενώ η *string*::*reserve()* ρίχνει (και) σε αυτήν την περίπτωση *bad_alloc*.[14]

Όπως βλέπεις, αυτήν τη μέθοδο δεν τη λές απλώς «*setReserved*»!

Αν προσαρμόσεις το παραπάνω προγραμματάκι στη *BString* θα πάρεις:

```
test text  9  16
test text  9  48
```

21.11 Λίστα με Απλή Σύνδεση

Στο Παράδ. 3 της §16.13 είδαμε μια *λίστα με απλή σύνδεση* (τύπος *SList*), με περιεχόμενο αντικείμενα τύπου *GrElmn*. Τώρα, θα ξαναλύσουμε το πρόβλημα γράφοντας την κλάση *SList* με εφαρμογή της συνταγής που είδαμε στην προηγούμενη παράγραφο.

Αναλλοίωτη (ΑΝ): Η λίστα είναι ένα σύνολο κόμβων που για τον καθένα –ας πούμε *aNode*– ισχύει το εξής:

είναι πρώτος κόμβος της λίστας (`*`*slHead* `==` *aNode*)[15]

ή (αποκλειστικώς, **xor**)

υπάρχει ένας και μόνον ένας άλλος κόμβος *bNode* που είναι προηγούμενος του *aNode*

(`*(`*bNode*`.`*lnNext*`)` `==` *aNode*)[16]

Ακόμη:

υπάρχει ένας και μόνον ένας κόμβος που δεν έχει επόμενο και

τον δείχνει το *slTail* (*slTail*`->`*lnNext* `==` 0)

Αυτά ισχύουν για κάθε λίστα με απλή σύνδεση.

Για τη συγκεκριμένη λίστα:

- Το περιεχόμενο της είναι ένα *σύνολο* –ας το πούμε *content*– αντικειμένων τύπου *GrElmn*:
  - Σε κάθε κόμβο της λίστας, εκτός από τον `*`*slTail*, υπάρχει ένα και μόνον ένα στοιχείο του *content*. Αφού το *content* είναι σύνολο θα έχουμε:

    (*aNode* `!=` *bNode*) ⇒ (*aNode*`.`*lnData* `!=` *bNode*`.`*lnData*)
- Η σχέση *content* – κόμβων είναι ολική συνάρτηση (αλλά όχι "*επί*", αφού υπάρχει ο φρουρός στο `*`*slTail*).

Επομένως, η κλάση μας έχει μη τετριμμένη αναλλοίωτη.

Από τα παραπάνω γίνεται σαφές και κάτι άλλο: ο τύπος *ListNode* έχει θέση *μέσα* στην *SList*.

Έτσι, θα υλοποιήσουμε την κλάση μας ως:

```
class SList
{
public:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

---

[14] Δεν εξετάζουμε την περίπτωση: *n* > *max_size()*.

[15] Ή: *slHead* `==` `&`*aNode*.

[16] Ή: *bNode*`.`*lnNext* `==` `&`*aNode*.

Όπως καταλαβαίνεις, έξω από την κλάση ο τύπος του κόμβου θα πρέπει να γράφεται "`SList::ListNode`".

Ερήμην Δημιουργός (ΔΕ): Πρέπει να γράψουμε Ερήμην Δημιουργό για να βάλουμε τον φρουρό. Θα κρατήσουμε αυτόν που γράψαμε στην §16.13 με μια αλλαγή: Δεν θα ρίχνουμε εξαίρεση *ApplicXptn* αλλά *SListXptn*:

```
SList::SList()
{
   try {  slHead = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
   slHead->lnNext = 0;  slTail = slHead;
} // SList
```

Δημιουργός Αντιγραφής (ΔΑ): Κάθε λίστα δεσμεύει πόρους –συγκεκριμένα: δυναμική μνήμη– του συστήματος. Επομένως χρειάζεται να γράψουμε εμείς τον (σωστό) δημιουργό αντιγραφής:

```
SList::SList( const SList& other )
{
   try
   {
      slHead = new ListNode;
      slTail = slHead;
      ListNode* p( other.slHead );
      while ( p != other.slTail )
      {
         slTail->lnData = p->lnData;
         slTail->lnNext = new ListNode;
         slTail = slTail->lnNext;
         p = p->lnNext;
      }
      slTail->lnNext = 0;
   }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
} // SList::SList
```

Και είναι σωστός αυτός ο δημιουργός αντιγραφής; Ας πούμε ότι έχουμε να αντιγράψουμε μια λίστα με 200 στοιχεία και παίρνουμε *bad_alloc* όταν προσπαθήσουμε να πάρουμε μνήμη για το 110το στοιχείο· με τον παραπάνω δημιουργό θα έχουμε διαρροή μνήμης που πήραμε για τα πρώτα 109 στοιχεία! Ας την ξαναγράψουμε πιο προσεκτικά:

```
SList::SList( const SList& other )
{
   try {  slHead = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
   slTail = slHead;
   ListNode* p( other.slHead );
   while ( p != other.slTail )
   {
      slTail->lnData = p->lnData;
      try {  slTail->lnNext = new ListNode;  }
      catch( bad_alloc& )
      {
         while ( slHead != slTail )
         {
            ListNode* p( slHead );
            slHead = slHead->lnNext;
            delete p;
         }
         delete slHead;
         throw SListXptn( "SList", SListXptn::allocFailed );
      }
      slTail = slTail->lnNext;
      p = p->lnNext;
```

```
   }
   slTail->lnNext = 0;
} // SList::SList
```

Όπως βλέπεις, στην "`catch`" το πρώτο που κάνουμε είναι να επιστρέψουμε τη μνήμη που πήραμε για τους κόμβους που ήδη αντιγράψαμε· μετά ρίχνουμε την εξαίρεση. Αυτός ο δημιουργός είναι σωστός και έχει *βασική εγγύηση ασφάλειας* ως προς τις εξαιρέσεις.

Καταστροφέας (ΚΑ): Αφού το κάθε αντικείμενο έχει δεσμεύσει δυναμική μνήμη πρέπει να γράψουμε καταστροφέα που θα την ανακυκλώνει:

```
SList::~SList()
{
   while ( slHead != slTail )
   {
      ListNode* p( slHead );
      slHead = slHead->lnNext;
      delete p;
   }
   delete slHead;
   slTail = slHead = 0;
} // SList::~SList
```

Εδώ βλέπεις έναν καταστροφέα που δεν έχει «μόνο μια γραμμή». Φυσικά, δεν είναι τίποτε άλλο από την *SList_deleteAll()*.

Τελεστής Εκχώρησης (ΤΕ): Αφού γράψαμε δημιουργό αντιγραφής και καταστροφέα θα πρέπει να γράψουμε και τελεστή εκχώρησης:

```
SList& SList::operator=( const SList& rhs )
{
   if ( &rhs != this )
   {
      SList tmp( rhs );
      this->swap( tmp );
   }
   return *this;
} // SList::operator=
```

όπου:

```
void SList::swap( SList& other )
{
   std::swap( slHead, other.slHead );
   std::swap( slTail, other.slTail );
} // SList::swap
```

Συναρτήσεις *get* (GE): «Καλώς εχόντων των πραγμάτων» δεν θα έπρεπε να δώσουμε συναρτήσεις "*get*". Πάντως, η *list* της STL δίνει παρόμοιες μεθόδους και αντί για *getHead()* και *getTail()* τις ονομάζει *begin()* και *end()* αντιστοίχως. Αυτό θα κάνουμε και εδώ. Τις ορίζουμε `inline`:

```
   ListNode* begin()  {  return slHead;  }
   ListNode* end()  {  return slTail;  }
```

Στο επόμενο κεφάλαιο θα δούμε και μια περίπτωση που μπορούμε να τις χρησιμοποιήσουμε αφού τις αλλάξουμε λίγο.

Συναρτήσεις *set* (SE): Φυσικά αποκλείεται να δώσουμε δικαίωμα για αλλαγή τιμών των δύο μελών αφού κάτι τέτοιο θα ήταν καταστροφικό για τη λίστα.

Διαχείριση Περιεχομένου: Θα γράψουμε μεθόδους διαχείρισης του περιεχομένου όπως περίπου δουλέψαμε και στη *Route*. Η διαχείριση του περιεχομένου θα γίνεται με βάση τον ατομικό αριθμό που τον επιλέξαμε ως κλειδί για τον *GrElmn*. Θα ξεκινήσουμε με την αντίστοιχη της *findNdx()*. Η συνάρτηση που θα γράψουμε θα επιστρέφει βέλος και όχι δείκτη αφού τώρα δεν έχουμε πίνακα:

```
SList::ListNode* SList::findPtr( int aAn ) const
{
   ListNode* fv;
   if ( aAn <= 0 )
```

```
      fv = slTail;
   else
   {
      slTail->lnData = GrElmn( aAn );
      fv = slHead;
      while ( (fv->lnData).getANumber() != aAn )  fv = fv->lnNext;
   }
   return fv;
} // SList::findPtr
```

Πρόσεξε τα εξής:

1. Η μέθοδος ψάχνει να βρει κόμβο που να έχει στο *lnData* τα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*. Αν βρει τέτοιο κόμβο επιστρέφει βέλος προς αυτόν· αλλιώς επιστρέφει βέλος προς τον φρουρό (== *slTail*).
2. Αν πάρει μη θετικό ακέραιο ως όρισμα δεν ρίχνειε εξαίρεση αλλά αναφέρει «δεν το βρήκα.»
3. Αυτή η μέθοδος είναι η αρχή της εγκατάλειψης της λειτουργίας της στοίβας που είχαμε στην αρχική λίστα. Σε μια «γνήσια» στοίβα δεν υπάρχει διαδικασία αναζήτησης· το μόνο που βλέπουμε είναι η κορυφή.

Αναζήτηση: Η *findPtr()*, όπως και η *findNdx()*, θα είναι **private**. Η αναζήτηση από ένα πρόγραμμα-πελάτη θα γίνεται με τη μέθοδο:

```
bool SList::find1Elmn( int aAn ) const
{
   return ( findPtr(aAn) != slTail );
} // SList::find1Elmn
```

που επιστρέφει "`true`" αν βρει στη λίστα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*.

Ανάκτηση: Και η ανάκτηση των δεδομένων για ένα στοιχείο γίνεται με βάση τον ατομικό αριθμό:

```
const GrElmn& SList::get1Elmn( int aAn ) const
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode == slTail )
      throw SListXptn( "get1Elmn", SListXptn::notFound, aAn );
   return ptrToNode->lnData;
} // SList::get1Elmn
```

Όπως κάναμε στην *get1RouteStop()* έτσι και εδώ, αν δεν βρούμε το στοιχείο που ζητείται, ρίχνουμε εξαίρεση. Αυτό σημαίνει ότι θα πρέπει να έχουμε εξασφαλίσει την ύπαρξη των δεδομένων πριν δώσουμε το μήνυμα *get1RouteStop()*.

Εισαγωγή: Η *push_front()* δεν μας εξασφαλίζει τη μοναδικότητα των αντικειμένων που περιέχει η λίστα. Γράφουμε λοιπόν την:

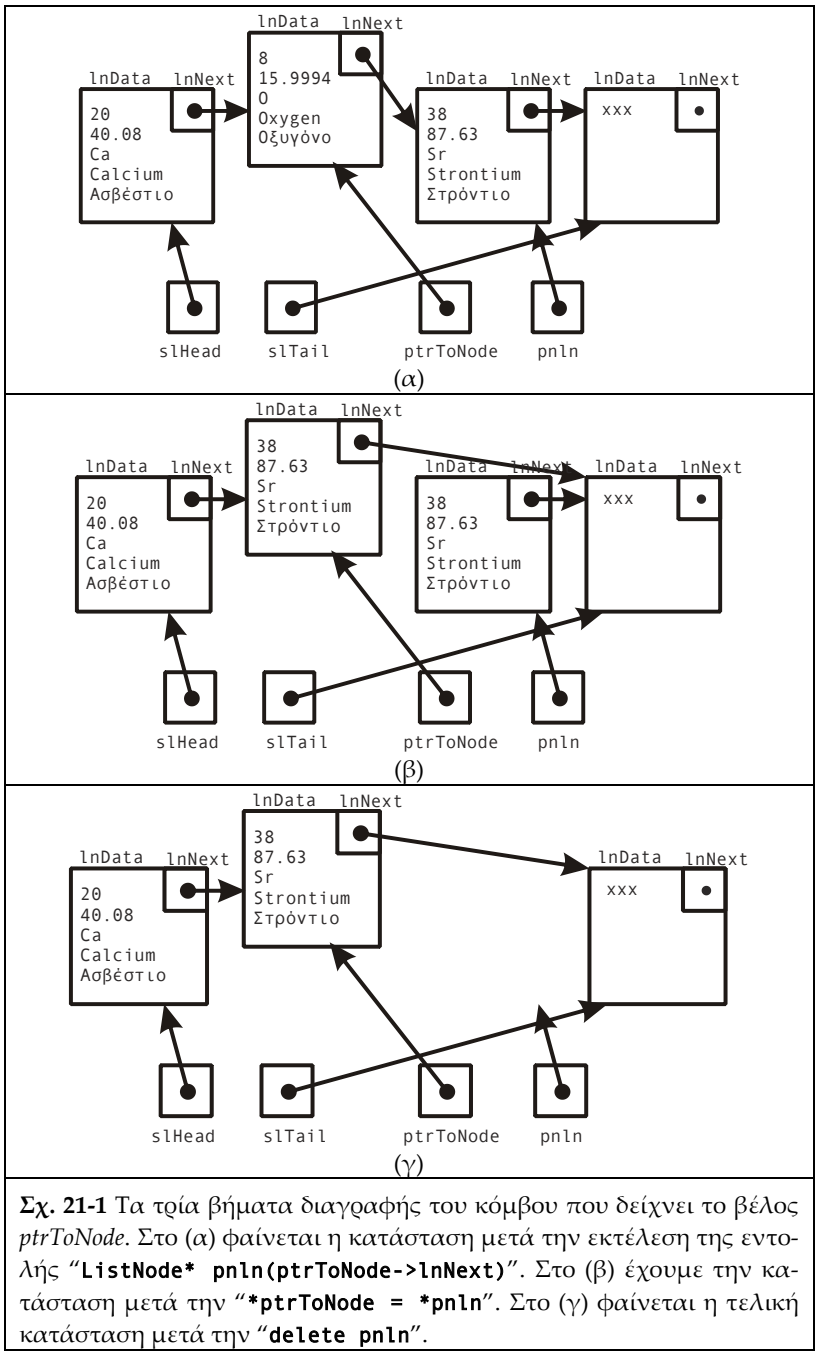
```
void SList::insert1Elmn( const GrElmn& aItem )
{
   ListNode* ptrToNode( findPtr(aItem.getANumber()) );
   if ( ptrToNode == slTail ) // δεν υπάρχει
   {
      push_front( aItem );
   }
} // SList::insert1Elmn
```

Αυτή καλεί την *push_front()* αφού έχει εξασφαλίσει ότι τα δεδομένα που θέλουμε να εισαγάγουμε δεν υπάρχουν ήδη στη λίστα. Για να αποτρέψουμε κακή χρήση της *push_front()* που θα παραβίαζε την αναλλοίωτη θα πρέπει να την «κρύψουμε» σε περιοχή **private**.

Όπως στην *insert1RouteStop()*, αν βρούμε το προς εισαγωγή στοιχείο στη λίστα δεν κάνουμε οτιδήποτε άλλο.

Διαγραφή: Εδώ έχουμε μεγάλη απόκλιση από τη λογική στοίβας: στη στοίβα η μόνη διαγραφή που μπορεί να γίνει είναι αυτή της κορυφής (*pop_front*). Εδώ, αν μας ζητηθεί να διαγράψουμε τα δεδομένα του στοιχείου με ατομικό αριθμό *aAn*, το αναζητούμε:

```
   ListNode* ptrToNode( findPtr(aAn) );
```



Σχ. 21-1 Τα τρία βήματα διαγραφής του κόμβου που δείχνει το βέλος *ptrToNode*. Στο (α) φαίνεται η κατάσταση μετά την εκτέλεση της εντολής "`ListNode* pnln(ptrToNode->lnNext)`". Στο (β) έχουμε την κατάσταση μετά την "`*ptrToNode = *pnln`". Στο (γ) φαίνεται η τελική κατάσταση μετά την "`delete pnln`".

και αν το βρούμε (*ptrToNode* `!=` *slTail*) πρέπει να διαγράψουμε τον κόμβο που δείχνει το *ptrToNode*.

Υπάρχει όμως ένα πρόβλημα: θα πρέπει να αλλάξουμε την τιμή του *lnNext* του *προηγούμενου* κόμβου που δεν ξέρουμε ποιος είναι! Θα κάνουμε κάτι άλλο:

```
ListNode* pnln( ptrToNode->lnNext );
*ptrToNode = *pnln;
delete pnln;
```

Δηλαδή: *αντιγράφουμε τον επόμενο κόμβο* (υπάρχει πάντοτε, να είναι καλά ο φρουρός) *στον στοχευόμενο και διαγράφουμε τον επόμενο*. Δες το Σχ. 21-1 όπου υποτίθεται ότι θέλουμε (α) να διαγράψουμε το "Οξυγόνο". Αντιγράφουμε (β) τα δεδομένα για το "Στρόντιο" σβήνοντας αυτά του Οξυγόνου. Αντιγράφουμε και το βέλος *lnNext* που τώρα πια δείχνει τον φρουρό. Τελικώς (γ) ανακυκλώνουμε τον κόμβο που ήταν το "Στρόντιο".

Αν ο κόμβος που ανακυκλώνουμε (*pnln*) είναι ο φρουρός θα πρέπει να αλλάξουμε την τιμή του *slTail*. Στην περίπτωση αυτή δεν υπάρχει λόγος να αντιγράψουμε ολόκληρον τον κόμβο· αρκεί ο μηδενισμός του *lnNext*:

```
void SList::erase1Elmn( int aAn )
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode != slTail ) // υπάρχει
   {
      ListNode* pnln( ptrToNode->lnNext );
      if ( pnln != slTail )
         *ptrToNode = *pnln;
      else
      {
         ptrToNode->lnNext = 0;
         slTail = ptrToNode;
      }
      delete pnln;
   }
} // SList::erase1Elmn
```

Τροποποίηση: Ο πιο απλός τρόπος να αποφύγουμε παραβίαση της αναλλοίωτης από τροποποίηση δεδομένων ενός στοιχείου είναι αυτός που χρησιμοποιήσαμε και στη *RouteStop*:

```
GrElmn tmp( lst.get1Elmn(n) );
Τροποποίησε το tmp
lst.erase1Elmn( n );
lst.insert1Elmn( tmp );
```

21.11.1 Άλλες Μέθοδοι;

Αυτά που είδαμε μέχρι εδώ ήταν αυτά που βγαίνουν από τη συνταγή μας. Υπάρχουν άλλες ανάγκες; Ναι, η φύλαξη των δεδομένων που έχουμε αποθηκευμένα στη λίστα. Στο αρχικό πρόγραμμα, για τη φύλαξη, καλούσαμε τη συνάρτηση *saveUpdateList()*. Μπορούμε να την τροποποιήσουμε και να τη χρησιμοποιήσουμε και τώρα. Η τροποποίηση έχει να κάνει με το ότι έχουμε «κρύψει» τα δυο μέλη της κλάσης:

```
void saveUpdateList( fstream& bInOut, const SList& lst )
{
   for ( const SList::ListNode* p(lst.begin());
         p != lst.end();
         p = p->lnNext )
      writeRandom( p->lnData, bInOut );
} // saveUpdateList
```

Ο τύπος της *p* (μεταβλητή-βέλος) με την οποία διασχίζουμε τη λίστα είναι "`const SList::ListNode*`". Αντί για *lst.slHead*, *lst.slTail* έχουμε `lst.begin()`, `lst.end()` αντιστοίχως.

«Καλά», θα πεις, «εδώ προσπαθούμε να κρύψουμε τις λειτουργίες της λίστας και τώρα θα πρέπει να τη διασχίσουμε σε εξωτερική συνάρτηση;! Δεν γίνεται να κάνουμε τη φύλαξη με μια μέθοδο;» Σωστή παρατήρηση! Αλλά σκέψου το εξής: Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. Εδώ, για παράδειγμα, τα αντικείμενα που αποτελούν το περιεχόμενο της λίστας φυλάγονται σε αρχείο τυχαίας πρόσβασης και έχουν ένα μέλος που καθορίζει τη θέση τους μέσα στο αρχείο. Καταλαβαίνεις λοιπόν ότι η διάσχιση της λίστας δεν μπορεί να «κρυφτεί» σε κάποια μέθοδο. Στο επόμενο κεφάλαιο θα δούμε κάποια εργαλεία για να κάνουμε τη διάσχιση χωρίς να βλέπουμε τη «λεπτομέρεια» "`p = p->lnNext`".

Παρ' όλα αυτά, περισσότερο για λόγους επίδειξης, ας κάνουμε μια προσπάθεια να γράψουμε μια μέθοδο που να κάνει αυτή τη δουλειά:

```
void SList::save( ostream& bout,
```

```
                  void (*wrProc)(const GrElmn&, ostream&) ) const
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      (*wrProc)( p->lnData, bout );
} // save
```

Εδώ έχουμε μια μέθοδο που διασχίζει τη λίστα και για κάθε κόμβο καλεί μια συνάρτηση που περνούμε ως παράμετρο. Η κλήση της γίνεται ως εξής:

```
      lst.save( bInOut, writeRandom );
```

Είναι καλύτερη αυτή η λύση; Όπως νομίζεις... Πλεονεκτεί στο ότι έκρυψε τη διάσχιση της λίστας, μειονεκτεί στο ότι περνούμε μια εξωτερική συνάρτηση ως παράμετρο σε μια μέθοδο της κλάσης.

Στη συνέχεια θα προτιμήσουμε τη *saveUpdateList()*.

21.11.2 Το Πρόγραμμα

Το πρόγραμμα τώρα θα είναι ως εξής:

```
#include <iostream>
#include <fstream>
#include <string>

#include "GrElmn.cpp"
#include "SList.cpp"

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, size_t& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

void saveUpdateList( fstream& bInOut, const SList& lst );
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      SList   lst;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            GrElmn tmp;
            if ( lst.find1Elmn(aa) )
            {
               tmp = lst.get1Elmn( aa );
```

```
               lst.erase1Elmn( aa );
            }
            else
               readRandom( tmp, bInOut, aa );
            editGrNameMM( tmp );
            lst.insert1Elmn( tmp );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
   }
   catch( GrElmnXptn& x )
   // . . .
   catch( SListXptn& x )
   // . . .
   catch( ApplicXptn& x )
   // . . .
} // main
```

Πρόσεξε ότι με την εγκατάλειψη της *elmntInList* το κομμάτι

```
ListNode* pos;
elmntInList( lst, bInOut, aa, pos );
editGrNameMM( pos->lnData );
```

αντικαταστάθηκε από το

```
GrElmn tmp;
if ( lst.find1Elmn(aa) )
{
   tmp = lst.get1Elmn( aa );
   lst.erase1Elmn( aa );
}
else
   readRandom( tmp, bInOut, aa );
editGrNameMM( tmp );
lst.insert1Elmn( tmp );
```

21.12 * Βέλος προς Μέθοδο

Επιστρέφουμε στη μέθοδο *SList*::*save* με την εξής ερώτηση: Η *GrElmn* έχει τρεις μεθόδους – τις *save()*, *display()* και *writeToTable()*– με τις οποίες φυλάγουμε την τιμή του αντικειμένου σε κάποιο αρχείο. Είναι δυνατόν, με χρήση της *SList*::*save()*, να φυλάγουμε το περιεχόμενο της λίστας με οποιονδήποτε από τους τρεις τρόπους; Ναι, αλλά με την προϋπόθεση ότι θα γράψουμε την κατάλληλη εξωτερική συνάρτηση-κέλυφος.

Ας πάρουμε τη *GrElmn*::*display()*. Αυτή έχει μια παράμετρο τύπου *ostream*&. Όταν καλούμε την *SList*::*save* πρέπει να βάλουμε ως δεύτερο όρισμα μια συνάρτηση **void** με δύο παραμέτρους: η πρώτη είναι τύπου "`const GrElmn&`" και η δεύτερη τύπου *ostream*&. Η *GrElmn*::*display()* δεν ταιριάζει! Γράφουμε λοιπόν μια εξωτερική συνάρτηση:

```
void exDisplay( const GrElmn& aElmn, ostream& outF )
{
   aElmn.display( outF );
} // exDisplay
```

Αυτή κάνει τη δουλειά που θέλουμε και ταιριάζει ως δεύτερο όρισμα όταν καλούμε την *SList*::*save()*.

Αυτή φυσικά δεν είναι μια καλή λύση: Για να έχουμε αυτό που θέλουμε θα πρέπει να γράψουμε τρεις εξωτερικές συναρτήσεις, «ενδιάμεσες» μεταξύ *SList* και *GrElmn*.

Αυτό που θέλουμε είναι να γράψουμε μια:

```
void SList::toFile( ostream& out, OutFunc outProc ) const
```

```
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      ((p->lnData).*outProc)( out );
} // toFile
```

που όταν καλείται το *outProc* θα είναι βέλος προς τη *save* ή τη *display* ή τη *writeToTable*. Θέλουμε δηλαδή να χρησιμοποιήσουμε της τεχνική *συναρτήσεων ανάκλησης* (callback, §14.4) για μεθόδους. Πώς θα ορίσουμε τον τύπο *OutFunc*; Οι δηλώσεις των τριών μεθόδων στην *GrElmn* είναι:

```
class GrElmn
{
// . . .
   void save( ostream& bout ) const;
   void display( ostream& tout ) const;
   void writeToTable( ostream& tout ) const;
// . . .
}; // GrElmn
```

Να πούμε:

```
typedef void (*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc* δείχνει μια συνάρτηση **void** και **const** με μια παράμετρο τύπου *ostream*&; Όχι! Ο ορισμός πρέπει να είναι λίγο διαφορετικός:

```
typedef void (GrElmn::*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc δείχνει μια μέθοδο της GrElmn* **void** και **const** με μια παράμετρο τύπου *ostream*&.

Βάζουμε αυτόν τον ορισμό μετά τον ορισμό της *GrElmn* αλλά πριν τον ορισμό της *SList* και οι εντολές:

```
      ofstream tout( "test.txt" );
      lst.toFile( tout, &GrElmn::display );
      tout.close();
```

θα γράψουν το περιεχόμενο της λίστας στο αρχείο test.txt.

21.13 Μετατροπές Τύπου

Στα παραδείγματά μας μέχρι τώρα είδαμε μερικές περιπτώσεις μετατροπών τύπου:

- Ο δημιουργός `BString(const char* rhs)` παίρνει ένα κείμενο σε μορφή πίνακα χαρακτήρων και μας δίνει έναν αντικείμενο κλάσης *BString* με τιμή το ίδιο κείμενο.
- Αντιστρόφως, η *c_str()* μας δίνει ένα βέλος (`const char*`) προς πίνακα χαρακτήρων (με `'\0'` στο τέλος.)
- Είδαμε τον δημιουργό της *Date* να παίρνει τον φυσικό 2008 και να τον μετατρέπει σε αντικείμενο (01.01.2008)

Επειδή τέτοιες μετατροπές είναι, γενικώς, χρήσιμες οι προγραμματιστές της C++ έχουν παγιώσει την εξής συνταγή:

- Οι μετατροπές τύπου προς μια κλάση γίνονται από δημιουργούς της κλάσης.
- Οι μετατροπές τύπου από μια κλάση προς άλλους τύπους γίνεται με ειδικές συναρτήσεις μέλη της κλάσης.

Φυσικά, κανείς δεν σε εμποδίζει να παραβείς αυτούς τους κανόνες αρκεί να το κάνεις σωστά.

21.13.1 Μετατροπή με Δημιουργό

Λέγαμε πιο πριν (§21.1.1): «*Υπάρχει περίπτωση να έχουμε δημιουργό που χρειάζεται οπωσδήποτε κάποια αρχική τιμή στη δήλωση; Δεν είναι τόσο συχνή περίπτωση αλλά υπάρχει*

πιθανότητα να χρειαστείς κάτι τέτοιο.» Ο **δημιουργός μετατροπής** (conversion constructor) είναι ακριβώς μια τέτοια περίπτωση: ένας δημιουργός με μία παράμετρο που μετατρέπει μια τιμή του τύπου της παραμέτρου σε αντικείμενο της κλάσης που ανήκει. Για την ακρίβεια: ένας δημιουργός μετατροπής *θα πρέπει να μπορεί να κληθεί με ένα όρισμα μόνον*. Αυτό σημαίνει ότι μπορεί να έχει και άλλες παραμέτρους που όμως έχουν ερήμην τιμές.

Παραδείγματα

Μετά τους διαχωρισμούς που κάναμε στην §21.1.1 έχουμε τον

```
BString( const char* rhs );
```

που καλείται με ένα και μόνον ένα όρισμα και μετατρέπει μια τιμή "`const char*`"[17] σε αντικείμενο *BString* και έτσι –αν η *a* είναι τύπου *BString*– μπορούμε να δώσουμε:

```
a = BString( "abc" );
```

Έχουμε ακόμη τον

```
Date( int yp, int mp = 1, int dp = 1 );
```

που μπορεί να κληθεί με ένα όρισμα και να μετατρέψει μια τιμή "`int`" σε αντικείμενο *Date* και έτσι –αν η *d* είναι τύπου *Date*– με την εντολή:

```
d = Date( 375 );
```

δίνουμε στην *d* τιμή "01.01.375".

Στην §15.14.1 είδαμε τη συνάρτηση

```
GrElmn GrElmn_copyFromElmn( const Elmn& a )
```

–που από μια τιμή *Elmn* μας δίνει μια τιμή *GrElmn*– και υποσχεθήκαμε ότι «*αργότερα θα μάθουμε πώς μπορούμε να δώσουμε καλύτερη λύση.*» Τη δώσαμε στην §19.6 με τον δημιουργό μετατροπής

```
GrElmn::GrElmn( const Elmn& rhs )
```

Παρατήρηση: ▶

Πρόσεξε ότι και οι «2 σε 1» δημιουργοί των *Date* και *BString* είναι δημιουργοί μετατροπής αφού είναι δυνατόν να κληθούν με ένα όρισμα.◀

Τώρα πρόσεξε το εξής: παρ' όλο που δεν έχουμε ορίσει

```
BString& operator=( const char* rhs );
```

η εκχώρηση

```
a = "abc";
```

εκτελείται χωρίς κανένα πρόβλημα. Γιατί; Διότι καλείται αυτομάτως ο δημιουργός μετατροπής και μετατρέπει το `"abc"` σε αντικείμενο κλάσης *BString*. Δηλαδή και αυτή η εντολή εκτελείται ως "`a = BString("abc")`". Αυτό είναι καλό και βολικό.

Για τη *Date* όμως τα πράγματα αλλάζουν. Αν γράψεις

```
if ( d < 2012 )
{ . . .
```

γίνεται αυτομάτως η σύγκριση "`d < Date(2012)`" που υπολογίζεται ως «η *d* είναι *πριν από* την 01.01.2012» και είναι ακριβώς αυτό που θέλουμε. Αν όμως γράψουμε "`d > 2012`" αυτό θα υπολογιστεί ως «η *d* είναι *μετά* την 01.01.2012» ενώ εμείς θέλουμε να πούμε «η *d* είναι *μετά* την 31.12.2012». Ένα τέτοιο λάθος στο πρόγραμμα είναι πολύ δύσκολο να εντοπιστεί.

---

[17] Μπορούμε να τον κάνουμε να μετατρέπει ένα τμήμα του ορίσματος. Με τον:

```
BString( const char* rhs, int n=0 );
```

δημιουργούμε αντικείμενο που έχει αρχική τιμή το κομμάτι του *rhs* από τη θέση *n* και μετά. Με την "`a = BString("abcde", 2)`" η *a* θα πάρει ως τιμή `"cde"`.

Η C++ μας δίνει τη δυνατότητα να αποφύγουμε τέτοια λάθη. Αλλάζουμε τη δήλωση του δημιουργού μετατροπής:

```
   explicit Date( int yp, int mp = 1, int dp = 1 );
```

Αυτό το "**explicit**" έχει το εξής νόημα: «*μην κάνεις αυτόματες μετατροπές. Θα κάνεις μετατροπή αν ζητείται ρητώς.*» Δηλαδή δεν είναι δεκτές οι "`d = 375`", "`d < 2012`", "`d > 2012`" και θα πρέπει να γραφούν ως "`d = Date( 375 )`", "`d < Date(2012)`", "`d > Date(2012)`". Έτσι, (υποτίθεται ότι) ο προγραμματιστής θα σκεφτεί καλύτερα αυτό που γράφει.

Παρομοίως, στη *GrElmn* καλό θα ήταν να γράψουμε:

```
   explicit GrElmn( int aan=0, float aaw=0,
                    string as="", string anm="", string agn="" );
```

ώστε να μη γίνεται ερήμην μας η μετατροπή του "`88`" σε αντικείμενο *GrElmn* που αναφέρεται στο ράδιο.

Και για τη *BString* θα μπορούσαμε να δηλώσουμε:

```
   explicit BString( const char* rhs );
```

αλλά, όπως είπαμε, στην περίπτωση αυτή η αυτόματη μεταροπή μας βολεύει. Πάντως, γενικώς, καλό θα είναι να ακολουθείς τον *Κανόνα OBJ32* του (CERT 2009):[18]

- ***Εξασφάλισε ότι οι δημιουργοί που μπορεί να κληθούν με ένα όρισμα δηλώνονται*** `explicit`.

21.13.2 Συναρτήσεις Μετατροπής

Οι συναρτήσεις μετατροπής είναι συναρτήσεις-μέλη που μετατρέπουν την τιμή ενός αντικειμένου σε έναν άλλον τύπο. Όπως είπαμε και πιο πάνω, τέτοια δουλειά κάνει και η *c_str()*· η C++ όμως μας δίνει μια πάγια μορφή για τέτοιες συναρτήσεις.

Μια συνάρτηση που κάνει την ίδια δουλειά με τη *c_str()* ορίζεται ως εξής:

```
public:
// . . .
   operator const char*() const
   {  bsData[bsLen] = '\0';  return bsData;  };
```

Ας πούμε ότι έχουμε δηλώσει:

```
      BString q( "abc" );
      char z[30];
```

Όπως ξέρουμε, μπορούμε να γράψουμε:

```
      strcpy( z, q.c_str() );
      cout << z << endl;
```

Τώρα, με τη νέα συνάρτηση, μπορούμε να γράψουμε και:

```
      strcpy( z, q );
      cout << z << endl;
```

Πρόσεξε ότι στη *strcpy* χρησιμοποιούμε το όνομα του αντικειμένου μόνο.

Πάντως, κανείς δεν μας εμποδίζει να γράψουμε:

```
public:
// . . .
   operator const char*()
   {  bsData[bsLen] = '\0';  return bsData;  };
   operator size_t() {  return bsReserved;  };
```

και να το χρησιμοποιήσουμε:

```
      size_t n( q );
```

Βέβαια, δημοκρατία έχουμε και κάνεις ό,τι θέλεις, αλλά μια τέτοια χρήση της δυνατοτητας που μας δίνεται είναι ανόητη.

---

[18] OBJ32: *Ensure that single-argument constructors are marked* "`explicit`".

Τελικώς όμως: Πόσο καλό είναι το ότι μπορούμε και γράφουμε `strcpy(z, q)` αντί για `strcpy(z, q.c_str())`; Ε, δεν είναι και καμιά μεγάλη κατάκτηση. Μπορούμε να ζήσουμε και χωρίς αυτήν τη συνάρτηση· ή *c_str()* είναι σαφώς προτιμότερη.[19]

21.14 Στατικά Μέλη Κλάσης

Ας πούμε τώρα ότι έχουμε το εξής πρόβλημα: θέλουμε να έχουμε τη δυνατότητα να ξέρουμε πόσα αντικείμενα κλάσης *Date* υπάρχουν στο πρόγραμμά μας. Τι κάνουμε; Μια ιδέα είναι να δηλώσουμε μια καθολική μεταβλητή που οι δημιουργοί θα αυξάνουν κατά 1 ενώ ο καταστροφέας θα την μειώνει κατά 1 (θα πρέπει να γράψουμε έναν καταστροφέα που θα κάνει μόνον αυτήν τη δουλειά). Αυτό όμως είναι πολύ άκομψο και καθόλου ασφαλές: μια καθολική μεταβλητή που έχει σχέση αποκλειστικά με την κλάση! Η C++ μας δίνει τη δυνατότητα να κάνουμε κάτι καλύτερο.

Μπορούμε να δηλώσουμε μια *στατική* μεταβλητή-μέλος με όνομα *objCnt* και της δίνουμε αρχική τιμή έξω από τη δήλωση της κλάσης:

```
class Date
{
public:
// ...
private:
   static size_t objCnt;
   unsigned int  dYear;
// ...
}; // Date

   size_t Date::objCnt = 0;
```

Δηλώνουμε έναν καταστροφέα:

```
   ~Date()  {  --objCnt;  };
```

και αλλάζουμε τους δημιουργούς:

```
Date::Date()
{
   dYear = 1;  dMonth = 1;  dDay = 1;
                                  // 1η Ιανουαρίου του έτους 1 μ.Χ.
   ++objCnt;
} // Date::Date

Date:: Date( int yp, int mp, int dp )
{
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   dYear = yp;
   if ( mp <= 0 || 12 < mp )
      throw DateXptn( "Date", DateXptn::monthErr, mp );
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
   ++objCnt;
} // Date::Date

Date::Date( const Date& d )
{
   dYear = d.dYear;  dMonth = d.dMonth;  dDay = d.dDay;
   ++objCnt;
```

[19] Άλλοι είναι πιο «κάθετοι». Στο (Lockheed-Martin 2005) ο *AV Rule* 177 λέει: *"User-defined conversion functions should be avoided"* δηλαδή: συναρτήσεις μετατροπής (τύπου) που ορίζονται από τον χρήστη πρέπει να αποφεύγονται!

```
} // Date::Date
```

Ακόμη, δηλώνουμε μια ανοικτή μέθοδο:

```
   static int getObjCnt()  {  return objCnt;  };
```

Αν τώρα δώσουμε:

```
   Date k1, k2, k3[5];
   cout << Date::getObjCnt() << endl;
   {
      Date s1( 1950, 2, 1 ), s2( 2000, 2, 3 ),
           s3( 1960, 3, 1 ), *c;
      cout << Date::getObjCnt() << endl;
      c = new Date [3];
      cout << Date::getObjCnt() << endl;
      delete [] c;
      cout << Date::getObjCnt() << endl;
   }
   cout << Date::getObjCnt() << endl;
```

θα πάρουμε:

```
7
10
13
10
7
```

Στην αρχή έχουμε 7 αντικείμενα κλάσης *Date*: πέντε του *k*3 και άλλα 2 από τις *k*1, *k*2. Στη συνέχεια, στη σύνθετη εντολή δηλώνουμε 3 αντικείμενα κλάσης *Date*: *s*1, *s*2, *s*3 και τα αντικείμενα γίνονται 10. Όταν πάρουμε με τον **new** άλλα 3 αντικείμενα έχουμε συνολικά 13. Στη συνέχεια ανακυκλώνουμε τη μνήμη (**delete**) που πήραμε για τον *c* και τα αντικείμενα ξαναμένουν 10. Στο τέλος της σύνθετης εντολής καλείται αυτομάτως ο κατάστροφέας και καταστρέφει τα *s*1, *s*2, *s*3· έτσι ξαναμένουμε με 7 αντικείμενα.

Ας κάνουμε τώρα μερικές παρατηρήσεις στο παράδειγμά μας:

1. Το (στατικό) μέλος *objCnt* δηλώνεται στην κλάση και παίρνει αρχική τιμή έξω από αυτήν.
2. Όπως καταλαβαίνεις από τον χειρισμό του, το *objCnt* είναι μέλος της κλάσης και όχι του κάθε αντικειμένου της.
3. Έτσι όταν αναφερόμαστε στο *objCnt* χρησιμοποιούμε επίλυση εμβέλειας ("**Date::objCnt**") και όχι επιλογή μέλους ("**.objCnt**").
4. Η μέθοδος *getObjCnt*, που χειρίζεται μόνον το μέλος *objCnt*, είναι, και αυτή, *στατική*. Και αυτή έχει σχέση με ολόκληρη την κλάση και, όπως βλέπεις αναφερόμαστε και σε αυτήν με το πρόθεμα "**Date::**".

- ***Γενικώς με το* `static` *δηλώνουμε μεταβλητές, μεθόδους ή και σταθερές που ανήκουν σε ολόκληρη την κλάση και όχι στο κάθε αντικείμενο ξεχωριστά.***

Και σταθερές; Ναι! Όπως ακριβώς δηλώσαμε το στατικό μέλος *objCnt* μπορούμε να δηλώσουμε και να δώσουμε τιμή σε σταθερά (**const**) μέλη οποιουδήποτε τύπου. Θα το δούμε στην επόμενη παράγραφο.

Στην §19.1.2 λέγαμε για τις βοηθητικές συναρτήσεις «*αργότερα ... θα τις διακοσμήσουμε και με ένα* "**static**"». Πράγματι, οι *lastDay()* και *isLeapYear()* επεξεργάζονται τα δεδομένα που τους περνούμε μέσω των παραμέτρων τους χωρίς να έχουν οποιαδήποτε κατ' ευθείαν σχέση με τα (μη στατικά) μέλη *dYear*, *dMonth*, *dDay*. Το ίδιο ισχύει και για τη *cStrLen* της *BString*. Από εδώ και πέρα, όλες αυτές θα τις δηλώνουμε "**static**"· το ίδιο θα κάνουμε και για όλες τις βοηθητικές συναρτήσεις που θα γράφουμε.

Αν ξαναγυρίσουμε στην Άσκ. 19-3 τώρα μπορούμε να δώσουμε την πιο καλή λύση για την *isValidDate()*· μια στατική συνάρτηση:

```
   static bool isValidDate( int ay, int am, int ad )
   {
      bool fv( true );
      try{  Date( ay, am, ad );  }
```

```
      catch( DateXptn& ) {  fv = false;  }
      return fv;
   } // isValidDate
```

που τη χρησιμοποιούμε κάπως έτσι:

```
   if ( Date::isValidDate(iy, im, id) ) . . .
```

21.15 «Σταθερά» Μέλη Κλάσης

Όπως είπαμε στην §19.1.1, η C++ επιτρέπει να βάλουμε το χαρακτηριστικό "`const`" σε ορισμένες μεθόδους. Τι σημαίνει; *Η μέθοδος που έχει το χαρακτηριστικό* `const` *δεν αλλάζει τις τιμές των μελών*. Για παράδειγμα, στη *BString* είχαμε δηλώσει:

```
// . . .
   BString& operator=( const BString& rhs );
   inline const char* c_str() const;
   inline size_t length() const;
   inline bool empty() const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   int compare( const BString& rhs ) const;
// . . .
```

αφού οι *c_str()*, *length()*, *empty()* και *compare()* δεν μεταβάλλουν τις τιμές μελών.

Γιατί να δηλώσουμε `const` κάποια μέθοδο;

- Για λόγους τεκμηρίωσης. Χωρίς να εξετάσουμε το τι κάνει η μέθοδος καταλαβαίνουμε ότι δεν μεταβάλλει τις τιμές των μελών.
- Για λόγους ασφάλειας. Αν προσπαθήσουμε να μεταβάλλουμε την τιμή κάποιου μέλους, ο μεταγλωττιστής θα μας επαναφέρει στην τάξη.

Μπορείς να δηλώσεις και σταθερά αντικείμενα, π.χ.:

```
   const BString aSt( "qwdfvbnh" );
```

Αν προσπαθήσεις να χειριστείς ένα τέτοιο αντικείμενο με μέθοδο που δεν είναι `const`, π.χ.:

```
   aSt.swap( bSt );
```

θα πάρεις μήνυμα λάθος από τον μεταγλωττιστή.

Υπάρχουν περιπτώσεις που θέλουμε να αλλάζουν οι τιμές ορισμένων μελών ακόμη και σε σταθερά αντικείμενα. Αν δηλώσεις τα μέλη αυτά ως "`mutable`" (μεταλλάξιμα) έχεις αυτήν τη δυνατότητα.

Μπορούμε να έχουμε μέλη-σταθερές; Ο πιο απλός τρόπος είναι αυτός που χρησιμοποιούμε στις κλάσεις εξαιρέσεων: δηλώνουμε μέσα στην κλάση έναν απαριθμητό τύπο με όσες σταθερές θέλουμε, π.χ.:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr, outOfLimits };
// . . .
}; // DateXptn
```

και χρησιμοποιούμε στο πρόγραμμα τις σταθερές *yearErr*, *monthErr*, *dayErr*, *outOfLimits* ως: `DateXptn::yearErr`, `DateXptn::monthErr`, `DateXptn::dayErr`, `DateXptn::outOfLimits`.

Πρόσεξε τώρα το παρακάτω παράδειγμα για να δεις τι άλλες δηλώσεις σταθερών μπορείς να κάνεις:

```
#include <iostream>
#include <string>

using namespace std;

class A
{
private:
```

```
// . . .
   static const int size = 10;
public:
// . . .
   static const double cx;
   static const string car;
   static const char   carc[size];
}; // A
   const double A::cx = 1.234;
   const string A::car = "123456789";
   const char  A::carc[A::size] = "123456789";

int main()
{
   cout << A::cx << "  " << A::car << "  " << A::carc << endl;
}
```

Μέσα στην κλάση μπορείς να ορίσεις *ακέραιες* σταθερές, όπως η *size*.

Σταθερές άλλων τύπων, όπως οι *cx*, *car*, *carc*, δηλώνονται μέσα στην κλάση αλλά ορίζονται έξω από αυτήν.

Όπως καταλαβαίνεις, με βάση τα παραπάνω, μπορούμε να γράψουμε:

```
class BString
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
public:
// ...
private:
   static const size_t bsIncr = 16;
   char*  bsData;
   size_t bsLen;
   size_t bsReserved;
}; // BString
```

21.16 «Σταθερά» Μέλη Αντικειμένου

Καλά τα σταθερά μέλη κλάσης έχουν νόημα· τα *σταθερά μέλη ενός μη-σταθερού αντικειμενου* τι νόημα έχουν; Ως απάντηση, θα ξαναθυμίσουμε αυτά που λέγαμε στο παράδειγμα της μπαταρίας (§19.7) «*δεν θα πρέπει να υπάρχουν μέθοδοι που να αλλάζουν τις τιμές των bVoltage και bMaxEnergy, αφού αυτά παριστάνουν σταθερά χαρακτηριστικά της μπαταρίας που της αποδίδονται με την κατασκευή* (*δημιουργία*) *της*.» Και ακόμη καλύτερο θα ήταν να δηλώναμε:

```
class Battery
{
public:
// . . .
private:
   const double bVoltage;   // volts
   const double bMaxEnergy; // joules
   double       bEnergy;    // joules
}; // Battery
```

Ωραία! Αυτό επιτρέπεται, αλλά *πώς θα δίνουμε τιμές στα bVoltage και bMaxEnergy του κάθε αντικειμένου που δηλώνουμε*; Από τη στιγμή που δημιουργούνται τα μέλη απαγορεύεται να αλλάξουμε την (όποια) τιμή τους Ο μόνος τρόπος είναι η *λίστα εκκίνησης* που ορίζει τις τιμές τους όταν δημιουργούνται. Και πάλι δηλώνουμε:

```
   Battery( double v = 12, double me = 5e6 );
```

αλλά τώρα ορίζουμε:

```
Battery::Battery( double v, double me )
   :  bVoltage( v ), bMaxEnergy( me )
{
   if ( v <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::voltageErr, v );
```

```
   if ( me <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::energyErr, me );
   bEnergy = bMaxEnergy;
} // Battery::Battery
```

Και στην κλάση για την πισίνα, οι διαστάσεις της πισίνας θα πρέπει να είναι σταθερές. Αφήνουμε ως άσκηση τις σχετικές αλλαγές.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

21-1 Όπως ξέρεις μπορούμε να δηλώσουμε:

```
   string os1( 7, '#' );
```

και η *os1* να πάρει ως τιμή το "**#######**". Εφοδίασε τη *BString* με το εργαλείο που χρειάζεται για να αποκτήσει αυτήν τη δυνατότητα.

Β Ομάδα

21-2 Γράψε μια μέθοδο *setToday()* για τη *Date* ώστε να δίνει στο αντικείμενο ως τιμή την τρέχουσα ημερομηνία από το ρολόι του υπολογιστή.

21-3 Θέλουμε ένα εργαλείο, ας το πούμε *today()*, της *Date* που θα μας δίνει την τρέχουσα ημερομηνία. Πώς θα μπορούσε να γίνει; Σκέψου το σε σχέση με το εργαλείο της προηγούμενης άσκησης αλλά και ανεξάρτητα από εκείνο.

κεφάλαιο

22

Επιφόρτωση Τελεστών

Ο στόχος μας σε αυτό το κεφάλαιο:
Να γνωρίσουμε μερικούς κανόνες για την επιφόρτωση τελεστών σε κλάσεις. Αυτοί είναι κάπως διαφορετικοί από αυτούς που ξέρουμε μέχρι τώρα· οι αλλαγές είναι απαραίτητες αφού στις κλάσεις έχουμε και κρυμμένα μέλη. Η δυνατότητα για επιφόρτωση με μεθόδους αξιοποιείται στον μέγιστο βαθμό.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να επιφορτώνεις οποιονδήποτε τελεστή σε οποιαδήποτε κλάση και να χρησιμοποιείς μερικές πάγιες τεχνικές επιφόρτωσης. Θα μπορείς να αποφασίσεις αν είναι σωστό να γίνει κάποια επιφόρτωση.

Έννοιες κλειδιά:

- *επιφόρτωση τελεστή*
- *συναρτήσεις* ***friend***
- *κλάσεις* ***friend***
- *προσεγγιστές*
- *τεχνική* ***pimpl***

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Είναι πολύ βολικό να επιφορτώνεις τελεστές, για μια κλάση που γράφεις, αντί να γράφεις συναρτήσεις. Ή, αν θέλεις, να επιφορτώνεις *και* τελεστές εκτός από τις συναρτήσεις. Ο λόγος;

- Όπως είδαμε ήδη με τις δομές, μπορείς να χρησιμοποιήσεις για τα αντικείμενα περιγράμματα συναρτήσεων (ή και κλάσεων, όπως θα δούμε στη συνέχεια). Π.χ. για να χρησιμοποιήσεις περίγραμμα συνάρτησης για αναζήτηση σε πίνακα με στοιχεία τύπου *T* συνήθως απαιτείται να έχουμε επιφορτώσει για τον *T* με τον τελεστή "`==`", ενώ για να χρησιμοποιήσεις περίγραμμα συνάρτησης για ταξινόμηση απαιτείται τουλάχιστον ο "`<`".
- Θυμάσαι τι κάνουν! Αντί να ψάχνεις μέσα στην κλάση να βρεις αν τη μέθοδο την έχεις πει *next* ή *nextDay* ή *epomenh* ή *epomeni* ή *epom* χρησιμοποιείς τον "`++`" και τελείωσες! Αυτό βέβαια με μια προϋπόθεση:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο, η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

Για πρώτη φορά μιλήσαμε για επιφόρτωση τελεστών στην §14.6 ενώ στα κεφάλαια 20 και 21 ασχοληθήκαμε διεξοδικώς με την επιφόρτωση του τελεστή εκχώρησης "`=`".

Στο κεφάλαιο αυτό, που ασχολείται με την επιφόρτωση τελεστών για κλάσεις, θα συμπληρώσουμε (και θα τροποποιήσουμε) τις οδηγίες της §14.6.4.

22.1 Επιφόρτωση Τελεστών: Τι Ξέρουμε Μέχρι Τώρα

Όπως έχουμε μάθει, μπορούμε να επιφορτώσουμε οποιονδήποτε τελεστή εκτός από τους

"`::`", "`.`", "`.*`", "`?:`"

Ο τρόπος που μάθαμε να κάνουμε την επιφόρτωση ήταν ένας: μια καθολική συνάρτηση με το όνομα "`operator@`" για τον τελεστή "`@`". Αλλά για δες πώς επιφορτώσαμε τον "`==`" στην §15.5.1:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day == rhs.day );
}; // operator==( const Date
```

και πώς στην §19.1.4:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.getYear() == rhs.getYear() &&
            lhs.getMonth() == rhs.getMonth() &&
            lhs.getDay() == rhs.getDay() );
}; // operator==( const Date
```

Τι μεσολάβησε; Στο κεφάλαιο 19 «κρύψαμε» τα μέλη της *Date* σε περιοχή **private**! Είπαμε βέβαια ότι οι "*get*" δεν αυξάνουν τον υπολογιστικό χρόνο αφού αυτές οι μέθοδοι είναι **inline**. Τι γίνεται όμως στις περιπτώσεις που θα χρειαστούμε μέλη για τα οποία δεν γράφουμε τέτοιες μεθόδους;

Και κάτι ακόμη: Ας πούμε ότι, για να προχωρούμε στην επόμενη μέρα, θέλουμε να επιφορτώσουμε τον "`++`" για τη *Date*. Θα χρειαστούμε τη *lastDay()*· αλλά και αυτήν την «κρύψαμε» σε περιοχή **private**.

Υπάρχει λοιπόν πρόβλημα με τα «κρυμμένα» μέλη και οι λύσεις –εκτός από τις "*get*"– είναι δύο:

- Να επιφορτώσουμε τους τελεστές με μεθόδους.
- Να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου στην (καθολική) συνάρτηση που επιφορτώνει κάποιον τελεστή.

Θα δούμε και τις δύο περιπτώσεις. Αλλά, όπως θα δεις στην επόμενη παράγραφο, η «επιλεκτική αποκάλυψη μυστικών» είναι μια, κατ' αρχήν τουλάχιστον, επικίνδυνη τεχνική. Θα προσπαθούμε λοιπόν να κάνουμε τις επιφορτώσεις μας με μεθόδους ή με καθολικές συναρτήσεις που χρησιμοποιούν συναρτήσεις-μέλη "*get*" που είναι **inline**.

Είδαμε ήδη την επιφόρτωση τελεστή με μέθοδο: αυτήν του "`=`" για τη *BString*. Η επιφόρτωση έγινε με μια μέθοδο:

```
BString& BString::operator=( const BString& rhs );
```

Όταν τη γράφαμε, παίρναμε υπόψη μας ότι στην εκχώρηση "`a = Π`" το *Π* γίνεται *rhs* ενώ `*this` είναι το *a*. Αν δηλαδή είχαμε υλοποίηση με καθολική συνάρτηση[1], σύμφωνα με τις αντιστοιχίες που είδαμε στην §19.1 και 19.1.1 θα είχαμε:

```
BString& operator=( BString& lhs, const BString& rhs );
```

Ο δυαδικός τελεστής "`=`" έχει τα εξής χαρακτηριστικά:

- *Έχει αριστερά το αντικείμενο* που μας ενδιαφέρει (`*this`).
- *Δεν είναι αντιμεταθετικός*, δηλαδή δεν είναι δυνατόν να γράψουμε "`Π = a`" αντί για "`a = Π`".

Οποιονδήποτε τελεστή με αυτά τα χαρακτηριστικά (άλλα παραδείγματα: "`+=`", "`[]`" κλπ) τον επιφορτώνουμε με μέθοδο.

Με μέθοδο επιφορτώνουμε κάι κάθε ενικό τελεστή.

Αν ο τελεστής είναι *δυαδικός αντιμεταθετικός* η επιφόρτωση με μέθοδο δεν είναι καλή ιδέα. Ας πάρουμε τον "==" για τη *BString*: Αν τον επιφορτώσεις με μέθοδο:

```
bool BString::operator==( const BString& rhs ) const;
```

η σύγκριση "`a == "qwerty"`" θα είναι εφικτή ενώ η "`"qwerty" == a`" είναι παράνομη αφού το `"qwerty"` δεν είναι τιμή *BString*. Εδώ η επιφόρτωση θα πρέπει να γίνει όπως ξέρουμε: με καθολική συνάρτηση.

Με καθολική συνάρτηση επιφορτώνονται και οι *δυαδικοί μη αντιμεταθετικοί* τελεστές με το αντικείμενο στο *δεξιό μέρος*.

22.2 Προβλήματα Συμβατότητας

Στην §14.6, μιλώντας για επιφόρτωση τελεστών για πρώτη φορά βάζαμε έναν βασικό κανόνα:

♦ ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***

πράγμα που λέμε με άλλα λόγια και στην εισαγωγή του παρόντος κεφαλαίου.

Η τήρηση αυτού του κανόνα είναι απαραίτητη για να διευκολυνόμαστε από τις επιφορτώσεις. Η παράβασή του αντί για διευκόλυνση προκαλεί σύγχυση και προβλήματα στον προγραμματιστή-χρήστη της κλάσης.

Το επόμενο πρόβλημα είναι αυτό που ήδη αντιμετωπίσαμε με τον τελεστή "`=`" για τη *BString* και τη μέθοδο *assign()*. Για την επιφόρτωση του "`=`" η γλώσσα επιβάλλει μέθοδο με επικεφαλίδα:

```
BString& operator=( const BString& rhs );
```

[1] *Ο μεταγλωττιστής δεν θα σου επιτρέψει* να επιφορτώσεις τον "`=`" με άλλον τρόπο

Η *assign()*, που θα πρέπει να έχει ακριβώς την ίδια λειτουργία θα πρέπει –κατά παράβαση άλλων κανόνων που έχουμε θέσει– να έχει ακριβώς την ίδια επικεφαλίδα:

```
BString& assign( const BString& rhs );
```

Και μετά; Αντιγράφουμε το σώμα της πρώτης συνάρτησης ως σώμα της δεύτερης; Όπως είπαμε, η αντιγραφή είναι προσωρινή μόνον εγγύηση ταυτόσημης λειτουργίας. Η ταυτόσημη λειτουργία μπορεί να χαθεί όταν, αργότερα, θα χρειαστεί να τροποποιήσεις τις μεθόδους.

Η μοναδική εγγύηση είναι το να γράψουμε τον κώδικα μια φορά μόνον –είτε στην `operator=()` είτε στην *assign()*– και η δεύτερη επιφόρτωση να καλεί την πρώτη. Στην §20.4.1 ορίσαμε την `operator=` και ορίσαμε –**inline**– την *assign()* ως:

```
BString& assign( const BString& rhs ) { return (*this = rhs); }
```

Έτσι, αν έχουμε δηλώσει:

```
BString a, b;
```

η εντολή:

```
b.assign( a );
```

θα μετατραπεί σε "`b = a`".

Αν προτιμούσαμε να ορίσουμε την *assign()*, θα έπρεπε να ορίσουμε –και πάλι **inline**–

```
BString& operator=( const BString& rhs )
{  return ( this->assign(rhs) );  }
```

Σε μια τέτοια περίπτωση, η εντολή:

```
b = a;
```

θα μετατραπεί σε "`b.assign( a )`".

Είναι φανερό ότι –και στις δύο περιπτώσεις– το πρόγραμμά μας δεν «πληρώνει» οποιοδήποτε κόστος κλήσης συνάρτησης.

Όπως θα δεις στη συνέχεια, με τον ίδιο τρόπο θα αντιμετωπίσουμε το ζεύγος "`+=`", *append()* –για τη *BString*– και το ζεύγος "`+=`", *forward()*, για τη *Date*.

Γενικώς λοιπόν:

- ***Αν θέλουμε δύο μέθοδοι, ας πούμε a και b, να εκτελούν την ίδια ακριβώς λειτουργία ορίζουμε τη μια από αυτές –έστω την a– και μετά ορίζουμε τη b με κλήση της a.***

Ένα άλλο πρόβλημα συμβατότητας είναι αυτό των τελεστών με σχετικό νόημα, π.χ.: "`+`" και "`+=`". Αυτοί οι δύο τελεστές επιφορτώνονται για τη *BString* ώστε να κάνουν –όπως και στη *std*`::`*string*– σύνδεση δύο ορμαθών. Η "`a + b`" μας δίνει έναν νέο ορμαθό που αποτελείται από το κείμενο του *a* και στη συνέχεια το κείμενο του *b* χωρίς να αλλάζουν οι τιμές των *a* και *b*. Η "`a += b`" επισυνάπτει στο κείμενο του *a* το κείμενο του *b* αλλάζοντας την τιμή του *a*. Όπως θα δεις, η επιφόρτωση του "`+`" γίνεται με κλήση του "`+=`"· δηλαδή ο ορισμός της πράξης γράφεται μια φορά μόνον, για τον "`+=`".

Με τους "`+`" και "`-`" το πρόβλημα μπορεί να είναι ευρύτερο. Για παράδειγμα, για τη *Date* επιφορτώνουμε τον "`+`"

`+`: *Date* × `int` ↛ *Date*

με το εξής νόημα: *η "*`d1 = d + n`*" δίνει στη d1 ως τιμή ημερομηνία μεταγενέστερη κατά n μέρες της τιμής της d.* Αν τώρα θέλουμε να επιφορτώσουμε τον "`+=`" θα πρέπει να το κάνουμε έτσι που η "`d += n`" να έχει το ίδιο νόημα με την "`d = d + n`"· οτιδήποτε άλλο θα ήταν παραπλανητικό για τον προγραμματιστή-χρήστη της κλάσης. Τα ίδια ισχύουν και για τον "`++`": το "`++d`" δεν μπορεί παρά να σημαίνει "`d = d + 1`" ή "`d += 1`". Όπως θα δεις στη συνέχεια, ορίζουμε αρχικώς τη *forward*, στη συνέχεια επιφορτώνουμε τον "`+=`" ως ταυτόσημο με αυτήν και από τον "`+=`" ορίζουμε τις επιφορτώσεις των "`+`" και "`++`". Ο μεταθεματικός "`++`" επιφορτώνεται με κλήση του προθεματικού. Η άσκ. 22-1 σου ζητάει να συνεχίσεις, με την ίδια λογική, με τους "`-`", "`--`" και "`-=`".

- ***Αν έχεις να επιφορτώσεις για μια κλάση τους "+", "+=", "++", "-", "-=", "—" όρισε έναν από αυτούς και όρισε με κλήσεις σε αυτόν τις επιφορτώσεις των άλλων.***

Παρόμοια ισχύουν και τους "`*`", "`*=`","`/`", "`/=`" κ.ο.κ.

Θα πει κάποιος: «Και τι θα γίνει αν εγώ επιφορτώνω τον κάθε τελεστή με όποιο νόημα μου έλθει;» Δεδομένου του δημοκρατικού πολιτεύματος στο οποίο ζούμε, είναι σίγουρο ότι δεν θα πάει στη φυλακή! Αυτό που δεν είναι σίγουρο είναι ότι θα βρίσκει πελάτες για να πουλάει το λογισμικό που γράφει...

22.3 Συναρτήσεις και Κλάσεις "`friend`"

Ας δούμε τώρα πώς μπορούμε να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου σε μια καθολική συνάρτηση.

Η C++ μας δίνει όμως την εξής δυνατότητα: να δηλώσουμε μέσα σε μια κλάση, ας πούμε τη *Date*, ως **φίλη** (friend) μια συνάρτηση (τελεστή), ας πούμε την `operator==()`:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs );
public:
   Date();
// . . .
```

Με ποιο αποτέλεσμα;

- ***Αν μια συνάρτηση δηλωθεί ως `friend` (φίλη) μέσα σε μια κλάση, έχει τη δυνατότητα να «βλέπει» τα εσωτερικά μέλη των αντικειμένων της κλάσης.***

Έτσι, μπορούμε να γράψουμε, όπως παλιά:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
```

Με αυτόν τον τρόπο γλυτώσαμε από κλήσεις συναρτήσεων. Για την ακρίβεια δεν τις βλέπουμε πια· διότι –όπως είπαμε– και οι κλήσεις των "*get*" εξαφανίζονταν από τον μεταγλωττιστή λόγω του "`inline`".

Για μικρές καθολικές συναρτήσεις μπορείς, αν θέλεις, να βάλεις ολόκληρο τον ορισμό μαζί με τη δήλωση:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
public:
   Date();
// . . .
```

Στην περίπτωση αυτήν η συνάρτηση (στο παράδειγμά μας: `operator==()`) ορίζεται "`inline`".

Εκτός από συναρτήσεις είναι δυνατόν να δηλώσουμε και *φίλες κλάσεις*. π.χ.:

```
class A
{
friend class B;
public:
// ...
```

Όλες οι μέθοδοι ενός αντικειμένου κλάσης *B* που δηλώνεται ως "`friend`" μέσα στην κλάση *A*, έχουν τη δυνατότητα να βλέπουν τα εσωτερικά μέλη των αντικειμένων της κλάσης *A*. Αλλά:

- ***Η φιλία δεν είναι μεταβατική.***

Δηλαδή: το ότι η κλάση *B* δηλώνεται ως "`friend`" της *A* και μια συνάρτηση ή κλάση δηλώνεται ως "`friend`" της *B* δεν σημαίνει ότι είναι και φίλη της *A*.

Όπως καταλαβαίνεις, αυτό το «άνοιγμα» των «μυστικών» ενός αντικειμένου σε συναρτήσεις και –πολύ περισσότερο– σε κλάσεις δεν είναι ακίνδυνο. Όπως σου λέει και η πείρα σου, μερικές φορές φτάνει ένα "`==`" να γίνει "`=`" για να γίνει το «κακό». Για τον λόγο αυτόν θα χρησιμοποιούμε τις φιλίες όσο γίνεται λιγότερο και με τη μεγαλύτερη δυνατή προσοχή.

Ειδικώς για τις φιλίες με κλάσεις, ο κίνδυνος μπορεί να μετριασθεί αν δηλώσεις "`friend`" όχι ολόκληρη την κλάση αλλά συγκεκριμένη (-ες) μέθοδο (-ους) της. Ας πούμε ότι έχεις:

```
class Date;

class K
{
public:
   K( int aa=5 )  {  a = aa;  };
   int f( const Date& d ) const;
   int g( const Date& d ) const;
private:
   int a;
}; // K
```

και δηλώσεις:

```
class Date
{
friend int K::f( const Date& d ) const;
//  . . .
```

μπορείς στον ορισμό της *K*`::`*f* να χρησιμοποιείς τα κρυμμένα μέλη της παραμέτρου *d* ενώ αυτό απαγορεύεται στον ορισμό της *K*`::`*g*.

Εκείνο το "`class Date;`" στην αρχή τί είναι; Διάβασε παρακάτω...

22.4 Προειδοποιητική Δήλωση

Για να μπορέσουμε να κάνουμε τη δήλωση

```
class K
{
// . . .
   int f( const Date& d ) const;
// . . .
```

στο προηγούμενο παράδειγμα θα πρέπει ο μεταγλωττιστής να ξέρει τη *Date*. Δεν χρειάζεται να την ξέρει πλήρως· αρκεί μια **προειδοποιητική δήλωση** (forward declaration)

```
   class Date;
```

που –μοιάζει με τις επικεφαλίδες συναρτήσεων που βάζουμε πριν από την κλήση τους και– λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

22.5 Ενικοί Τελεστές

Δύο από τις οδηγίες της §14.6.4 που θα αλλάξουμε είναι αυτές για την *επιφόρτωση ενικών τελεστών για κλάσεις*.

- ***Οι ενικοί τελεστές για τις κλάσεις θα επιφορτώνονται με μεθόδους.***

22.5.1 Προθεματικοί Ενικοί Τελεστές

Λέγαμε στην §14.6.4:

- «*Επιφορτώνουμε έναν προθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

  Τώρα συμπληρώνουμε, αν *C* είναι μια κλάση:

- Αν ο *T* είναι *C*`&` η επιφόρτωση μπορεί θα γίνεται με μέθοδο (χωρίς παραμέτρους):

  `Trv operator@( )`

  Αν ο *T* είναι *C* ή `const` *C*`&` η επιφόρτωση μπορεί θα γίνεται με μέθοδο:

  `Trv operator@( ) const`

Παράδειγμα

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον *προθεματικό* "`++`" για την *Date*. Τι θα κάνει; Θα προχωράει την ημερομηνία κατά μια ημέρα. Μόνον αυτό; Για να είμαστε συνεπείς με τη φιλοσοφία της C++ (C) θα πρέπει να επιστρέφει τη νέα ημερομηνία. Σύμφωνα με τις οδηγίες της §14.6.4 θα έπρεπε η επιφόρτωση να είναι της μορφής:

```
Date& operator++( Date& a )
```

αλλά σύμφωνα με την παραπάνω συμπλήρωση θα πρέπει να γίνει με μέθοδο:

```
Date& Date::operator++()
{
   if ( dDay < lastDay(dYear, dMonth) )
      ++dDay;
   else // αλλαγή μήνα
   {
      dDay = 1;
      if ( dMonth < 12 )
         ++dMonth;
      else // αλλαγή έτους
      {
         dMonth = 1;
         ++dYear;
      } // if ( dMonth...
   } // if ( dDay...
   return *this;
} // Date::operator++
```

Πως τον χρησιμοποιούμε;

```
Date d1( 2011, 3, 31 );
cout << d1 << endl;
++d1;
cout << d1 << endl;
```

Αντί για "`++d1`" μπορείς να γράψεις:

```
d1.operator++();
```

αλλά γιατί να το κάνεις;

Αν θελήσεις να επιφορτώσεις τον "`++`" με καθολική συνάρτηση (με μια παράμετρο) θα πρέπει να τη δηλώσεις "`friend`" οπωσδήποτε για να μπορέσεις να χρησιμοποιήσεις τη *lastDay*, που είναι **private**.

Παρατήρηση: ▶

Μετά αυτά που είπαμε στην §22.2, καταλαβαίνεις ότι με τον "`++`" της *Date* δεν τελειώσαμε...◀

Αργότερα θα επιφορτώσουμε και άλλους προθεματικούς ενικούς τελεστές.

22.5.2 Ενικοί Μεταθεματικοί Τελεστές

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν μεταθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a, int )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

και τώρα συμπληρώνουμε: αν *C* είναι μια κλάση

- Αν ο *T* είναι *C*`&` η επιφόρτωση μπορεί να γίνει με μέθοδο της *C*:

  `Trv operator@( int )`

  Αν ο *T* είναι *C* ή `const` *C*`&` η επιφόρτωση μπορεί να γίνει με μέθοδο:

  `Trv operator@( int ) const`

Παράδειγμα

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" για τη *Date*. Ο *T* σίγουρα είναι ο *Date*`&`. Επομένως συζητούμε για την περίπτωση:

`Trv operator++( int )`

Ο *Trv* ποιος θα είναι; Ο *Date* ή ο *Date*`&`; Ο τελεστής αυτός αλλάζει την τιμή του αντικειμένου αλλά επιστρέφει την προηγούμενη τιμή του. Αυτή θα πρέπει να έχει κρατηθεί σε μια *τοπική μεταβλητή της συνάρτησης*. Άρα δεν είναι δυνατόν να επιστρέψει τιμή τύπου *Date*`&`. Επομένως θα έχουμε:

`Date operator++( int )`

Κατά τα άλλα, η υλοποίηση είναι πολύ απλή αν χρησιμοποιήσουμε τον προθεματικό "`++`":

```
Date Date::operator++( int )
{
   Date sv( *this );
   ++(*this);
   return sv;
} // Date::operator++(int)
```

22.6 Μη-Αντιμεταθετικοί Δυαδικοί Τελεστές

Λέμε ότι ένας δυαδικός τελεστής είναι μη-αντιμεταθετικός όταν διαχείριζόμαστε με διαφορετικό τρόπο τα δύο μέρη του. Συνήθως αυτό συνοδεύεται και από διαφορά τύπου (τουλάχιστον κατά ένα "`const`").

22.6.1 Αντικείμενο Αριστερά

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν δυαδικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( Tl a, Tr b )`

  όπου Trv είναι ο τύπος του αποτελέσματος, Tl ο τύπος της πρώτης παραμέτρου και Tr ο τύπος της δεύτερης παραμέτρου.»

  Τώρα συμπληρώνουμε: Αν έχουμε το αντικείμενο αριστερά, αν δηλαδή ο *Tl* είναι *C*`&` ή *C* ή `const` *C*`&` η επιφόρτωση γίνεται με μέθοδο της κλάσης *C* και πιο συγκεκριμένα:

- Αν ο *Tl* είναι *C*`&` με μέθοδο με μια παράμετρο:

  `Trv operator@( Tr b )`

  Αν ο *Tl* είναι *C* ή `const` *C*`&` με μέθοδο:

  `Trv operator@( Tr b ) const`

Ως παράδειγμα θα επιφορτώσουμε τον τελεστή "`[]`" για τη *BString* και τον "`+=`" για τη *BString* και για τη *Date*.

22.6.1.1 Ο Τελεστής "`[]`" για τη *BString*

Μετά τη δήλωση:

```
string q( "qazwsx" );
```

η εντολή:

```
cout << q << "  " << q[2] << endl;
```

δίνει:

```
qazwsx  z
```

Ακόμη οι:[2]

```
q[2] = 'a';
cout << q << endl;
```

δίνουν:

```
qaawsx
```

Για να είναι αυτό δυνατόν στη *BString*, ο τελεστής "`[]`" θα πρέπει να επιφορτώνεται ως:

```
char& operator[]( BString a, int b )
```

Πρόσεξε ότι επιστρέφει τιμή τύπου "`char&`" (τιμή-*l*) και όχι "`char`". Σύμφωνα με αυτά που είπαμε παραπάνω θα πρέπει να τον επιφορτώσουμε ως μέθοδο

```
char& operator[]( int pos ) const {  return bsData[pos];  }
```

Μετά από αυτό, αν έχουμε ορίσει:

```
   BString q( "qazwsx" );
```

οι εντολές:

```
    cout << q.c_str() << "  " << q[2] << endl;
    q[2] = 'a';
    cout << q.c_str() << endl;
```

θα δώσουν:

```
qaawsx
qaawsx  a
```

Το ίδιο αποτέλεσμα παίρνεις και με τις:

```
    cout << q.c_str() << "  " << q.operator[](2) << endl;
    q.operator[](2) = 'b';
    cout << q.c_str() << endl;
```

22.6.1.2 Ο Τελεστής "+=" για τη *BString*

Έστω ότι θέλουμε να χρησιμοποιήσουμε τον τελεστή "`+=`" για να συνδέσουμε στο τέλος ορμαθού έναν άλλο ορμαθό.[3] Η επιφόρτωση θα είναι της μορφής:

```
BString& operator+=( BString& lhs, const BString& rhs )
```

ώστε στο πρόγραμμά μας να μπορούμε να γράφουμε:

```
s0 += s1;
```

Σύμφωνα με αυτά που είπαμε θα πρέπει να τον επιφορτώσουμε με μια μέθοδο:

```
BString& BString::operator+=( const BString& rhs )
```

---

[2] Να υπενθυμίσουμε: για την *std*`::`*string* η διαφορά των *at* και "`operator[]`" είναι ότι
- η *at* κάνει έλεγχο του δείκτη ενώ
- ο "`operator[]`" τον αφήνει στον προγραμματιστή.

[3] Αν s_1, s_2 ορμαθοί, τότε η s_1 `+=` s_2 ισοδυναμεί με $s_1 \leftarrow s_1 \frown s_2$, όπου "$\frown$" είναι ο τελεστής σύνδεσης ακολουθιών.

Το πρόγραμμα δεν είναι απλό. Ας δούμε τι περιπτώσεις υπάρχουν (είμαστε «μέσα στο *s0*» και *rhs* είναι το *s1*):

- Να μην χρειαστούμε επιπλέον μνήμη. Πότε συμβαίνει αυτό; Όταν *bsLen* + *rhs*.*bsLen* + 1 ≤ *bsReserved*. Στην περίπτωση αυτή όλα είναι απλά:

```
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
```

- Όταν *bsLen* + *rhs*.*bsLen* + 1 > *bsReserved* χρειαζόμαστε μνήμη.

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc )
      {  throw BStringXptn( "operator+=",
                            BStringXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις, αν δεν καταφέρουμε να πάρουμε μνήμη –και ριχτεί εξαίρεση– δεν θα αλλάξει η τιμή του αντικειμένου (ασφάλεια προς τις εξαιρέσεις επιπέδου ισχυρής εγγύησης).

Να και ένα παράδειγμα χρήσης:

```
      BString s2( "qazwsxedcrf" ), s3( "res" ), s4( "qwerty" );

      cout << s2.c_str() << "  " << s2.length() << endl
           << s3.c_str() << "  " << s3.length() << endl
           << s4.c_str() << "  " << s4.length() << endl;

      s4 += s3;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
      s4 += s2;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
```

Αποτέλεσμα:

```
qazwsxedcrf  11
res  3
qwerty  6
qwertyres  qazwsxedcrf  res
qwertyresqazwsxedcrf  qazwsxedcrf  res
```

Μην αμφιβάλλεις: αν γράψεις "`s4.operator+=(s3)`" αντί για "`s4 += s3`" και "`s4.operator+=(s2)`" αντί για "`s4 += s2`" θα πάρεις τα ίδια αποτελέσματα ακριβώς.

Στη *string* υπάρχει και η μέθοδος *append* που κάνει τα ίδια ακριβώς με τον "`+=`". Για αυτήν τη μέθοδο θα πρέπει να παραβιάσουμε τους κανόνες μας (σύμφωνα με τους οποίους θα πρέπει να την κάνουμε "`void`"): την υλοποιούμε αντιγράφοντας την υλοποίηση του τελεστή και βάζοντας μόνον στην επικεφαλίδα "`append`" όπου "`operator+=`":

```
BString& BString::append( const BString& rhs )
// ΤΑ ΥΠΟΛΟΙΠΑ ΙΔΙΑ ΜΕ ΤΟΝ operator+=
```

ή, ακόμη καλύτερα, με τον ορισμό (**inline**)

```
BString& append( const BString& rhs ) { return (*this += rhs); }
```

22.6.1.3 Ο Τελεστής "+=" για τη *Date*

Έστω ότι θέλουμε να γράψουμε μια μέθοδο για την κλάση *Date*, ας την πούμε *forword*, που θα προχωράει μια ημερομηνία. Δηλαδή, αν έχουμε δηλώσει:

```
Date d( 2007, 10, 15 );
```

τότε η:

```
d.forward( 5 );
```

θα προχωρήσει τη *d* κατά 5 ημέρες και η τιμή της θα γίνει 20.10.2007.

Σύμφωνα με τους κανόνες μας, θα πρέπει να τη δούμε ως "`forward(d, 5)`" όπου μεταβάλλεται η τιμή του πρώτου ορίσματος και να υλοποιήσουμε ως `void`.

Για την ίδια λειτουργία θα επιφορτώσουμε και τον τελεστή "`+=`", πράγμα πολύ φυσικό αφού το ίδιο κάνει και για τους ακέραιους. Εδώ όμως τα πράγματα αλλάζουν: Η "`d += 5`" θα πρέπει να είναι πράξη που θα επιστρέφει τη νέα τιμή της *d*. Επειδή γίνεται το ίδιο με τους αριθμούς; Ναι, αλλά και για τον εξής λόγο: έχουμε ήδη επιφορτώσει τον "`++`". Είναι επιθυμητό οι "`++d`" και "`d += 1`" να έχουν το ίδιο αποτέλεσμα. Η επιφόρτωση θα είναι της μορφής:

```
Date& operator+=( Date a, int dd )
```

Θα την υλοποιήσουμε ως μέθοδο

```
Date& operator+=( long int dd );
```

Η *forword()* θα πρέπει να συμπεριφέρεται παρομοίως. Θα έχουμε λοιπόν:

```
Date& forward( long int dd );
```

Ο πιο απλός τρόπος για να υλοποιήσουμε τις μεθόδους είναι να χρησιμοποιήσουμε τις συναρτήσεις διαχείρισης χρόνου της C. Στην επόμενη υποπαράγραφο θα δούμε εν συντομία αυτά που μας ενδιαφέρουν.

22.6.1.4 * Ο Χρόνος στη C

Για να χρησιμοποιήσεις τις συναρτήσεις διαχείρισης χρόνου σε ένα πρόγραμμα θα πρέπει να περιλάβεις το ctime:

```
#include <ctime>
```

Η πρώτη συνάρτηση που θα δούμε είναι η:

```
time_t time( time_t* timer );
```

που μας δίνει τα δευτερόλεπτα που έχουν περάσει από την 01.01.1970, ώρα 00:00:00 GMT[4]. Ο τύπος *time_t* είναι μετονομασία του `long int`.

Μπορείς να πάρεις τον τρέχοντα χρόνο στο πρόγραμμά σου δηλώνοντας:

```
time_t t1;
```

και δίνοντας: "`t1 = time(0)`" ή "`time(&t1)`".

Ελάχιστη επιτρεπόμενη τιμή της *t1* είναι το 0, που αντιστοιχεί σε 01.01.1970, ώρα 00:00:00 GMT (ή 02:00:00 ώρα Ελλάδας). Μέγιστη επιτρεπόμενη τιμή είναι η `LONG_MAX` που αντιστοιχεί σε 19.01.2038, ώρα 03:14:07 GMT (ή 05:14:07 ώρα Ελλάδας).

Η συνάρτηση

```
tm* localtime( const time_t* timer );
```

μετατρέπει μια τιμή τύπου `time_t` σε τιμή τύπου:

```
struct tm
{
   int tm_sec;     // seconds
   int tm_min;     // minutes
```

[4] GMT: Greenwich Mean Time. Η «ώρα Ελλάδος» είναι δύο ώρες μπροστά (02:00:00)

```
   int tm_hour;    // hour (0 - 23)
   int tm_mday;    // day of month (1 - 31)
   int tm_mon;     // month (0 - 11)
   int tm_year;    // year (Έτος - 1900)
   int tm_wday;    // μέρα εβδομάδας (0 – 6, 0 για Κυριακή)
   int tm_yday;    // μέρα του έτους (0 -365)
   int tm_isdst;   // για την αλλαγή θερινής ώρας.
}; // struct tm
```

Η αντίστροφη λειτουργία γίνεται από τη

```
time_t mktime( tm* t );
```

Πώς μπορούμε να χρησιμοποιήσουμε τα παραπάνω μαζί με τη *Date*; Ας πούμε ότι έχουμε μια τιμή:

```
   Date d1;
```

και θέλουμε να βάλουμε στην *t*1 την τιμή τύπου `time_t` που ισοδυναμεί με *d1*. Δηλώνουμε μια:

```
   tm tm1;
```

και βάζουμε:

```
   tm1.tm_sec = 0;
   tm1.tm_min = 0;
   tm1.tm_hour = 12;   // μεσημέρι
   tm1.tm_mday = d1.dDay;
   tm1.tm_mon = d1.dMonth - 1;
   tm1.tm_year = d1.dYear - 1900;
   tm1.tm_wday = 0;
   tm1.tm_yday = 0;
   tm1.tm_isdst = 0;
```

Μετά χρησιμοποιούμε τη *mktime()*:

```
   t1 = mktime( &tm1 );
```

Αντιστρόφως, αν έχουμε την τιμή της *t1* παίρνουμε την αντίστοιχή της *d1* ως εξής:

```
   tm1 = *localtime( &t1 );
```

και στη συνέχεια:

```
   d1.dYear = tm1.tm_year + 1900;
   d2.dMonth = tm1.tm_month + 1;
   d2.dDay = tm1.tm_day;
```

22.6.1.5 * Υλοποίηση της *forward()*

Με βάση τα παραπάνω υλοποιούμε τη

```
   Date& Date::forward( int dd );
```

ως εξής:

- Δηλώνουμε τις:

```
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
```

- Μετατρέπουμε τις *dd* ημέρες σε 24×60×60×*dd* (= 86400×*dd*) δευτερόλεπτα:

```
   dd *= 86400;      // ημέρες σε δευτερόλεπτα
```

- Προσθέτουμε τη *dd* στην *currT*:

```
   currT += dd;
```

- Τέλος, μετατρέπουμε την τιμή *time_t* που προκύπτει στην «ισοδύναμη» τιμή *Date*:

```
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
```

Να ολόκληρη η μέθοδος:

```
Date& Date::forward( long int dd )
```

```
{
   const unsigned long secsPerDay = 86400;
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   dd *= secsPerDay;     // ημέρες σε δευτερόλεπτα
   if ( dd < 0 )
   {
      if ( currT < (-dd) )  // currT + dd < 0
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   else // dd >= 0
   {
      if ( dd > LONG_MAX - currT )  // currT + dd > LONG_MAX
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   currT += dd;
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
   return *this;
} // Date::forward
```

Στην **if** ελέγχουμε κατά πόσον η τιμή που προκύπτει είναι μέσα στα όρια.

Ο τελεστής "**+=**" υλοποιείται με τον ορισμό (**inline**)

```
   Date& operator+=( long int dd ) {  return forward( dd );  }
```

22.6.1.6 * Ο Τελεστής "++" της *Date* (ξανά)

Τώρα, για να είμαστε συνεπείς με αυτά που λέμε στην §22.2, θα πρέπει να ξαναορίσουμε την επιφόρτωση του "++" ως εξής:

```
   Date& operator++() {  return forward( 1 );  }
```

Καλό πράγμα η συνέπεια και η συμβατότητα, αλλά η προηγούμενη μορφή της επιφόρτωσης δεν ήταν ταχύτερη; Ναι, ήταν!

22.6.2 Ο Τελεστής "()" και η Χρήση του

Μια ιδιαίτερη περίπτωση δυαδικού τελεστή με το αντικείμενο αριστερά είναι ο τελεστής κλήσης συνάρτησης "**()**". Με την επιφόρτωση αυτού του τελεστή για μια κλάση *C*:

```
struct C
{
// . . .
   double operator()( double x ) const;
// . . .
};
```

αν έχεις δηλώσει

```
   C f;
```

μπορείς να γράφεις:

```
   if ( f(a)*f(b) < 0 ) m = f(a) + c/2;
// . . .
```

Δηλαδή, χρησιμοποιούμε το αντικείμενο *f* σαν συνάρτηση. Για τον λόγο αυτόν τέτοια αντικείμενα ονομάζονται **συναρτησιακά αντικείμενα** ή **συναρτησοειδή** (function objects, functors).

Όπως καταλαβαίνεις, από το παραπάνω παράδειγμα, αριστερά των παρενθέσεων βρίσκεται το όνομα του αντικειμένου αλλά στα δεξιά –και για την ακρίβεια μέσα στις παρεν-

θέσεις (όπως στην περίπτωση του "`[]`")– πρέπει να υπάρχει λίστα παραμέτρων που στο παράδειγμά μας είναι μονομελής αλλά, γενικώς, μπορεί να είναι πολυμελής ή και κενή.

Πού θα μπορούσε να είναι χρήσιμη αυτή η επιφόρτωση; Η πιο τυπική χρήση: στην αποφυγή της τεχνικής *συναρτήσεων ανάκλησης*. Ας δούμε ένα

Παράδειγμα

Ξαναγυρνούμε στη συνάρτηση *bisection()* στην §14.3. Η χρήση βέλους προς συνάρτηση δεν είναι και τόσο βολική. Να το αλλάξουμε λοιπόν, αλλά πώς;

Στο Παράδ. 2 της §14.7.1 λύσαμε ένα πρόβλημα που είχαμε με «μη βολικές» παραμέτρους καταφεύγοντας σε παραμέτρους περιγράμματος. Μήπως μπορούμε να κάνουμε το ίδιο και εδώ; Δηλαδή: να κάνουμε

- τη *bisection()* περίγραμμα συνάρτησης και
- την *f* της "*f*(*x*) = 0" παράμετρο του περιγράμματος.

Καλό θα ήταν, αλλά *το περίγραμμα δεν δέχεται παράμετρο-συνάρτηση*! Δεχεται όμως παράμετρο κλάση και το πρόβλημά μας λύνεται αν κανονίσουμε να βάζουμε κλάση που τα αντικείμενά της είναι συναρτησοειδή. Γράφουμε λοιπόν:

```
template < class Func >
void bisection( double a, double b, double epsilon,
                double& root, int& errCode )
{
   Func   f;
   double m;

   if ( f(a)*f(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b - a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                             else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Όπως βλέπεις, ένα αντικείμενο κλάσης *Func*, όπως το *f*, θα πρέπει να μπορεί να χρησιμοποιηθεί ως συνάρτηση, για να έχουν νόημα τα ""`f(a)`", "`f(b)`", "`f(m)`".

Τώρα, αντί για συναρτήσεις θα γράψουμε κλάσεις:

```
class qF
{
public:
   double operator()( double x ) const
   {  return ( x - log(x) - 2 );  }
}; // qF


class cosF
{
public:
   double operator()( double x ) const
   {  return cos( x );  }
}; // cosF
```

Και να πώς θα γίνει το πρόγραμμα:

```
      bisection< qF >( 0.1, 1.0, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
      bisection< cosF >( 0.0, pi, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
```

```
                    else  cout << " Ρίζα = " << riza << endl;
```

Προσοχή: Ο μεταγλωττιστής θα δημιουργήσει δύο *bisection()* –δύο στιγμιότυπα του περιγράμματος–

- ένα για την εξίσωση $q(x) = 0$ και
- ένα για την εξίσωση $συν(x) = 0$.

Οι κλάσεις των συναρτησοειδών μπορεί να έχουν και αντίστοιχες κλάσεις εξαιρέσεων. Έτσι, μπορούμε να γράψουμε μια *qFXptn* και την *qF*· ακριβέστερα:

```
struct qFXptn
{
   enum { domain };
   char   funcName[100];
   int    errorCode;
   double errDblVal;
   qFXptn( const char* mn, int ec, double dv=0.0 )
      : errorCode( ec ), errDblVal( dv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // qFXptn

class qF
{
public:
   double operator()( double x ) const
   {
      if ( x <= 0 )
         throw qFXptn( "operator()", qFXptn::domain, x );
      return ( x - log(x) - 2 );
   }
}; // qF
```

Αν στο πρόγραμμά μας βάλουμε:

```
   try {
   // . . .
      bisection< qF >( -1, 1.0, 1e-5, riza, errCode );
   // . . .
   }
   catch( qFXptn& x )
   {
      cout << "domain error (" << x.errDblVal << ")" << endl;
   }
```

θα μας δώσει:

```
domain error (-1)
```

Η συνάρτηση

```
bool mult7( int a )  {  return ( (a%7) == 0 );  }
```

μας επιστρέφει **true** αν το όρισμα είναι διαιρετό δια 7, αλλιώς **false**. Αυτό είναι χαρακτηριστικό παράδειγμα **κατηγορήματος** (predicate).

Δες όμως και το πρόγραμμα

```
#include <iostream>
using namespace std;

struct Mult7
{
   bool operator()( int a )  {  return ( (a%7) == 0 );  }
};

int main()
{
   Mult7 aMult7;

   cout << boolalpha;
```

```
   cout << aMult7(35) << "  " << aMult7(37) << endl;
}
```

που θα δώσει:

```
true  false
```

Και αυτό το συναρτησοειδές, που η συνάρτησή του είναι τύπου **bool**, είναι ένα *κατηγόρημα*.

22.6.2.1 * Μέλος – Περίγραμμα Συνάρτησης

Στην §21.12 είδαμε τα βέλη προς μεθόδους ως εργαλείο για τη χρήση της τεχνικής των συναρτήσεων ανάκλησης. Μήπως μπορούμε να χρησιμοποιήσουμε και στην περίπτωση αυτήν αυτά που μάθαμε παραπάνω για να απαλλαγούμε από τα βέλη προς μεθόδους; Για να κάνουμε κάτι τέτοιο θα πρέπει η *SList*::*toFile()* να γίνει περίγραμμα. Μπορεί να γίνει αυτό; Ναι, *η C++ σου επιτρέπει να βάλεις σε μια κλάση μέλη που να είναι περιγράμματα*. Έτσι στην περίπτωσή μας μπορείς να βάλεις:

```
class SList
{
public:
// . . .
   template < class OutFunc >
   void toFile( ostream& out ) const
   {
      OutFunc toStream;
      for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
         toStream( out, p->lnData );
   } // toFile
// . . .
};
```

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις τρεις κλάσεις, μια για κάθε μια από τις *save()*, *display()* και *writeToTable()* της *GrElmn*:

```
struct SaveGE
{
   void operator()( ostream& bout, const GrElmn& geObj )
   {  geObj.save( bout );  }
}; // SaveGE


struct DisplayGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.display( tout );  }
}; // DisplayGE


struct WriteToTableGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.writeToTable( tout );  }
}; // writeToTableGE
```

Όταν ο μεταγλωττιστής βρει στο πρόγραμμά μας την

```
      lst.toFile< DisplayGE >( tout );
```

θα εφοδιάσει αυτομάτως την *SList* με την αντίστοιχη μέθοδο.

22.6.3 Αντικείμενο Δεξιά

Εδώ αναφερόμαστε σε επιφορτώσεις της μορφής

```
Trv operator@( Tl a, Tr b )
```

Τέτοιοι τελεστές:

- Αν ο *Tl* είναι κλάση, μπορεί να επιφορτωθούν ως μέθοδοί της, όπως ήδη είδαμε.
- Αν ο *Tr* είναι κλάση, δεν είναι δυνατόν να επιφορτωθούν ως μέθοδοι αφού, όπως έχουμε πει, το αντικείμενο που μας ενδιαφέρει παίζει –στη μέθοδο– ρόλο πρώτης παραμέτρου. Μπορούμε να τους επιφορτώσουμε ως *καθολικές συναρτήσεις*, όπως ακριβώς μάθαμε στην §14.6.4.

Για παράδειγμα, στην §15.5 επιφορτώσαμε τον "`<<`" για τη *Date* ως:

```
ostream& operator<<( ostream& tout, const Date& rhs )
```

Αυτός:

- Θα μπορούσε να επιφορτωθεί ως μέθοδος στην *ostream*, αλλά
- Για τη *Date* θα επιφορτωθεί ως καθολική συνάρτηση.

Η επιφόρτωση που κάναμε στην §19.1.4 είναι μια χαρά:

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << rhs.getDay() << '.' << rhs.getMonth() << '.'
               << rhs.getYear();
} // operator<<( ostream& tout, const Date
```

Να τη δηλώσουμε `friend`; Δεν χρειάζεται, αφού οι τρεις "*get*" είναι `inline`.

Παρομοίως γίνεται και η επιφόρτωση του "`<<`" για τη *BString*:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   return tout << rhs.c_str();
} // operator<<( ofstream, BString
```

Αυτός είναι ένας απλός αλλά κάπως επιπόλαιος τρόπος. Η εκτύπωση μέσω της *c_str()* (και γενικώς η χρήση της) τελειώνει στον πρώτο `'\0'` που τυχόν θα βρεθεί. Στην §20.1 επικαλεστήκαμε ως λόγο επιλογής της συγκεκριμένης παράστασης για τα αντικείμενα *BString* το ότι μας επιτρέπει «*να έχουμε και χαρακτήρες* `'\0'` *στην τιμή που αποθηκεύουμε.*» Αν λοιπόν το *bsData* δείχνει κάτι σαν `"ab\0cd"` θα πρέπει η εκτύπωση να τελειώνει στο `'d'` και όχι στο `'b'`. Να πώς μπορεί να γίνει αυτό:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.bsLen; ++k )  tout << rhs.bsData[k];
   return tout;
} // operator<<( ofstream, BString
```

Αν τη γράψεις έτσι θα πρέπει να δηλώσεις και:

```
class BString
{
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
// . . .
```

Αφού όμως και η *length()* και ο `operator[]` είναι μέθοδοι `inline` μπορούμε να γράψουμε χωρίς επιπλέον κόστος:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

που δεν χρειάζεται να δηλωθεί ως `friend`.

22.7 Αντιμεταθετικοί Τελεστές

Ένας δυαδικός *αντιμεταθετικός τελεστές* είναι ένας τελεστής που δέχεται δύο ορίσματα και έχει την ίδια δράση αν αντιμεταθέσουμε το αριστερό με το δεξιό μέρος· για παράδειγμα ο "`+`", οι τελεστές σύγκρισης κλπ. Στην §22.1 είδαμε με ένα παράδειγμα γιατί δεν μπορούμε

να επιφορτώσουμε τον "==" για τη *BString* με μέθοδο. Παρόμοιο πρόβλημα υπάρχει και με τον "+" και άλλους παρόμοιους τελεστές για την ίδια κλάση (και όχι μόνο).

Οι τελεστές αυτοί υλοποιούνται με *καθολικές συναρτήσεις* που μπορεί να δηλωθούν **friend** για να γίνουν ταχύτερες.

Έτσι, στη *BString* θα κάνουμε την επιφόρτωση του "+" ως

```
   BString operator+( const BString lhs, const BString rhs );
```

Πώς; Αφού έχουμε επιφορτώσει τον "+=" υπάρχει μια πάγια τεχνική.

22.7.1 Από τον "@=" στον "@"

Έχοντας τον "+=" (γενικώς: τον "@=") παίρνουμε τον "+" (γενικώς: τον "@") ως εξής:

```
BString operator+( const BString& lhs, const BString& rhs )
{
   BString fv( lhs );
   fv += rhs;
   return fv;
} // BStribg operator+
```

Έτσι, έχουμε εξασφαλισμένη και τη συμβατότητα με τη λειτουργία του "+=".

Πρόσεξε τον τύπο του αποτελέσματος: δεν είναι *BString*`&` αλλά *BString*. Γιατί; Διότι η επιστρεφόμενη τιμή είναι τοπική μεταβλητή της συνάρτησης και η ζωή της τελειώνει όταν τελειώνει η εκτέλεση της συνάρτησης. Στον "+=" (και την *append*) επιστρεφόμενη τιμή είναι το ίδιο το αντικείμενο (`*this`) που επιζεί και μετά τον τερματισμό εκτέλεσης της μεθόδου.

Πρόσεξε ακόμη ότι η υλοποίηση δεν χρησιμοποιεί μέλη του αντικειμένου και επομένως δεν χρειάζεται να δηλωθεί ως `friend`.

Μπορούμε να χρησιμοποιήσουμε την ίδια τεχνική για να επιφορτώσουμε τον "+" για τη *Date*; Βεβαίως, αλλά πριν το κάνουμε θα πρέπει να σκεφτούμε και κάποια άλλα πράγματα.

Κατ' αρχήν πρέπει να συμφωνήσουμε στο νόημα της πράξης έτσι ώστε να είναι συμβατή με τον "+=". Τι σημαίνει συμβατή; Σημαίνει ότι οι: "`d = d + 5`" και "`d += 5`" ("`d.forwatd(5)`") θα έχουν το ίδιο αποτέλεσμα. Αυτό μας λέει ότι μπορούμε να χρησιμοποιήσουμε την παραπάνω τεχνική.

Το άλλο σημείο που πρέπει να προσέξουμε είναι η αντιμεταθετικότητα: μετά τη δήλωση:

```
   Date d( 2007, 10, 15 ), d1;
```

οι:

```
   d1 = d + 5;
```

και

```
   d1 = 5 + d;
```

θα πρέπει να έχουν το ίδιο αποτέλεσμα.

Δηλαδή, έχουμε να υλοποιήσουμε δύο τελεστές με το ίδιο σύμβολο:[5]

`+`: *Date* × `int` ↛ *Date*

`+`: `int` × *Date* ↛ *Date*

Πέρα από την αντιμεταθετικότητα του τελεστή έχουμε και μια *ασύμμετρία* λόγω της διαφοράς τύπου των δύο ορισμάτων. Έτσι, η επιφόρτωση του γίνεται όπως και στη *BString* αλλά είναι διπλή αφού έχουμε δύο συναρτήσεις:

```
Date operator+( const Date& lhs, int rhs )
{
   Date fv( lhs );
   fv += rhs;
   return fv;
} // Date operator+
```

[5] Γιατί μερικές συναρτήσεις; Δες την υλοποίηση του "+=" και θα καταλάβεις.

```
Date operator+( int lhs, const Date& rhs )
{
   return ( rhs + lhs );
} // Date operator+
```

Και εδώ, ο τύπος του αποτελέσματος δεν είναι *Date*`&` αλλά *Date*, για ίδιους λόγους που είπαμε και στη *BString*.

22.7.2 Σύγκριση Ημερομηνιών

Μια άλλη κατηγορία «συμμετρικών»[6] τελεστών που υλοποιούνται με καθολικές συναρτήσεις (με δύο παραμέτρους) είναι οι τελεστές σύγκρισης. Έτσι, οι επιφορτώσεις των "`==`" και "`<`" για την κλάση *Date*, όπως τις κάναμε στην §19.1.4 είναι μια χαρά.

Τώρα θέλουμε να τονίσουμε μια καλή προγραμματιστική πρακτική που μπορεί να σε προφυλάξει από «δύσκολα» προγραμματιστικά λάθη:[7]

- ***Όταν επιφορτώνεις έναν τελεστή σύγκρισης "@" επιφόρτωσε και τον αντίθετό του με χρήση του "@".***

Αυτό σημαίνει ότι μετά τους "`==`" και "`<`" θα πρέπει να επιφορτώσουμε και τους "`!=`" και "`>=`" και μάλιστα ως εξής:

```
bool operator!=( const Date& lhs, const Date& rhs )
{  return !(lhs == rhs);  }

bool operator>=( const Date& lhs, const Date& rhs )
{  return !(lhs < rhs);  }
```

Παρατήρηση: ▶

Αν δηλώσεις τον δημιουργό της *Date* ως (§21.13.1):

```
   explicit Date( int yp, int mp = 1, int dp = 1 );
```

δεν μπορείς να γράψεις "`d == 2013`" ή "`2013 == d`" αλλά, υποχρεωτικώς, "`d == Date(2013)`" ή "`Date(2013) == d`" αντιστοίχως. Έτσι, δεν υπάρχει πια πρόβλημα αν επιφορτώσουμε τον "==" (και τους άλλους τελεστές σύγκρισης) με μεθόδους (με μια παράμετρο), π.χ.:

```
bool Date::operator==( const Date& rhs ) { /*. . .*/ }
```

Για τη *BString*, στον δημιουργό της οποίας δεν βάλαμε το "`explicit`", αυτό δεν ισχύει και η καθολική συνάρτηση είναι η μόνη σωστή επιλογή.◀

22.7.3 Οι Τελεστές Σύγκρισης της *BString*

Στη συνέχεια θέλουμε να προχωρήσουμε στην επιφόρτωση των τελεστών σύγκρισης ("`==`", "`<`", κλπ) στον τύπο *BString*. Θα πρέπει όμως πρώτα να συζητήσουμε τις συγκρίσεις ορμαθών χαρακτήρων, που θα χρειαστούμε στη συνέχεια.

Ας πούμε ότι έχουμε δύο πίνακες:

```
   char lhs[10], s2[12];
```

οι οποίοι περιέχουν κείμενα με μήκη *len1* και *len2*. Βάζουμε:

```
   minLen = min( len1, len2 );
```

---

[6] Γιατί «*συμμετρικών*» και όχι «*αντιμεταθετικών*»; Φυσικά, ο "`<`" δεν είναι αντιμεταθετικός: αν η *lhs* `<` *rhs* έχει τιμή `true` τότε η *rhs* `<` *lhs* έχει τιμή `false`. Εδώ με το *συμμετρικός* εννοούμε οτι από *συντακτική άποψη* ό,τι μπορεί να μπει στο ένα μέρος μπορεί να μπει και στο άλλο.

[7] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposites (such as "==" and "!="), it is appropriate to define both.*» Την πρωτοείδαμε στο Project 2 (§Prj02.3).

Αφού υπενθυμίσουμε ότι στις θέσεις *lhs*[*len*1] και *s*2[*len*2] υπάρχει ο φρουρός '\0', θεωρούμε τις:

```
k = 0;
while ( lhs[k] == s2[k] && k < minLen-1 ) ++k;
d = static_cast<int>(lhs[k]) - static_cast<int>(s2[k]);
```

Ας δούμε την τιμή που θα πάρει η *d* με μερικά παραδείγματα:

α) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2 τα ίδια: abcd. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η **while** θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == *s*2[3] == '\0' η *d* θα πάρει τιμή 0.

β) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: abce. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η while θα σταματήσει και πάλι όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'e' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('e') = 100 – 101 = –1 < 0.

γ) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: abcc. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η while θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'c' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('c') = 100 – 99 = 1 > 0.

Αυτή είναι η βασική ιδέα:

- Αν το *lhs προηγείται αλφαβητικώς* (λεξικογραφικώς) *του s*2 (περίπτωση β) η *d* παίρνει *αρνητική* τιμή.
- Αν το *lhs έπεται αλφαβητικώς του s*2 (περίπτωση γ) η *d* παίρνει *θετική* τιμή.
- Αν το *lhs είναι ίσο με το s*2 (περίπτωση α) η *d* παίρνει τιμή 0 (μηδέν).

Η σύγκριση γίνεται, κατ' αρχήν, με βάση τη θέση των χαρακτήρων στον πίνακα χαρακτήρων του υπολογιστή. Το «κατ' αρχήν» θα το συζητήσουμε αργότερα.

Δες μερικά παραδείγματα ακόμη:

δ) Έστω ότι στο *lhs* έχουμε ab και στο *s*2: abcd. Θα έχουμε *minLen* = *min*(2, 4) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

ε) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: ab. Θα έχουμε *minLen* = *min*(4, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

στ) Έστω ότι στο *lhs* έχουμε ab␣ και στο *s*2: ab. Θα έχουμε *minLen* = *min*(3, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[3] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

Εδώ έχουμε τρεις περιπτώσεις που παίρνουμε *d* == 0 χωρίς να έχουμε ισότητα. Τι σημαίνει όμως αυτό; Ότι όσο μπορούμε να συγκρίνουμε έχουμε ισότητες χαρακτήρων. Άρα, ο ορμαθός που τελειώνει μέχρι εκεί που μπορούμε να συγκρίνουμε προηγείται ενώ ο άλλος, με το μεγαλύτερο μήκος έπεται.

Με βάση τα παραπάνω, γράφουμε μια συνάρτηση-εργαλείο, εσωτερική της *BString*, για την υλοποίηση των μεθόδων που θα προδιαγράψουμε στη συνέχεια[8]:

```
int BString::stringCmpr( const char* lhs, const char* rhs, int n )
{
   int k( 0 );
   while ( lhs[k] == rhs[k] && k < n-1 ) ++k;
   return ( static_cast<int>(lhs[k]) - static_cast<int>(rhs[k]) );
```

[8] Η υλοποίηση που θα δώσουμε είναι μια παραλλαγή της υλοποίησης που δίνεται στην STL της Silicon Graphics Inc (SGI).

```
} // BString::stringCmpr
```

Η *stringCmpp()*[9] συγκρίνει τους πρώτους *n* (0 .. *n*–1) χαρακτήρες των *lhs* και *s*2 και επιστρέφει:

- *αρνητική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs προηγούνται αλφαβητικώς* (λεξικογραφικώς) των *n* πρώτων χαρακτήρων *του s*2,
- *θετική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs έπονται αλφαβητικώς* των *n* πρώτων χαρακτήρων *του s*2,
- 0 (μηδέν) αν οι *n* πρώτοι χαρακτήρες του *lhs είναι ίσοι* με τους αντίστοιχους *n* πρώτους χαρακτήρες *του s*2.

Πρόσεξε ότι δεν ελέγχουμε πουθενά την τιμή της *n*: αυτό θα πρέπει να το κάνει η κάθε μέθοδος που καλεί τη *stringCmpr()*.

Η *string* έχει τις εξής μεθόδους[10] για σύγκριση:

```
   int compare( const BString& rhs ) const;
   int compare( int pos, int n, const BString& rhs) const;
   int compare( int pos, int n,
                const BString& rhs, int pos2, int n2 ) const;
```

Όλες οι μορφές της *compare()* επιστρέφουν μιαν ακέραιη τιμή με την ίδια λογική που έχουμε για τη *stringCmpr()*. Με τη βοήθεια της *stringCmpr()* θα υλοποιήσουμε την πρώτη από αυτές τις μεθόδους για τη *BString*. Για τα παρακάτω παραδείγματα υποθέτουμε ότι έχουμε δηλώσει:

```
   BString bs1, bs2;
```

Η παράσταση **bs1.compare(bs2)** θα πρέπει να επιστρέφει τιμή *μικρότερη από, ίση με* ή *μεγαλύτερη από* 0 αν, αντιστοίχως, η τιμή της *bs*1 *προηγείται από, είναι ίση με* ή *έπεται της* τιμής της *bs*2. Μπορούμε λοιπόν να ορίσουμε:

```
int BString::compare( const BString& rhs ) const
{
   int fv( stringCmpr(bsData, rhs.bsData, min(bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = bsLen - rhs.bsLen;
   return fv;
} // BString::compare
```

Ας έλθουμε τώρα στους τελεστές. Θα επιφορτώσουμε τους τελεστές σύγκρισης με *καθολικές συναρτήσεις*. Μπορούμε να γράψουμε:

```
bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::stringCmpr(lhs.bsData, rhs.bsData,
                               min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
```

Φυσικά, όπως και στην περίπτωση του **operator<** της *Date*, θα πρέπει να δηλώσουμε μέσα στην κλάση ως **friend** τη συνάρτηση (τελεστή) **operator==**:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
public:
   BString( const char* cs = "" );
// ...
```

και ο ορισμός του τελεστή που δώσαμε πιο πάνω γίνεται νόμιμος.

---

[9] Η *stringCmpr()* είναι μια βοηθητική συνάρτηση, σαν τις *lastDay()* και *leapYear()* της *Date*.

[10] Δεν είναι ακριβώς έτσι. Αν ψάξεις λίγο το αρχείο string θα δεις ότι είναι πιο πολύπλοκα.

22.7.3.1 * Η Σειρά Ταξινόμησης

Οι συγκρίσεις που είδαμε παραπάνω στηρίζονται στη θέση των χαρακτήρων μέσα στο σύνολο χαρακτήρων του υπολογιστή. Αυτό δεν μας βολεύει καθόλου όταν θέλουμε να ταξινομήσουμε λέξεις κατά τη συνήθη αλφαβητική σειρά. Όπως είναι τοποθετημένα τα ελληνικά γράμματα στο πρότυπο ΕΛΟΤ 928 θα έχεις τα εξής περίεργα: το όνομα `ΝΙΚΟΛΑΪΔΗΣ` ταξινομείται μετά το `ΝΙΚΟΛΑΚΗΣ` και η λέξη `μικροψυχία` πριν από τα `μικροϋλικά`.

Αυτό το πρόβλημα λύνεται με τη **σειρά ταξινόμησης** (collating sequence). Αυτή είναι μια συνάρτηση που καθορίζει τη σειρά των χαρακτήρων για την ταξινόμηση. Για σύνολο 256 χαρακτήρων μπορείς να την υλοποιήσεις πολύ εύκολα με έναν πίνακα:

```
   int cs[256];
```

Σε κάθε στοιχείο του cs βάζουμε ως τιμή τη σειρά με την οποία θέλουμε να ταξινομείται. Π.χ.:

```
   for ( int k = 0; k <= 255; ++k )  cs[k] = k;
// ...
// Λατινικοί
   cs['A'] = cs['a'] = 65;
   cs['B'] = cs['b'] = 66;
// ...
// Ελληνικοί
   cs[256+'Α'] = cs[256+'Ά'] = cs[256+'α'] = cs[256+'ά'] = 161;
   cs[256+'Β'] = cs[256+'β'] = 162;
// ...
   cs[256+'Υ'] = cs[256+'Ύ'] = cs[256+'Ϋ'] = cs[256+'υ']
               = cs[256+'ύ'] = cs[256+'ϋ'] = cs[256+'ΰ'] = 213;
// ...
```

Θυμίσου ότι τα ελληνικά γράμματα, ως σταθερές τύπου (`signed`) `char`, έχουν αρνητική τιμή[11]. Αν δεν καταλαβαίνεις τι ακριβώς κάνουμε γύρισε πίσω στο Κεφ. 3.

Αφού η βάση του μηχανισμού σύγκρισης είναι η *stringCmpr()* την αλλάζουμε ως εξής:

```
int BString::stringCmprCS( const char* lhs, const char* rhs, int n ) const
{
   int k = 0;

   while ( cs[ lhs[k] ] == cs[ rhs[k] ] && k < n-1) ++k;
   return ( cs[ lhs[k] ] - cs[ rhs[k] ] );
} // BString::stringCmprCS
```

22.8 Τελεστές για τη *Vector3*

Επιστρέφουμε τώρα στην κλάση *Vector3*, που είδαμε στο Project 2, για να ξαναδούμε –όπως έχουμε υποσχεθεί– τις επιφορτώσεις τελεστών.

Εξ αρχής να πούμε το εξής: Η *Vector3* έχει τετριμμένη αναλλοίωτη, όλα τα μέλη είναι ανοικτά και έτσι δεν υπάρχει πρόβλημα για την επιφόρτωση όλων των τελεστών με καθολικές συναρτήσεις. *Αυτά που δώσαμε στο Project 2 είναι απολύτως σωστά*. Αυτά που θα πούμε εδώ είναι παραδείγματα εφαρμογής κάποιων από τις συνταγές που είδαμε στο παρόν κεφάλαιο.

Ας ξεκινήσουμε με τον (προθεματικό) *ενικό τελεστή* "`-`" (πρόσημο). Αν **v** = (*x*, *y*, *z*) τότε `-`**v** = (`-`*x*, `-`*y*, `-`*z*). Σύμφωνα με τη συνταγή της §14.6.4 θα έχουμε:

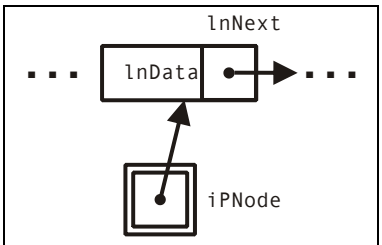
```
   Vector3 operator-( Vector3 rhs )
```

(ο τελεστής δεν αλλάζει το αντικείμενο) και σύμφωνα με τη συνταγή της §22.3.1 θα πρέπει να επιφορτώσουμε με μέθοδο:

```
   Vector3 operator-() const {  return Vector3( -x, -y, -z );  }
```

[11] Κανονικά πρέπει να γράψουμε `cs[256+static_cast<int>('A')] = 193` αλλά και έτσι περνάει.

Με μεθόδους θα πρέπει να επιφορτώσουμε και τους μη-αντιμεταθετικούς τελεστές με το αντικείμενο αριστερά "`+=`", "`-=`", "`*=`":



Σχ. 22-1 Ένας προσεγγιστής: το μοναδικό του μέλος, *iPNode*, είναι ένα βέλος που δείχνει κάποιον κόμβο της λίστας.

```
Vector3& Vector3::operator+=( const Vector3& rhs )
{
   x += rhs.x;  y += rhs.y;  z += rhs.z;
   return *this;
} // Vector3::operator+=

Vector3& Vector3::operator-=( const Vector3& rhs )
{
   x -= rhs.x;  y -= rhs.y;  z -= rhs.z;
   return *this;
} // Vector3::operator-=

Vector3& Vector3::operator*=( double rhs )
{
   x *= rhs;  y *= rhs;  z *= rhs;
   return *this;
} // operator*=( const Vector3&
```

Μέθοδοι θα γίνουν και οι *abs()* και *abs2()*:

```
   double abs2() const {  return x*x + y*y + z*z;  }
   double abs() const {  return sqrt( abs2() );  }
```

Οι υπόλοιπες καθολικές συναρτήσεις παραμένουν αλλά αλλάζουν οι ορισμοί των "`operator+`", "`operator-`", "`operator*`" που υλοποιούνται όπως μάθαμε στην §22.6.1, π.χ.:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv( lhs );
   fv += rhs;
   return fv;
} // operator+( const Vector3&
```

22.9 Διασχίζοντας τη Λίστα με τον "++" – Προσεγγιστές

Στην §21.11 λέγαμε: «*Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. ... η διάσχιση της λίστας δεν μπορεί να "κρυφτεί" σε κάποια μέθοδο.*» Έτσι, δίνουμε στο πρόγραμμα-πελάτη τη δυνατότητα να δίνει την εντολή "`p = p->lnNext`" με τον κίνδυνο να πάρει η *p->lnNext* τιμή που βγάζει από τη λίστα.

Τώρα ας εξετάσουμε την εξής ιδέα:

- όπως η "`++n`" προχωρεί την τιμή της *n* στον επόμενο ακέραιο,
- όπως η "`++d`" προχωρεί την τιμή της *d* στην επόμενη ημερομηνία

να επιφορτώσουμε τον "`++`" ώστε να προχωρεί τον *p* στον επόμενο κόμβο της λίστας. Και ακόμη να επιφορτώσουμε τον (ενικό) "`*`" ώστε η "`*p`" να μας δίνει το *p->lnData*. Έτσι δεν θα υπάρχει λόγος να «ανοίγουμε» τη δομή της λίστας στο πρόγραμμα-πελάτη.

Καλή ιδέα, αλλά έχει ένα προβληματάκι: η C++ δεν μας επιτρέπει να επιφορτώνουμε τελεστές για τους πρωτογενείς τύπους. Όμως αυτό το πρόβλημα λύνεται: Γράφουμε μια κλάση **προσεγγιστή** (iterator) που έχει ένα μόνον μέλος, το βέλος μας! Στο Σχ. 22-1 βλέπεις τι περίπου εννοούμε. Το βλέπεις και στη συνέχεια:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
```

```
public:
   class Iterator
   {
   public:
// . . .
      Iterator& operator++()
      {  iPNode = iPNode->lnNext;  return *this;  };
      GrElmn& operator*() {  return ( iPNode->lnData );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
// . . .
}; // SList
```

Εδώ όμως βλέπεις και άλλα:

- Η κλάση *SList* έχει τώρα δύο περιοχές **private**.
- Στην πρώτη περιοχή **private** έχουμε «κρύψει» τον ορισμό της *ListNode* που πρέπει να προηγείται της δήλωσης

```
      ListNode* iPNode;
```

- Στη νέα κλάση, που είναι κλάση «περιτυλίγματος», βλέπεις την επιφόρτωση των "`++`" και "`*`". Εδώ βρίσκεται το ενδιαφέρον!

Τι άλλο πρέπει να αλλάξουμε στην *SList*; Προφανώς τις *begin()* και *end()*· δεν είναι δυνατόν πια να επιστρέφουν βέλος. Θα τις κάνουμε:

```
   Iterator begin() {  return Iterator( slHead );  }
   Iterator end() {  return Iterator( slTail );  }
```

Προφανώς χρειάζεται να γράψουμε και έναν δημιουργό για τη νέα κλάση:

```
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
```

Πώς θα διασχίσουμε τη λίστα για να φυλάξουμε το περιεχόμενό της; Έτσι:

```
      for ( SList::Iterator it(lst.begin());
            it != lst.end();
            ++it )
         writeRandom( *it, bInOut );
```

Εδώ τα βλέπεις όλα:

- Χρησιμοποιούμε τον προσεγγιστή *it* για να πάμε από `lst.begin()` μέχρι `lst.end()`.
- Προχωρούμε με "`++it`".[12]
- Σε κάθε βήμα φυλάγουμε το περιεχόμενο ενός κόμβου που δίνεται από την αποπαραπομπή "`*it`".

Βλέπεις όμως ότι μας λείπει και κάτι: δεν επιφορτώσαμε τον "`!=`". Μπορούμε να τον επιφορτώσουμε με καθολική συνάρτηση:

```
bool operator!=( const SList::Iterator& a, const SList::Iterator& b )
{  return ( a.iPNode != b.iPNode );  }
```

αλλά θα πρέπει να την δηλώσουμε ως "`friend`".

Αφού έχουμε δηλώσει "`explicit Iterator`" δεν υπάρχει πρόβλημα αν κάνουμε την επιφόρτωση με μέθοδο:

---

[12] Οι προσεγγιστές που βλέπουμε εδώ έχουν μια διεύθυνση κίνησης: από την αρχή της λίστας προς το τέλος της («κινούνται» μόνον με τον "`++`"). Ένας τέτοιος προσεγγιστής καλείται **πρόσθιος** (forward) **προσεγγιστής**. Αν κάθε κόμβος είχε και βέλος προς τον προηγούμενο κόμβο θα ήταν δυνατή κίνηση του προσεγγιστή και από το τέλος προς την αρχή (και επιφόρτωση του "`--`"). Ένας τέτοιος προσεγγιστής λέγεται **αμφίδρομος** (bidirectional).

```
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
```

Κατά τη συνήθειά μας θα ορίσουμε και την:

```
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
```

Κλάση εξαιρέσεων θα γράψουμε; Ναι –κατ' αρχήν– διότι οι δύο βασικές επιφορτώσεις είναι ανοικτές σε λάθος χρήση. Δεν είναι όμως και λάθος να χρησιμοποιήσουμε την *SListXptn*.

- Ο "++" δεν μπορεί να χρησιμοποιηθεί αν έχουμε φτάσει στον φρουρό:

```
SList::Iterator& SList::Iterator::operator++()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator++", SListXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SList::Iterator::operator++
```

- Το ίδιο ισχύει και για τον "*":

```
GrElmn& SList::Iterator::operator*()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator*", SListXptn::listEnd );
   return ( iPNode->lnData );
} // SList::Iterator::operator*
```

Ξαναδίνουμε το τμήμα του ορισμού της *SList* που έχει τις αλλαγές:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      GrElmn& operator*();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
   SList( const SList& rhs );
   ~SList();
   SList& operator=( const SList& rhs );
   void swap( SList& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
// . . .
}; // SList

struct SListXptn
{
   enum { noMemory, notFound, listEnd };
// . . .
```

```
}; // SListXptn
```

Παρατήρηση:▶

Τώρα που έχουμε τον προσεγγιστή και επιφορτώσαμε τους "`++`" και "`*`" –και έχοντας επιφορτώσει τον "`==`" για τη *GrElmn*– μπορούμε να χρησιμοποιήσουμε την (*std*::)*find()* για αναζήτηση στη λίστα.

Για να τη δοκιμάσεις βάλε στο πρόγραμμα της §21.11 τη νέα μορφή της *SList*, μην ξεχάσεις την "`include <algorithm>`" και στη **main**, μετά το τέλος της **do-while**, βάλε άλλη μια:

```
      do {
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            SList::Iterator it;
            it = std::find( lst.begin(), lst.end(), GrElmn(aa) );
            if ( it != lst.end() )  (*it).display( cout );
                                else  cout << "not found" << endl;
         }
      } while ( aa != 0 );
```

Σημείωσε τους ατομικούς αριθμούς των στοιχείων που θα βάλεις στη λίστα με την πρώτη **do-while** και σύγκρινε με αυτά που θα σου βγάζει η δεύτερη.◀

22.9.1 Επιφόρτωση του "->"

Αν επιφορτώσουμε τον "`->`" για την κλάση *SList*::*Iterator* μπορούμε αντί για

"`(*it).display(cout)`"

να γράψουμε

"`it->display(cout)`"

Ας δούμε πώς γίνεται αυτή η επιφόρτωση.

Ο "`->`" είναι, κατ' αρχήν, ένας δυαδικός τελεστής: αριστερά υπάρχει (για την περίπτωσή μας) ένας προσεγγιστής και δεξιά ένα μέλος ή μια μέθοδος της κλάσης που περιέχει η λίστα. Στη συνάρτηση που θα γράψουμε δεν υπάρχει δεύτερη παράμετρος (αντίστοιχη του δεξιού μέρους). Η συνάρτηση επιστρέφει τη διεύθυνση αυτού (βέλος προς αυτό) που επιστρέφει η `operator*()`. Και αφού η `operator*()` επιστρέφει "`iPNode->lnData`" θα έχουμε:

```
      GrElmn* operator->() const
      {
         if ( iPNode->lnNext == 0 )
            throw SListXptn( "Iterator::operator->",
                             SListXptn::listEnd );
         return ( &(iPNode->lnData) );
      } // SList::Iterator::operator->
```

Η επιστρεφόμενη τιμή –που είναι βέλος– αντικαθιστά στην παράσταση τον προσεγγιστή και ο "`->`" δρα σε αυτό το βέλος. Στην περίπτωσή μας θα έχουμε:

`(&(iPNode->lnData))->display(cout)`

ή αλλιώς:

`(iPNode->lnData).display(cout)`

Αντί για "`it->display(cout)`" μπορείς να γράψεις:

"`(it.operator->())->display(cout)`"

22.10 * Απόκρυψη Υλοποίησης – Τεχνική "pimpl"

Και τώρα που τα μάθαμε όλα ας κλείσουμε μια εκκρεμότητα που έχουμε από την §19.3: να δούμε πώς αποκρύπτουμε την υλοποίηση της κλάσης (για κάποιον από τους λόγους που είδαμε στην §19.3.1).

Αυτό που θα θέλαμε να κάνουμε είναι το εξής: Να παραδίδουμε στον προγραμματιστή που θα χρησιμοποιήσει μια κλάση *C* (που έχουμε γράψει) το αρχείο **C.obj** (ή **C.o**) και το αρχείο **C.h** με τον ορισμό της κλάσης στον οποίον όμως θα υπάρχει μόνον το μέρος "**public**" με τις δηλώσεις των μεθόδων. Το μέρος "**private**" και οι ορισμοί των μεθόδων θα πρέπει να βρίσκονται μεταγλωττισμένα στο **C.obj**.

Ο συνδέτης δεν θα μας επιτρέψει κάτι τέτοιο. Θα μας επιτρέψει όμως κάτι που μοιάζει με αυτό: ένα μέρος "**private**" που θα έχει μέσα μόνο ένα βέλος. Ας δούμε αυτή τη λύση με ένα συγκεκριμένο παράδειγμα.

Η κλάση *BString*, όπως την έχουμε αναπτύξει μέχρι τώρα, είναι:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs ) { return (*this = rhs); }
   const char* c_str() const;
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   char& at( int k ) const;
   char& operator[]( int pos ) const {  return bsData[pos];  }
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs )
   { return (*this += rhs); }
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τώρα θα κάνουμε την εξής αλλαγή: Θα ορίσουμε μια άλλη κλάση:

```
struct BStringImpl
{
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BStringImpl
```

και στο μέρος "**private**" της κλάσης θα βάλουμε μόνο ένα βέλος:

```
private:
   BStringImpl* bsHandle;
}; // BString
```

Για να μπορέσουμε να κάνουμε αυτήν τη δήλωση για το *bsHandle* θα πρέπει ο μεταγλωττιστής να βρει *προειδοποιητική δήλωση* της *BStringImpl*:

```
   struct BStringImpl;
```

που λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

Μπορούμε να βάλουμε αυτήν τη δήλωση πριν από τον ορισμό της *BString* (στο `BString.h`). Είναι όμως προτιμότερο να τη βάλουμε μέσα στον ορισμό της κλάσης. Θα δεις στη συνέχεια γιατί αυτό μας συμφέρει.

Στο μέρος "`public`" δεν θα αφήσουμε οτιδήποτε έχει σχέση με υλοποίηση:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τα υπόλοιπα θα πάνε στο `BString.cpp`. Εκεί θα πάνε και ο ορισμοί της νέας κλάσης και των μεθόδων της.

Αν θέλεις μπορείς, όπως κάνουμε μέχρι τώρα, να βάλεις όσα αφορούν την κλάση *BStringImpl* σε χωριστά αρχεία `BStringImpl.h` και `BStringImpl.cpp`. Στην περίπτωση αυτή θα πρέπει να βάλεις "`#include "BStringImpl.h"`" και "`#include "BStringImpl.cpp"`" στην αρχή του `BString.cpp`.

Σημείωση: ▶
Τώρα μπορείς να καταλάβεις και το (συνηθισμένο) όνομα της τεχικής: ***p***ointer to (ή ***p***rivate) ***impl***ementation (βέλος προς την υλοποίηση). Θα συναντήσεις ακόμη τα ονόματα "handle class", "Cheshire Cat" και "compiler firewall". Η *BStringImpl* είναι μια **αδιαφανής** (opaque) δομή δεδομένων ενώ **αδιαφανές** ονομάζεται και το βέλος *bsHandle*· ονομάζεται ακόμη και **λαβή** (handle).

Λαβές για αδιαφανείς δομές δεδομένων χρησιμοποιούνται σε μικρή ή μεγάλη έκταση στις API για διάφορα προγραμματιστικά περιβάλλοντα.◀

Ας δούμε τώρα μερικά βασικά σημεία της υλοποίησης που θα αλλάξουν.

Ο ερήμην δημιουργός θα είναι:

```
BString::BString( const char* rhs )
{
   try {  bsHandle = new BStringImpl( rhs );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Αυτός ο δημιουργός μας επιβάλλει να γράψουμε έναν δημιουργό:

```
   BStringImpl( const char* rhs="" );
```

για την *BStringImpl*:

```
BString::BStringImpl::BStringImpl( const char* rhs )
{
   bsLen = cStrLen( rhs );
```

```
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   bsData = new char[bsReserved];
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BStringImpl::BStringImpl
```

Πρόσεξε ότι η νέα κλάση αναφέρεται ως "`BString::BStringImpl`".

Αν αποτύχει η "`new`" και ριχτεί *bad_alloc* θα συλληφθεί από την "`catch`" του δημιουργού της *BString*. Και αν σε κάποιο πρόγραμμα έχουμε τη δήλωση:

```
   BStringImpl a;
```

και κατά την εκτέλεση του ερήμην δημιουργού ριχτεί μια *bad_alloc* πού θα συλληφθεί; Το ότι η κλάση μας είναι δηλωμένη στο μέρος "`private`" της *BString* μας εξασφαλίζει ότι δεν θα υπάρξει τέτοια δήλωση· θα την «απαγορεύσει» ο μεταγλωττιστής.

Αφού κάθε αντικείμενο δεσμεύει δυναμική μνήμη –για ένα αντικείμενο *BStringImpl*– θα πρέπει να γράψουμε δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
   try {  bsHandle = new BStringImpl( *(rhs.bsHandle) );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Στη "`new BStringImpl( *(rhs.bsHandle) )`" καλείται ο δημιουργός αντιγραφής της *BStringImpl*. Αφού ένα αντικείμενο αυτής της κλάσης δεσμεύει επίσης μνήμη για το *bsData* θα πρέπει να γράψουμε δημιουργό αντιγραφής και για αυτήν:

```
BString::BStringImpl::BStringImpl( const BString::BStringImpl& rhs )
{
   bsData = new char[rhs.bsReserved];
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BStringImpl::BStringImpl
```

Αυτά που είπαμε για τις *bad_alloc* από τους ερήμην δημιουργούς των δύο κλάσεων ισχύουν και για τους δημιουργούς αντιγραφής.

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε (επιφορτώσουμε) και τελεστή εκχώρησης:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
} // BString::operator=
```

Αυτός είναι ολόιδιος με αυτόν που γράψαμε στην §21.6.1 (τελευταία μορφή) αλλά έχουμε διαφορετική *swap()*:

```
void BString::swap( BString& rhs )
{
   std::swap( bsHandle, rhs.bsHandle );
} // BString::swap
```

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε τελεστή εκχώρησης και για τη *BStringImpl*:

```
BString::BStringImpl& BString::BStringImpl::operator=(
                                          const BString::BStringImpl& rhs )
```

```
{
   BString::BStringImpl tmp( rhs );
   if ( &rhs != this )
      swap( tmp );
   return *this;
} // BString::BStringImpl::BStringImpl& operator=
```

Βέβαια, προς το παρόν μας είναι άχρηστος αλλά μπορεί να μας χρειαστεί αν συνεχίσουμε την ανάπτυξη της *BString*.

Οι καταστροφείς είναι:

```
BString::~BString() {  delete bsHandle;  }
```

και

```
BString::BStringImpl::~BStringImpl() {  delete bsData;  }
```

Άλλες μεθόδους για τη *BStringImpl* θα γράψουμε; Μόνον τη *swap()*:

```
void BString::BStringImpl::swap( BString::BStringImpl& rhs )
{
   std::swap( bsData, rhs.bsData );
   std::swap( bsLen, rhs.bsLen );
   std::swap( bsReserved, rhs.bsReserved );
} // BString::BStringImpl::swap
```

Όπως βλέπεις πρόκειται για την παλιά `BString::swap` σε νέα συσκευασία.

Οι μέθοδοι της *BString* πρέπει να ξαναγραφούν με τις εξής τροποποιήσεις: Θα πρέπει να γράφεις

- το πρόθεμα "`bsHandle->`" πριν από τα *bsData, bsLen, bsReserved*.
- το πρόθεμα "`BStringImpl::`" πριν από τη σταθερά *bsIncr* και τις (στατικές) βοηθητικές συναρτήσεις *cStrLen, stringCmpr*.

Ας δούμε πώς θα γραφεί η επιφόρτωση του "`+=`":

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 >
                                          bsHandle->bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsHandle->bsLen+(rhs.bsHandle)->bsLen+1)/
                    BStringImpl::bsIncr+1)*BStringImpl::bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "operator+=",
                           BStringXptn::allocFailed );  }
      for ( int k(0); k < bsHandle->bsLen; ++k )
         tmp[k] = bsHandle->bsData[k];
      delete[] bsHandle->bsData;
      bsHandle->bsData = tmp;
   }
   for ( int j(0), k(bsHandle->bsLen);
         j < (rhs.bsHandle)->bsLen;
         ++j, ++k )
      bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];
   bsHandle->bsLen += (rhs.bsHandle)->bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις:

- Στη συνθήκη της **if** αντί για

  `bsLen + rhs.bsLen + 1 > bsReserved`

  γράφουμε

  `bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 > bsHandle->bsReserved`
- Στην αρχική τιμή της *tmpRes* γράφουμε "`BStringImpl::bsIncr`" και όχι `bsIncr`.
- Στη δεύτερη **for** αντιγράφουμε με τη

  `bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];`

και όχι με τη

```
bsData[k] = rhs.bsData[j];
```

Τα ίδια ισχύουν και για τις επιφορτώσεις των τελεστών "`==`" και "`<<`" που τώρα τις έχουμε δηλώσει "`friend`":[13]

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < (rhs.bsHandle)->bsLen; ++k )
      tout << (rhs.bsHandle)->bsData[k];
   return tout;
} // operator<<( ofstream, BString

bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::BStringImpl::stringCmpr(
                                          (lhs.bsHandle)->bsData,
                                          (rhs.bsHandle)->bsData,
            min((lhs.bsHandle)->bsLen, (rhs.bsHandle)->bsLen)) );
   if ( fv == 0 )
      fv = (lhs.bsHandle)->bsLen - (rhs.bsHandle)->bsLen;
   return ( fv == 0 );
} // operator==
```

Ακολουθώντας αυτά που είπαμε στην §18.3 μπορούμε από το `BString.cpp` (και το `BString.h`) να πάρουμε το μεταγλωττισμένο αρχείο `BString.o` (ή `BString.obj`).

Τώρα το `BString.h` είναι:

```
#ifndef _BSTRING_H
#define _BSTRING_H

#include <fstream>
#include <string>

using namespace std;

class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );

struct BStringXptn
{
   enum { allocFailed, outOfRange };
```

[13] Η επιφόρτωση του "+" παραμένει όπως δόθηκε στην §22.6.1.

```
   char funcName[100];
   int  errorCode;
   int  errorValue;
   BStringXptn( char* mn, int ec, int ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BStringXptn

#endif // _BSTRING_H
```

Ένας προγραμματιστής, έχοντας το **BString.o** (ή **.obj**) και το **BString.h**, *μπορεί να χρησιμοποιήσει στα προγράμματά του τη BString χωρίς να ξέρει πώς είναι γραμμένη.*

Και ακόμη: αν εμείς αλλάξουμε την υλοποίηση της κλάσης χωρίς να αλλάξουμε τη διεπαφή (δηλαδή αυτά που έχουμε στο "**public**" και τις επικεφαλίδες των καθολικών συναρτήσεων) οποιοδήποτε πρόγραμμα χρησιμοποιεί την κλάση δεν χρειάζεται να περάσει ξανά από τον μεταγλωττιστή· αρκεί να περάσει από τον συνδέτη.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

22-1 Επιφόρτωσε τους τελεστές "**-**", "**--**" και "**-=**" για τη *Date* ώστε να δίνουν αντιστοίχως τη διαφορά (σε ημέρες) δύο ημερομηνιών και την «οπισθοδρόμηση» μιας ημερομηνίας.

22-2 Επιφόρτωσε όλους τους τελεστές σύγκρισης για τις *Date* και *BString*.

project

4

Φοιτητές και Μαθήματα Αλλιώς

Περιεχόμενα:

Prj04.1 Το Πρόβλημα

Το πρόβλημα που έχουμε να λύσουμε είναι μια παραλλαγή του προβλήματος που λύσαμε στο Project 3. Οι επιπλέον απαιτήσεις είναι:

Α. Σε κάθε αντικείμενο κλάσης Student θα αποθηκεύονται –σε δυναμικό πίνακα– οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.

Αυτό θα μας δημιουργήσει το εξής πρόβλημα: Όταν φυλάγουμε τα αντικείμενα τύπου *Student* στο `students.dta` αυτά θα έχουν διαφορετικό μέγεθος. Έτσι, δεν θα μπορούμε να χειριστούμε το αρχείο με τον τρόπο που μάθαμε στην §15.13. Για να ανακτήσουμε τη δυνα-

τοτητα κατ' ευθείαν πρόσβασης στα αντικείμενα του αρχείου θα χρειαστούμε ένα **ευρετήριο** (index) με στοιχεία τύπου

```
struct IndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
}; // IndexEntry
```

Το πρόγραμμά μας θα πρέπει να οργανώνει αυτά τα στοιχεία σε έναν πίνακα index *που θα τον φυλάγει σε ένα (μη-μορφοποιημένο) αρχείο* `students.ndx`.

Β. Εκτός από το πρόγραμμα που γράψαμε –και θα ξαναγράψουμε με κάποιες διαφορές– και το ονομάζουμε ***πρόγραμμα δημιουργίας****, θέλουμε άλλο ένα πρόγραμμα που θα το ονομάσουμε* ***πρόγραμμα εκμετάλλευσης****.*

Αυτό το πρόγραμμα θα κάνει κατά βάση μια δουλειά: θα δέχεται από τον χρήστη έναν αριθμό μητρώου φοιτητή/τριας και θα δείχνει τα στοιχεία του/της όπως υπάρχουν στο `students.dta`*. Αν δεν υπάρχει τέτοιος αριθμό μητρώου θα δίνει κατάλληλο μήνυμα. Αυτό θα επαναλαμβάνεται μέχρι να ζητηθεί "ΤΕΛΟΣ" από τον χρήστη.*

Ακόμη, αυτή τη φορά:

- θα γράψουμε τις κλάσεις μας με βάση τη «συνταγή» της §21.8,
- θα χειριστούμε με πιο «ορθόδοξο» τρόπο τους εξωτερικούς πίνακες μαθημάτων και φοιτητών και
- θα γράψουμε κάπως διαφορετικά τις κλάσεις εξαιρέσεων: κάθε αντικείμενο-εξαίρεση θα περιέχει και το κλειδί του αντικειμένου που έχει το πρόβλημα. Το είχαμε υποσχεθεί στην §19.4.

Σε σχέση με το πρώτο σημείο θα πρέπει να τονίσουμε το εξής: *Ο σωστός τρόπος εργασίας είναι αυτός που είδαμε στο Project 3, δηλαδή βλέπουμε τις ανάγκες για την κάθε κλάση όπως υπαγορεύονται από τις εφαρμογές που τη χρησιμοποιούν και την εξοπλίζουμε καταλλήλως.* Η «συνταγή» της §21.8 ασχολείται με μερικές πολύ συνηθισμένες ανάγκες που καλό είναι να αντιμετωπίζονται με πάγιο τρόπο.

Prj04.2 Η Κλάση *Course*

Πριν αρχίσουμε να εφαρμόζουμε τη «συνταγή» μας για την κλάση *Course* θα πρέπει να πάρουμε υπόψη μας κάτι που υπάρχει στις προδιαγραφές και αφορα την κλάση *Student*: «*Σε κάθε αντικείμενο κλάσης Student θα αποθηκεύονται –σε δυναμικό πίνακα– οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.*»

Είναι φανερό ότι στον πίνακα αυτόν θα πρέπει να κάνουμε αναζητήσεις. Αν θέλουμε να χρησιμοποιήσουμε το περίγραμμα *linSearch()* θα πρέπει να έχουμε επιφορτώσει τον τελεστή "`!=`" για τον τύπο του *cCode*. Αυτό δεν μπορεί να γίνει αν έχουμε δηλώσει:

```
   char       cCode[cCodeSz];
```

Θα πρέπει να ορίσουμε έναν νέο τύπο (κλάση περιτυλίγματος), ας τον πούμε *CourseKey*, ως εξής:

```
class Course
{
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strcpy( s, aKey.c_str(), cCodeSz-1 );  s[cCodeSz-1] = '\0';  }
   }; // CourseKey
// . . .
```

και να δηλώσουμε:

```
// . . .
private:
// . . .
   CourseKey    cCode;              // κωδικός μαθήματος
```

Τώρα μπορούμε να επιφορτώσουμε τον "!=" ως εξής:

```
bool operator!=( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) != 0 );  }
```

Και τώρα ας εφαρμόσουμε τη συνταγή μας.

ΑΝ, αναλλοίωτη:

((*strlen*(*cCode*.*s*) == *cCodeSz*-1) || (*strlen*(*cCode*.*s*) == 0)) && (*strlen*(*cTitle*) < *cTitleSz*) &&
(1 <= *cFSem* <= 8) && (*cSector* ∈ {'Υ','Π','Μ','Γ','Ξ'})[1] &&
(*cCateg* ∈ {"ΜΓΥ","ΜΕΥ","ΜΕ","ΔΟΝ"})[2] && (*cWH* >= 0) && (*cUnits* >= 0) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) || (*strlen*(*cPrereq*.*s*) == 0)) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) ⇒ (*cPrereq* != *cCode*)) &&
(*cNoOfStudents* >= 0)

Παρατηρήσεις: ▶

1. Οι *strlen*(*cCode*.*s*) == *cCodeSz*-1 και *strlen*(*cTitle*) < *cTitleSz* είναι διαφορετικές: Ο κωδικός του μαθήματος πρέπει να έχει ακριβώς 7 χαρακτήρες (και άλλα χαρακτηριστικά δομής –που περιγράφονται στην άσκ. Prj03-1– αλλά εδώ τα αγνοούμε.) *Αν δοθούν λιγότεροι ή περισσότεροι θα πρέπει να ρίξουμε εξαίρεση.* Ο τίτλος του μαθήματος πρέπει να έχει λιγότερους από 80 χαρακτήρες. Αν μας δοθούν 80 ή περισσότεροι *δεν ρίχνουμε εξαίρεση*· απλώς κρατούμε τους πρώτους 79 (*cTitleSz*-1) και αγνοούμε τους παραπανήσιους.
2. Η *strlen*(*cCode*.*s*) == 0 θα πρέπει να επιτρέπεται στην περίπτωση δήλωσης της μορφής "Course c0" (και μόνον).
3. Οι *cWH* == 0 και *cUnits* == 0 είναι δεκτές μόνον στον δημιουργό και όχι στις αντίστοιχες "*set*".
4. Η

((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) || (*strlen*(*cPrereq*.*s*) == 0)) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) ⇒ (*cPrereq* != *cCode*))

δεν φτάνει: θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων. Για να μπορέσεις να βάλεις αυτόν τον περιορισμό στην αναλλοίωτη και τον αντίστοιχο έλεγχο στις μεθόδους θα πρέπει να σχεδιάσεις τις κλάσεις σου έτσι ώστε τα αντικείμενα τύπου *Course* να υπάρχουν μόνον μέσα σε κάποιον «πίνακα μαθημάτων».
5. Ο έλεγχος «αυτοαναφοράς» (κωδικός προαπαιτουμένου ίδιος με κωδικό μαθήματος) θα πρέπει να γίνεται όταν αλλάζουμε κωδικό προαπαιτουμένου αλλά και όταν αλλάζουμε κωδικό μαθήματος.◀

Η κλάση εξαιρέσεων θα είναι ως εξής:

```
struct CourseXptn
{
   enum { . . . };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
               const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
```

[1] Έχουν σχέση με το Τμήμα Βιομηχανικής Πληροφορικής του ΤΕΙ Καβάλας.

[2] Αναφέρονται στα προγράμματα σπουδών ΤΕΙ κάποιας εποχής...

```
}; // CourseXptn
```

Το νέο στοιχείο εδώ είναι το μέλος *objKey*. Εδώ θα βάζουμε το κλειδί του αντικειμένου που έχει το πρόβλημα.

Να σημειώσουμε ακόμη ότι για να μπορούμε να χρησιμοποιήσουμε στη δήλωση του *objKey* τη σταθερά *Course*`::`*cCodeSz* θα πρέπει να μεταφέρουμε τον ορισμό της σε περιοχή "`public`" στη δήλωση της *Course*.

ΔΕ, ερήμην δημιουργός**:** Θέλουμε να έχουμε δυνατότητα να κάνουμε δηλώσεις σαν τις:

```
   Course c0;
   Course c1( "EY01010" );
   Course c2( "EY01010", "Εισαγωγή στον Προγραμματισμό (0)" );
```

Σύμφωνα με αυτά που είπαμε στην §21.11.1, επειδή –όπως φαίνεται στο δεύτερο από τα παραδείγματα– ο δημιουργός θα επιτρέπει και τη μετατροπή τύπου (κλήση με μια και μόνη παράμετρο τύπου *string*), δηλώνουμε:

```
   explicit Course( string aCode="", string aTitle="" );
```

και ορίζουμε:

```
Course::Course( string aCode, string aTitle )
{
   if ( aCode.length() != cCodeSz-1 &&
        (aCode.length() != 0 || aTitle.length() != 0) )
      throw CourseXptn( CourseKey(""), "Course", CourseXptn::keyLen,
                        aCode.c_str() );
   cCode = CourseKey( aCode );
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
   cFSem = 0;
   cCompuls = false;
   cCateg[0] = '\0';
   cWH = 0;
   cUnits = 0;
   cPrereq = CourseKey( "" );
   cNoOfStudents = 0;
} // Course::Course
```

Να εξηγήσουμε τον έλεγχο που κάνουμε: Ο κωδικός μαθήματος θα πρέπει να έχει μήκος 7. Μπορεί να έχει και μήκος 0 αλλά μόνον σε δήλωση της μορφής "`Course c0`". Όμως δεν επιτρέπεται δήλωση της μορφής:

```
   Course c2( "", "Εισαγωγή στον Προγραμματισμό (0)" );
```

όπου έχουμε τίτλο μαθήματος αλλά όχι κωδικό. Δηλαδή, πιο σωστά, θα έπρεπε να έχουμε στην αναλλοίωτη:

(*strlen*(*cCode*`.`*s*) `==` *cCodeSz*-1) `||` (*strlen*(*cCode*`.`*s*) `==` 0 `&&` *strlen*(*cTitle*) `==` 0)

Η συνθήκη ελέγχου είναι άρνηση αυτής της συνθήκης για τις αντίστοιχες παραμέτρους του δημιουργού.

Αν μας δώσουν «παράνομο» κωδικό μαθήματος στον δημιουργό τι κλειδί θα βάλουμε στην εξαίρεση; Όπως βλεπεις, εδώ βάλαμε "`CourseKey("")`" (κενός ορμαθός). Η «αμαρτωλή» τιμή (*aCode*) περνάει με το τελευταίο όρισμα.

Κατά τα άλλα, αντί για "`cCode = CourseKey( aCode )`" θα μπορούσαμε να είχαμε γράψει:

```
   CourseKey cCode;
   strcpy( cCode.s, aCode.c_str() );
```

και αντί για "`cPrereq = CourseKey( "" )`" θα μπορούσαμε να έχουμε:

```
   cPrereq.s[0] = '\0';
```

Παρατήρηση: ▶

Μήπως έπρεπε να βάλουμε το "**explicit**" και στον δημιουργό της *CourseKey*; Όχι! Η τιμή που μας ενδιαφέρει είναι ακριβώς τύπου "`char[cCodeSz]`". Την «μεταμφιέσαμε» σε "**struct**" για τεχικούς λόγους.◀

ΔΑ, δημιουργός αντιγραφής: Τα αντικείμενα της κλάσης δεν δεσμεύουν *δυναμικώς* πόρους του συστήματος. Έτσι, δεν απαιτείται να γράψουμε δημιουργό αντιγραφής· αυτός που γράφεται αυτομάτως από τον μεταγλωττιστή είναι σωστός.
ΤΕ, τελεστής εκχώρησης: Για τον ίδιο λόγο δεν χρειάζεται να γράψουμε και τελεστή εκχώρησης.
ΚΑ, καταστροφέας: Ούτε καταστροφέα χρειάζεται να γράψουμε· εμείς όμως θα βάλουμε έναν "κενό":

```
   ~Course() { };
```

GE, συναρτήσεις "*get*":

```
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   const char* getPrereq() const {  return cPrereq.s;  }
```

Πρόσεξε την πρώτη και την τελευταία: Προτιμούμε να επιστρέψουμε τιμή τύπου "`const char*`" –δηλαδή τον «φυσικό» τύπο των μελών– και όχι `CourseKey`, που είναι «μεταμφιεσμένος».
SE, συναρτήσεις "*set*": Ξεκινούμε με τη *setCode* που τώρα έχει και ελέγχους:

```
void Course::setCode( string aCode )
{
   if ( aCode.length() != cCodeSz-1 )
      throw CourseXptn( CourseKey(cCode), "setCode", CourseXptn::keyLen,
                        aCode.c_str() );
   if ( (aCode.length() != 0) && (CourseKey(aCode) == cPrereq) )
      throw CourseXptn( CourseKey(cCode), "setCode", CourseXptn::autoRef,
                        aCode.c_str() );
   strcpy( cCode.s, aCode.c_str() );
} // Course::setCode
```

Πρόσεξε ότι

- Τώρα δεν δεχόμαστε κωδικούς που το μήκος τους δεν είναι 7.[3]
- Για να γίνει η σύγκριση "`CourseKey(aCode) == cPrereq`" θα πρέπει να έχουμε επιφορτώσει τον "`==`" για τον *CourseKey*:

```
bool operator==( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( !(lhs != rhs) );  }
```

Είναι λάθος να γράψουμε "`!(CourseKey(aCode) != cPrereq)`" αφού επιφορτώσαμε ήδη τον "`!=`"; Μια χαρά είναι! Απλώς, αυτό που γράφουμε με τον "`==`" είναι πιο κομψό.

Σημείωση:▶
Πρόσεξε τον τρόπο επιφόρτωσης του "`==`": γίνεται με βάση τον "`!=`". Δεν θα ήταν πιο γρήγορος αν γράφαμε "`return ( strcmp(lhs.s, rhs.s) == 0 )`"; Ναι, αλλά έτσι που το γράψαμε έχουμε σίγουρη συμβατότητα των δύο τελεστών. Αν πάμε για τα πιο γρήγορα μπορεί να κάνουμε κάποιο λάθος, μπορεί, αν χρειαστεί να κάνουμε κάποια αλλαγή, να ξεχάσουμε να την κάνουμε και στις δύο συναρτήσεις και άλλα τέτοια.◀

Η *setTitle()* είναι απλή:

```
void Course::setTitle( string aTitle )
```

[3] Δηλαδή αν θέλουμε να «καθαρίσουμε» ολόκληρο το αντικείμενο δεν θα μπορούμε να «καθαρίσουμε» τον κωδικό; Στην περίπτωση αυτή δεν καθαρίζουμε τα μέλη ένα προς ένα με τις "*set*"· χρησιμοποιούμε τον (ερήμην) δημιουργό χωρίς ορίσματα:

```
   c0 = Course();
```

```
{
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
} // Course::setTitle
```

Η

```
void Course::setFSem( int aFSem )
{
   if ( aFSem < 1 || 8 < aFSem)
      throw CourseXptn( cCode, "setFSem", CourseXptn::rangeError, aFSem );
   cFSem = aFSem;
} // Course::setFSem
```

ρίχνει εξαίρεση για την οποίαν θα πρέπει να κάνουμε ορισμένες αλλαγές στην *CourseXptn*:

- Δηλώνουμε άλλο ένα μέλος:

```
   int  errIntVal;
```

- και ορίζουμε άλλον έναν δημιουργό:

```
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
   {  objKey = obk;
      strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errIntVal = iv;  }
```

Η *setCompuls()* είναι πολύ απλή:

```
   void setCompuls( bool aCompuls ) { cCompuls = aCompuls; };
```

Να πούμε όμως ότι –για διαχείριση μελών τύπου `bool`– μπορεί να δεις αλλού ένα ζευγάρι μεθόδων:

```
   void setCompuls() {  cCompuls = true;  };
   void clearCompuls() {  cCompuls = false;  };
```

Η αναλλοίωτη μας λέει πώς να γράψουμε τις *setSector()* και *setCateg()*:

```
void Course::setSector( char aSector )
{
   if ( aSector != 'Υ' && aSector != 'Π' &&
        aSector != 'Μ' && aSector != 'Γ' && aSector != 'Ξ' )
      throw CourseXptn( cCode, "setSector",
                        CourseXptn::noSuchSector, aSector );
   cSector = aSector;
} // Course::setSector

void Course::setCateg( string aCateg )
{
   if ( aCateg != "ΜΓΥ" && aCateg != "ΜΕΥ" &&
        aCateg != "ΜΕ" && aCateg != "ΔΟΝ" )
      throw CourseXptn( cCode, "setCateg",
                        CourseXptn::noSuchCateg, aCateg.c_str() );
   strcpy( cCateg, aCateg.c_str() );
} // Course::setSector
```

Οι *setWH()* και *setUnits()* μοιάζουν με την *setFSem()*:

```
void Course::setWH( int aWH )
{
   if ( aWH <= 0 )
      throw CourseXptn( cCode, "setWH", CourseXptn::rangeError, aWH );
   cWH = aWH;
} // Course::setWH

void Course::setUnits( int aUnits )
{
   if ( aUnits <= 0 )
      throw CourseXptn( cCode, "setUnits", CourseXptn::rangeError, aUnits );
   cUnits = aUnits;
} // Course::setUnits
```

και θα έπρεπε να μοιάζουν περισσότερο: θα έπρεπε να υπάρχουν και περιορισμοί εκ των άνω.

Δεν θα γράψουμε *setNoOfStudents()*· ο καλύτερος τρόπος διαχείρισης του μέλους *cNoOfStudents* γίνεται με τις μεθόδους που γράψαμε αρχικώς:

```
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
```

Τώρα όμως θα πρέπει να προβλέψουμε και περίπτωση διαγραφής φοιτητή από κάποιο μάθημα. Για μια τέτοια περίπτωση θα πρέπει να γράψουμε μια:

```
void Course::delete1Student()
{
   if ( cNoOfStudents <= 0 )
      throw CourseXptn( cCode, "delete1Student", CourseXptn::noStudent );
   --cNoOfStudents;
} // Course::delete1Student
```

Η *setPrereq()* είναι κατ' αρχήν –δηλαδή σύμφωνα με όσα λέει η αναλλοίωτη– απλή:

```
void Course::setPrereq( const string& prCode )
{
   if ( prCode.length() != cCodeSz-1 && prCode.length() != 0 )
      throw CourseXptn( cCode, "setPrereq", CourseXptn::keyLen,
                        prCode.c_str() );
   if ( cCode != CourseKey("") && CourseKey(prCode) == cCode )
      throw CourseXptn( cCode, "setPrereq", CourseXptn::autoRef,
                        prCode.c_str() );
   cPrereq = CourseKey( prCode );
} // Course::setPrereq
```

Χρειάζεται όμως ιδιαίτερη μεταχείριση από το πρόγραμμα που τη χρησιμοποιεί διότι, όπως είπαμε, για να βάλουμε εκεί κάποιον κωδικό μαθήματος *«θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων.»* Με αυτό θα ασχοληθούμε στη συνέχεια.

Τέλος –και εκτός «συνταγής» πια– ας έλθουμε στις *save()* και *load()*. Η *save()* παραμένει σχεδόν όπως ήταν. Οι μοναδικές αλλαγές είναι στις γραμμές που φυλάγουν τα μέλη *cCode* και *cPrereq* που θα γίνουν:

```
// . . .
   bout.write( cCode.s, sizeof(cCode.s) );    // κωδικός μαθήματος
// . . .
   bout.write( cPrereq.s, sizeof(cPrereq.s) );   // προαπαιτούμενο
// . . .
```

Στη *load()* θα κάνουμε μεγαλύτερες αλλαγές: θα εφαρμόσουμε αυτά που είδαμε στην §21.6 για να της δώσουμε *ασφάλεια προς τις εξαιρέσεις* επιπέδου *ισχυρής εγγύησης*. Δηλαδή, αφού *«αποπειράται να αλλάξει την τιμή ενός αντικειμένου ... αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.»*

```
void Course::load( istream& bin )
{
   Course tmp;
   bin.read( tmp.cCode.s, cCodeSz );                        // κωδικός μαθήματος
   if ( !bin.eof() )
   {
      bin.read( tmp.cTitle, cTitleSz );                      // τίτλος μαθήματος
      bin.read( reinterpret_cast<char*>(&tmp.cFSem), sizeof(cFSem) );
                                                             // τυπικό εξάμηνο
      bin.read( reinterpret_cast<char*>(&tmp.cCompuls), sizeof(cCompuls) );
                                                  // υποχρεωτικό ή επιλογής
      bin.read( &tmp.cSector, sizeof(cSector) );                    // τομέας
      bin.read( tmp.cCateg, cCategSz );                           // κατηγορία
      bin.read( reinterpret_cast<char*>(&tmp.cWH), sizeof(cWH) );
                                                        // ώρες ανά εβδομάδα
      bin.read( reinterpret_cast<char*>(&tmp.cUnits), sizeof(cUnits));
                                                       // διδακτικές μονάδες
      bin.read( tmp.cPrereq.s, cCodeSz );                    // προαπαιτούμενο
```

```
      bin.read( reinterpret_cast<char*>(&tmp.cNoOfStudents),
                sizeof(cNoOfStudents) );             // αριθ. φοιτ. στο μάθημα
      if ( bin.fail() )
         throw CourseXptn( cCode, "load", CourseXptn::cannotRead );
      *this = tmp;
   } // if ( !bin.eof() ). . .
} // Course::load
```

Τι κάνουμε εδώ; Δηλώνουμε μια βοηθητική μεταβλητή *tmp* –τύπου *Course*– και αποθηκεύουμε σε αυτήν όσα διαβάζουμε. Αν όλα πάνε καλά αντιγράφουμε στο αντικείμενό μας (`*this`) αυτά που είναι αποθηκευμένα στο *tmp*. Δεν θα χρειαστεί να γράψουμε μια *swap*; Δεν είναι υποχρεωτικό! Εδώ δεν έχουμε δεσμεύσει πόρους του συστήματος, ο τελεστής εκχώρησης δουλεύει σωστά. Έτσι η "`*this = tmp`" κάνει τη δουλειά μας και μάλιστα κάπως ταχύτερα από μια "`swap(tmp)`" (αν τη γράψουμε.)

Η νέα εκδοχή της κλάσης είναι:

```
class Course
{ // version 2
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strncpy( s, aKey.c_str(), cCodeSz-1 );  s[cCodeSz-1] = '\0';  }
   }; // CourseKey
   explicit Course( string aCode="", string aTitle="" );
   ~Course() { };
// getters
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   const char* getPrereq() const {  return cPrereq.s;  }
// setters
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) {  cCompuls = aCompuls;  };
   void setSector( char aSector );
   void setCateg( string aCateg );
   void setWH( int aWH );
   void setUnits( int aUnits );
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   void delete1Student();
   void setPrereq( const string& prCode );
// other
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
```

```
}; // Course

bool operator!=( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) != 0 );  }

bool operator==( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( !(lhs != rhs) );  }
```

κσι της κλάσης εξαιρέσεων:

```
struct CourseXptn
{ // version 2
   enum { keyLen, rangeError, noSuchSector, noSuchCateg, autoRef, noCourse,
          noStudent, fileNotOpen, cannotWrite, cannotRead };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
      : objKey( obk ),  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // CourseXptn
```

Prj04.3 ... Και ο «Πίνακας Μαθημάτων»

Στο πρόγραμμα του Project 3 παραστήσαμε τον πίνακα μαθημάτων με έναν απλό δυναμικό πίνακα με στοιχεία τύπου *Course*. Η διαχείρισή του έγινε από το πρόγραμμα εφαρμογής και τις συναρτήσεις του.

Εδώ θα τον διαχειριστούμε πιο «νοικοκυρεμένα»: ως αντικείμενο κλάσης

```
class CourseCollection
{
public:
// . . .
private:
   enum { ccIncr = 30 };
   Course* ccArr;
   size_t  ccNOfCourses;
   size_t  ccReserved;
// . . .
}; // CourseCollection
```

με την οποία θα έχουμε την ευκαιρία να δούμε και μερικά νέα(;) πράγματα.

Η *CourseCollection* είναι μια κλάση για τη διαχείριση *συνόλου* που υλοποιούμε με έναν *δυναμικό πίνακα*.

Σύμφωνα με αυτά που είπαμε, θα χρειαστεί να γράψουμε δημιουργό αντιγραφής και τελεστή εκχώρησης. Εδώ όμως προσοχή: Σε οποιοδήποτε πρόγραμμα (μπορούμε να φανταστούμε) που χρησιμοποιεί αντικείμενα αυτής της κλάσης, καθένα από αυτά θα πρέπει να είναι μοναδικό. Διότι αλλιώς έχουμε το εξής πρόβλημα:

- Πόσα αντίγραφα του προγράμματος σπουδών μπορούμε να έχουμε;
- Με ποιο από αυτά θα κάνουμε τι;

Μια λύση του προβλήματος είναι η *απαγόρευση δημιουργίας αντιγράφων κλάσης CourseCollection*. Πρόσεξε πώς:

```
class CourseCollection
{
public:
```

```
// . . .
private:
   enum { ccIncr = 30 };
   Course* ccArr;
   size_t  ccNOfCourses;
   size_t  ccReserved;

   CourseCollection( const CourseCollection& rhs ) { };
   CourseCollection& operator=( const CourseCollection& rhs ) { };
// . . .
}; // CourseCollection
```

Αν στο πρόγραμμά σου (στη **main**) δώσεις:

```
   CourseCollection allCourses;
// . . .
      CourseCollection otherCollection( allCourses );
```

ο μεταγλωττιστής θα βγάλει σφάλμα:

```
'CourseCollection::CourseCollection(const CourseCollection &)' is not accessi-
ble in function main()
```

ή κάτι παρόμοιο, ενώ αν δώσεις:

```
      CourseCollection otherCollection;
// . . .
      otherCollection = allCourses;
```

θα πάρεις:

```
'CourseCollection::operator =(const CourseCollection &)' is not accessible in
function main()
```

ή κάτι παρόμοιο.

Ας έλθουμε τώρα στην *αναλλοίωτη* που περιέχει –κατ' αρχάς– τη συνθήκη διαχείρισης του δυναμικού πίνακα:

$$I_I \equiv (0 \le ccNOfCourses < ccReserved) \;\&\&\; (ccReserved \;\%\; ccIncr == 0)$$

Αλλά έχουμε και επιπλέον περιορισμούς για τα στοιχεία του πίνακα:

- Με τον πίνακα υλοποιούμε ένα *σύνολο*, ας το πούμε *Collection*. Επομένως, δεν θα πρέπει να υπάρχουν δύο ίδια στοιχεία:

$$\forall x, y: Course \bullet (x, y \in Collection \Rightarrow x \neq y)$$

- Είναι δεκτό ένα στοιχείο του πίνακα με κωδικό μαθήματος μηδενικού μήκους; Όχι! Θα πρέπει λοιπόν να έχουμε:[4]

$$\forall x: Course \bullet (x \in Collection \Rightarrow strlen(x.cCode) == cCodeSz\text{-}1)$$

- Στην §Prj04.2 λέγαμε για το προαπαιτούμενο: «*θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων. Για να μπορέσεις να βάλεις αυτόν τον περιορισμό στην αναλλοίωτη και τον αντίστοιχο έλεγχο στις μεθόδους θα πρέπει να σχεδιάσεις τις κλάσεις σου έτσι ώστε τα αντικείμενα τύπου Course να υπάρχουν μόνον μέσα σε κάποιον "πίνακα μαθημάτων"*». Εδώ λοιπόν θα πρέπει να βάλουμε:[5]

$$\forall x: Course \bullet (x \in Collection \;\&\&\; x.cPrereq \neq \texttt{""})$$
$$\Rightarrow \exists y: Course \bullet (y \in Collection \;\&\&\; y.cCode == x.cPrereq)$$

Η αναλλοίωτη είναι:

$$I \equiv I_D \;\&\&\; I_I$$

όπου το πρώτο κομμάτι είναι:

$$I_D \equiv (\forall x, y: Course \bullet (x, y \in Collection \Rightarrow x \neq y)) \;\&\&$$
$$(\forall x: Course \bullet (x \in Collection \Rightarrow strlen(x.cCode) == cCodeSz\text{-}1)) \;\&\&$$

---

[4] Στις Βάσεις Δεδομένων αυτός ο περιορισμός ονομάζεται **περιορισμός** (constraint) **ακεραιότητας οντότητας** (entity integrity)...

[5] ... και αυτός περιορισμός **ακεραιότητας αναφοράς** (referential integrity).

$$(\forall x: Course \bullet (x \in Collection \;\&\&\; x.cPrereq \neq \texttt{""})$$
$$\Rightarrow \exists y: Course \bullet (y \in Collection \;\&\&\; y.cCode == x.cPrereq))$$

Παρ' όλο που προσπαθήσαμε να γράψουμε την αναλλοίωτη κάπως αφηρημένα είναι σαφές ότι παίρνουμε υπόψη μας ότι το *cCode* είναι κλειδί.

Αν πάρουμε υπόψη μας ότι το σύνολο υλοποιείται με τα πρώτα *ccNOfCourses* στοιχεία του πίνακα ccArr η I_D γίνεται:

$$I_D \equiv (\forall j, k: [0\,..\,ccNOfCourses) \bullet (j \neq k \Rightarrow ccArr[j].cCode \neq ccArr[k].cCode)) \;\&\&$$
$$(\forall k: [0\,..\,ccNOfCourses) \bullet strlen(ccArr[k].cCode) == cCodeSz\text{-}1) \;\&\&$$
$$(\forall k: [0\,..\,ccNOfCourses) \bullet (ccArr[k].cPrereq \neq \texttt{""} \Rightarrow$$
$$\exists j: [0\,..\,ccNOfCourses) \bullet ccArr[j].cCode == ccArr[k].cPrereq))$$

Κατά τα άλλα:

- Έχουμε ερήμην δημιουργό:

```
CourseCollection::CourseCollection()
{
   try
   {
      ccReserved = ccIncr;
      ccArr = new Course[ ccReserved ];
      ccNOfCourses = 0;
   }
   catch( bad_alloc& )
   {
      throw CourseCollectionXptn( "CourseCollection",
                                  CourseCollectionXptn::allocFailed);
   }
} // CourseCollection::CourseCollection
```

- Πρέπει να γράψουμε *καταστροφέα*:

```
   ~CourseCollection()  {  delete[] ccArr;  };
```

- Θα γράψουμε συναρτήσεις "*get*"…

```
   size_t getNOfCourses() const {  return ccNOfCourses;  }
   const Course* getArr() const {  return ccArr;  }
```

- … αλλά δεν θα γράψουμε συναρτήσεις "*set*".

Θα πρέπει να γράψουμε και μεθόδους διαχείρισης των στοιχείων του πίνακα που θα μοιάζουν με αυτές που γράψαμε για τη *Route* (§20.7.2.3). Ξαναδές τι κάναμε με τον πίνακα rAllStops:

- Ο ccArr έχει μια σημαντική ομοιότητα με τον rAllStops: *και οι δύο χρησιμοποιούνται για την παράσταση συνόλων*. Έτσι, όπως είχαμε τις *find1RouteStop()*, *get1RouteStop()*, *delete-RouteStop()* – *erase1RouteStop()* και *addRouteStop()* – *insert1RouteStop()* θα πρέπει να γράψουμε τις *find1Course()*, *get1Course()*, *delete1Course()* – *erase1Course()* και *add1Course()* – *insert1Course()* αντιστοίχως.
- Τα στοιχεία του συνόλου που παριστάνει ο rAllStops έχουν, ανά δύο, διαφορετικό *sName* και διαφορετική *sDist*. Στην περίπτωση του ccArr τα στοιχεία έχουν, ανά δύο, διαφορετικό *cCode*. Ο κωδικός μαθήματος είναι άλλο ένα χαρακτηριστικό παράδειγμα *υποκατάστατου κλειδιού* (§15.5.1), όπως είναι ο αριθμός μητρώου για τη *Student*.
- Στην περίπτωση του rAllStops προτιμήσαμε να επιφορτώσουμε τον "`!=`" με σύγκριση στο μέλος *sDist* διότι είχαμε να χειριστούμε την ταξινόμηση του πίνακα σύμφωνα με την απόσταση από την αφετηρία. Αυτό μας στέρησε τη δυνατότητα χρήσης της *linSearch* –από τη `MyTmpltLib`– για αναζητήσεις στάσεων με βάση το όνομά τους. Εδώ έχουμε να κάνουμε αναζητήσεις με βάση τον κωδικό μαθήματος. Αν θέλουμε να χρησιμοποιήσουμε τη *linSearch()* μπορούμε να επιφορτώσουμε τον "`!=`" για την *Course* με σύγκριση των κωδικών μόνο. Η επιφόρτωση θα μπορούσε να γίνει, όπως στη *Date* (και τη *Student*), με μια καθολική συνάρτηση:

```
bool operator!=( const Course& a, const Course& b )
```

```
{  return ( strcmp(a.getCode(), b.getCode()) != 0 );  }
```

Αυτή περνάει από τον μεταγλωττιστή και το πρόγραμμά μας δουλεύει μια χαρά. Ας πούμε τώρα ότι –για κάποιον λόγο– θέλουμε να αλλάξουμε τη λογική σύγκρισης κλειδιών και αντικειμένων. Θα πρέπει να αλλάξουμε τις συναρτήσεις επιφόρτωσης του "`!=`" της *CourseKey* και της *Course* που υλοποιούν την ίδια λογική. Φυσικά, αυτό εισάγει στο λογισμικό μας μια δυνατότητα για μελλοντικό σφάλμα. Αυτό αποφεύγεται αν ορίσουμε:

```
bool operator!=( const Course& lhs, const Course& rhs )
{  return ( Course::CourseKey(lhs.getCode()) !=
            Course::CourseKey(rhs.getCode()) );  }
```

οπότε η λογική της σύγκρισης υλοποιείται μια φορά μόνον, για τον τύπο *CourseKey*. Πέρα από αυτό, εδώ λέμε ξεκάθαρα ότι δύο αντικείμενα τύπου *Course* είναι διαφορετικά αν και μόνον αν έχουν διαφορετικούς κωδικούς.

- Η –μικρότερης σημασίας– διαφορά του ccArr από τον rAllStops είναι η ταξινόμηση των στοιχείων του rAllStops. Αφού ο ccArr δεν είναι ταξινομημένος δεν χρειάζεται να κάνουμε πολλές μετακινήσεις στοιχείων στη διαγραφή και την εισαγωγή. Στην §20.7.2 δώσαμε ένα σχέδιο για τη *διαγραφή* σε μια τέτοια περίπτωση.

  Αρχίζουμε από την *find1Course()* που θα γίνει:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
```

όπου τώρα:

```
int CourseCollection::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( ccArr, ccNOfCourses, 0, ccNOfCourses-1, Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Πρόσεξε ότι:

- Και εδώ η *findNdx()* επιστρέφει τον δείκτη του στοιχείου (ή "-1" αν η αναζήτηση αποτύχει) ενώ η *find1Course()* είναι υλοποίηση της σχέσης "$\in$" και επιστρέφει `true` ή `false`.
- Και εδώ θα δηλώσουμε τη *findNdx()* στην περιοχή `private` με ίδιο σκεπτικό που είδαμε στην περίπτωση της *Route* (§20.7.2.3).
- Αποκλείεται να κληθεί ο δημιουργός της *Course* αν η τιμή της *code* έχει «παράνομο» μήκος.

  Ας δούμε τώρα την *erase1Course()*. Μετατρέπουμε το σχέδιο που δώσαμε στην §20.7.2:

```
void CourseCollection::delete1Course( string code )
{
   if ( υπάρχει μάθημα με κωδικό code στη θέση ndx )
   {
      αντίγραψε στη θέση ndx το τελευταίο στοιχείο του πίνακα
      --ccNOfCourses;
   }
}
```

και γράφουμε τη μέθοδο:

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // code found
   {
      ccArr[ndx] = ccArr[ccNOfCourses-1];
      --ccNOfCourses;
   }
```

```
} // CourseCollection::delete1Course
```

Όπως θα δούμε στη συνέχεια, αυτή η μέθοδος μπορεί να οδηγήσει σε παραβίαση της αναλλοίωτης. Στη συνέχεια θα τη συμπληρώσουμε και θα τη διασπάσουμε σε δύο μεθόδους.

Η *insert1Course()* μικρή σχέση έχει με την *insert1RouteStop()*· είναι πολύ απλούστερη. Αν δεν υπάρχει το στοιχείο που θέλουμε να εισαγάγουμε το αντιγράφουμε στο τέλος του πίνακα. Φυσικά, «μεγαλώνουμε» τον πίνακα αν δεν έχει χώρο για το νέο στοιχείο:

```
void CourseCollection::insert1Course( const Course& aCourse )
{
   if ( ccReserved <= ccNOfCourses+1 )
   {
      try {  renew( ccArr, ccNOfCourses, ccReserved+ccIncr );
             ccReserved += ccIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw CourseCollectionXptn( "insert1Course",
                                  CourseCollectionXptn::allocFailed );
      }
   }
   ccArr[ccNOfCourses] = aCourse;
   ++ccNOfCourses;
} // CourseCollection::insert1Course
```

Prj04.3.1 Οι Μέθοδοι *add1Course()* και *delete1Course()*

Η δουλειά που κάνει η *insert1Course()* δεν είναι αρκετή για την περίπτωσή μας αφού δεν συμμορφώνεται με την αναλλοίωτη. Έτσι, θα πρέπει να γράψουμε μια άλλη μέθοδο, ας την πούμε *add1Course()*, που θα καλεί την *insert1Course()* αφού κάνει τους εξής ελέγχους:

- Ο κωδικός μαθήματος θα πρέπει να έχει μήκος `cCodeSz-1`:

```
   if ( strlen(aCourse.getCode()) != cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::entity);
```

- Ο πίνακας δεν θα πρέπει να έχει στοιχείο με κωδικό ίδιο με αυτόν του εισαγόμενου:

```
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx >= 0 )
      throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::key,
                                  aCourse.getCode() );
```

  Αυτό όμως είναι πολύ «αυστηρό»! Αν υπάρχει το στοιχείο δεν το εισάγουμε και τελειώσαμε· η ρίψη εξαίρεσης είναι υπερβολή.

- Αν το εισαγόμενο στοιχείο έχει προαπαιτούμενο αυτό θα πρέπει να υπάρχει ήδη στον πίνακα:

```
   if ( strcmp(aCourse.getPrereq(), "") != 0 )       // υπάρχει προαπαιτούμενο
   {
      int ndx( findNdx(aCourse.getPrereq()) );
      if ( ndx < 0 )              // δεν βρέθηκε το κλειδί του προαπαιτούμενου
         throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::ref,
                                     aCourse.getPrereq() );
   }
```

- Το εισαγόμενο στοιχείο θα πρέπει να μην έχει εγγεγραμμένους φοιτητές. Δηλαδή θα πρέπει να ελέγχουμε το *cNoOfStudents*; Όχι, θα το μηδενίζουμε! Βέβαια, δεν θα πειράζουμε την παράμετρο (περνάει ως "`const Course& aCourse`") αλλά αυτό που εισάχθηκε στον πίνακα:

```
      ndx = findNdx( aCourse.getCode() );
      ccArr[ndx].clearStudents();
```

  Η *add1Course()* θα είναι:

```
void CourseCollection::add1Course( const Course& aCourse )
{
```

```
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )// υπάρχει
      {                                          // προαπαιτούμενο
         int ndx( findNdx(aCourse.getPrereq()) );
         if ( ndx < 0 )   // δεν βρέθηκε το κλειδί προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::ref,
                                        aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      insert1Course( aCourse );
      ndx = findNdx( aCourse.getCode() );
      ccArr[ndx].clearStudents();
   }
} // CourseCollection::add1Course
```

Όπως καταλαβαίνεις για να μην έχουμε δυσάρεστες εκπλήξεις θα πρέπει να «κρύψουμε» (**private**) την *insert1Code()*.

Πρόβλημα συμμόρφωσης με την αναλλοίωτη μπορεί να έχουμε και με τη διαγραφή: Αν διαγράψουμε κάποιο στοιχείο που είναι προαπαιτούμενο άλλου (-ων) στοιχείου (-ων). Θα πρέπει λοιπόν να «κρύψουμε» και την *erase1Course()* και να γράψουμε μια *delete1Course()*, που θα καλεί την *erase1Course()* αφού κάνει τους απαραίτητους ελέγχους.

Και κάτι ακόμη: όπως θα θυμάσαι από το Project 3, υπάρχει και ο *Πίνακας Δηλώσεων Μαθημάτων*. Αν το μάθημα που θέλουμε να σβήσουμε έχει *cNoOfStudents* > 0 θα πει ότι

- υπάρχουν στοιχεία του *Πίνακα Δηλώσεων* με τον κωδικό του,
- υπάρχουν αντικείμενα κλάσης *Student* με τον κωδικό του στον πίνακα μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.

Επομένως δεν μπορούμε να σβήσουμε το μάθημα!

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // υπάρχει
   {
      ccArr[ccNOfCourses] = Course();
      ccArr[ccNOfCourses].setPrereq( code ); // φρουρός
      int k(0);
      while ( strcmp(ccArr[k].getPrereq(), code.c_str()) != 0 )
         ++k;
      if ( k < ccNOfCourses )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::cannotDel,
                                     code.c_str() );
      int enrStdnt( ccArr[ndx].getNoOfStudents() );
      if ( enrStdnt > 0 ) // υπάρχουν εγγραφές φοιτητών
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::enrollRef,
                                     code.c_str(), enrStdnt );
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Αν βρούμε ότι στον πίνακα υπάρχει μάθημα με τον συγκεκριμένο κωδικό (*code*) κάνουμε μια γραμμική αναζήτηση με φρουρό στο μέλος *cPrereq*. Αν βρούμε έστω και ένα στοιχείο του πίνακα που έχει ως προαπαιτούμενο μάθημα με κωδικό *code* ρίχνουμε εξαίρεση.

Ακόμη ρίχνουμε εξαίρεση αν βρούμε `ccArr[ndx].getNoOfStudents() > 0`.

Σημείωση:▶
Αργότερα θα δούμε πώς μπορείς να χειρισθείς με άλλον τρόπο τη διαγραφή μαθήματος στο οποίο υπάρχουν εγγραφές φοιτητών.◀

Αφού οι έλεγχοι πέρασαν στη *delete1Course()*, η *erase1Course()* απλουστεύεται:

```
void CourseCollection::erase1Course( int ndx )
{
   ccArr[ndx] = ccArr[ccNOfCourses-1];
   --ccNOfCourses;
} // CourseCollection::erase1Course
```

Πρόσεξε ότι τώρα, αφού την καλούμε όταν ξέρουμε ότι το μάθημα υπάρχει σίγουρα, την τροφοδοτούμε με τον δείκτη του προς διαγραφή στοιχείου.

Prj04.3.2 Οι *save()* και *load()*

Οι *save()* και *load()* χρησιμοποιούν τις αντίστοιχες της *Course*. Πρώτα η

```
void CourseCollection::save( ofstream& bout )
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save", CourseCollectionXptn::fileNotOpen);
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( int k(0); k < ccNOfCourses; ++k )
      ccArr[k].save( bout );
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save", CourseCollectionXptn::cannotWrite);
} // CourseCollection::save
```

Η *load()* θέλει πιο πολλή προσοχή, αφού θα πρέπει να έχει *ασφάλεια προς τις εξαιρέσεις επιπέδου ισχυρής εγγύησης*. Δηλαδή, όπως λέγαμε στην §21.6, αφού «*αποπειράται να αλλάξει την τιμή ενός αντικειμένου ... αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.*»

Για να το διασφαλίσουμε θα χρησιμοποιήσουμε ένα τοπικό αντικείμενο

```
   CourseCollection tmp;
```

όπου θα αποθηκεύουμε τις τιμές που διαβάζουμε από το αρχείο. Αν όλα πάνε καλά, θα ανταλλάξουμε την τιμή του *tmp* με την τιμή του ***this**, με χρήση της

```
void CourseCollection::swap( CourseCollection& rhs )
{
   Course* sv( ccArr );
   ccArr = rhs.ccArr;  rhs.ccArr = sv;
   std::swap( ccNOfCourses, rhs.ccNOfCourses );
   std::swap( ccReserved, rhs.ccReserved );
} // CourseCollection::swap
```

Το πρώτο που κάνουμε είναι να διαβάσουμε το πλήθος των στοιχείων του πίνακα σε μια τοπική μεταβλητή:

```
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(ccNOfCourses) );
```

Η τιμή της *n* μας δίνει το πλήθος των τιμών τύπου *Course* που ακολουθούν στο αρχείο. Τις διαβάζουμε ως εξής:

```
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
```

Πρόσεξε ότι εδώ καλούμε την *insert1Course()* και όχι την *add1Course()* θεωρώντας –ως συνήθως– ότι το περιεχόμενο του μη μορφοποιημένου αρχείου δεν έχει σφάλματα.

Αν η εκτέλεση της **for** τελειώσει επειδή κάτι δεν πήγε καλά ρίχνουμε εξαίρεση. Αλλιώς δίνουμε στο αντικείμενό μας την τιμή του *tmp* με την (ασφαλή) *swap*:

```
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
```

Με την ολοκλήρωση εκτέλεσης της *load()* καταστρέφεται το *tmp* και ανακυκλώνεται η δυναμική μνήμη που έχει δεσμεύσει. Το ίδιο θα συμβεί και στην περίπτωση που η εκτέλεση της *load()* διακόπτεται επειδή ρίχνεται κάποια εξαίρεση. Εδώ έχουμε εφαρμογή της τεχνικής *RAII*.

Ολόκληρη η *load*:

```
void CourseCollection::load( ifstream& bin )
{
   CourseCollection tmp;
   unsigned int n;

   bin.read( reinterpret_cast<char*>(&n), sizeof(ccNOfCourses) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Prj04.3.3 Απώλειες Πρόσβασης

Κρύβοντας σε περιοχή **private** όλα τα αντικείμενα *Course* αχρηστεύουμε τις ανοιχτές μεθόδους τους. *Αυτά τα αντικείμενα θα παραμείνουν όπως είναι όταν τα βάζουμε για πρώτη φορά*! Δεν μπορούμε να χρησιμοποιήσουμε τη διαδικασία «*get-delete-edit-add*» που είδαμε στη *Route;* Όχι! Θα πάρουμε εξαίρεση

- αν προσπαθήσουμε να σβήσουμε ένα μάθημα που είναι προαπαιτούμενο για κάποιο άλλο·
- αν προσπαθήσουμε να σβήσουμε ένα μάθημα στο οποίο έχουν εγγραφεί φοιτητές (αυτό θα το φροντίσουμε στη συνέχεια).

Ναι, αλλά –πέρα από ανάγκες διόρθωσης λανθασμένων στοιχείων– θα χρειαστεί να αυξάνουμε (μειώνουμε) τον αριθμό των φοιτητών που γράφονται σε (διαγράφονται από) ένα μάθημα. Τι κάνουμε σε αυτήν την περίπτωση; Γράφουμε για την *CourseCollection* ενδιάμεσες μεθόδους που επιτρέπουν την χρήση των μεθόδων των αντικειμένων *Course*. Στην περίπτωσή μας θα δηλώσουμε τις:

```
   void add1Student( string code );
   void delete1Student( string code );
```

και θα τις ορίσουμε:

```
void CourseCollection::add1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
```

```
   ccArr[ndx].add1Student();
} // CourseCollection::add1Student

void CourseCollection::delete1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ccArr[ndx].delete1Student();
} // CourseCollection::delete1Student
```

Τα ονόματα είναι αυτά των αντίστοιχων μεθόδων της *Course*·αν δεν σου αρέσουν άλλαξέ τα.

Παρόμοιες ενδιάμεσες μεθόδους (της *CourseCollection*) μπορείς να γράψεις και για άλλες μεθόδους της *Course*· *μη γράψεις για τη setCode*. Να θυμάσαι ότι γενικώς, από τη στιγμή που το αντικείμενο μπήκε στη συλλογή, *το κλειδί δεν αλλάζει*. Αν αλλάξεις κάποιον κωδικό μαθήματος θα μείνουν «ξεκρέμαστα»

- τα αντικείμενα του πίνακα που έχουν το μάθημα ως προαπαιτούμενο και
- τα αντικείμενα του *Πίνακα Δήλωσης Μαθημάτων* για εγγραφή σε αυτό το μάθημα.

Prj04.3.4 Επιφορτώνουμε τον "[]";

Πιθανότατα θα αναρωτηθείς: «Αφού ένα αντικείμενο της *CourseCollection* είναι ένας πίνακας στοιχείων τύπου *Course* γιατί να μην επιφορτώσουμε τον τελεστή "`[]`";»

Ναι, ένα αντικείμενο της *CourseCollection* είναι ένας πίνακας στοιχείων τύπου *Course* αλλά «κρύβουμε» αυτό το γεγονός για να μην ενθαρρύνουμε την ανάπτυξη εφαρμογών που να στηρίζονται στην παράσταση των δεδομένων του συνόλου. Έτσι, η *get1Course()* υλοποιείται ως εξής:

```
const Course& CourseCollection::get1Course( string code ) const
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return ccArr[ndx];
} // CourseCollection::get1Course
```

Ο δείκτης *ndx* είναι τοπική μεταβλητή και το πρόγραμμα που χρησιμοποιεί αυτήν τη μέθοδο παίρνει αυτό που ζητάει δίνοντας μια τιμή τύπου *string*.

Αν θέλεις μπορείς να επιφορτώσεις τον "`[]`" ως εναλλακτικό τρόπο γραφής της *get1Course*:

```
const Course& CourseCollection::operator[]( string code ) const
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return ccArr[ndx];
} // CourseCollection::operator[]
```

Αλλά ποιο από τα παρακάτω είναι προτιμότερο:

```
   get1C = allCourses.get1Course( "EY02010" );
   get1C = allCourses["EY02010"];
```

Το δεύτερο φαίνεται λίγο «παράξενο»: μεταξύ των "`[`" και "`]`" έχουμε συνηθίσει να βλέπουμε παράσταση που δίνει έναν φυσικό αριθμό, τον δείκτη για το στοιχείο του πίνακα.

Υπάρχει και ένα άλλο πρόβλημα: Συνήθως ο τύπος του αποτελέσματος της συνάρτησης επιφόρτωσης του "`[]`" είναι *τύπος αναφοράς χωρίς το* "`const`". Έτσι μπορούμε να αλλάζουμε την τιμή του στοιχείου που μας δίνει. Εδώ δεν θέλουμε να δώσουμε τέτοια δυνατότητα.

Όπως βλέπεις, μια τέτοια επιφορτωση θα ήταν ανορθόδοξη από πολλές απόψεις και θα προτιμήσουμε να την αποφύγουμε. Αργότερα θα δούμε ότι κάτι τέτοιο γίνεται στην STL.

Prj04.4 Περί Διαγραφών

Είδαμε πιο πάνω την *CourseCollection*::*delete1Course()* και μην αμφιβάλλεις για τη συνέχια: θα δούμε και *StudentCollection*::*delete1Student()* και *StudentInCourseCollection*::*delete1StudentInCourse()*. Κάνουμε βεβαίως ελέγχους, αλλά σβήνουμε μάλλον εύκολα.

Σκέψου όμως τα εξής: τα διαγραφόμενα μπορεί να περιέχουν στοιχεία που έχουν εισαχθεί με το χέρι· αυτή είναι μια «ακριβή» διαδικασία. Τέτοια στοιχεία δεν τα σβήνουμε «τελεσιδίκως» διότι είναι πιθανό να χρειαστεί να εισαχθούν εκ νέου στους πίνακές μας (όπως είναι ή μετά από κάποια διόρθωση). Να θυμάσαι ότι γενικώς:

- ***Σπανίως σβήνουμε δεδομένα που έχουν εισαχθεί σε μια ΒΔ με το χέρι ή άλλη χρονοβόρα –και επομένως «ακριβή»– διαδικασία.***

Αυτό που μπορείς να κάνεις είναι να γράψεις τις μεθόδους διαγραφής έτσι ώστε τα αντικείμενα που θα τα αφαιρέσει από κάποιον πίνακα να μεταφέρονται σε έναν άλλον πίνακα με την ίδια (περίπου) δομή. Τι εννοούμε με το «περίπου»; Συνήθως κάθε αντικείμενο που εισάγεται στα «διαγραφέντα» συνοδεύεται από τον χρόνο της διαγραφής.

Prj04.5 Η Κλάση *Student*

Η *Student* θα είναι τώρα:

```
class Student
{
public:
// . . .
private:
   enum { sNameSz = 20 };
   enum { sIncr = 3 };
   unsigned int       sIdNum;                  // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];
   unsigned int       sWH;                     // ώρες ανά εβδομάδα
   size_t             sNoOfCourses;            // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
   size_t             sReserved;
}; // Student
```

Το βέλος *sCourses* θα δείχνει τον δυναμικό πίνακα όπου «*θα αποθηκεύονται ... οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.*»

Η –αντίστοιχη της *Student*– κλάση εξαιρέσεων, όπως –σχεδόν– την έχουμε από το Project 3 είναι:

```
struct StudentXptn
{
   enum { incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, int ev = 0 )
```

```
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Η μόνη διαφορά –προς το παρόν– είναι η εισαγωγή ενός ακόμη μελους, του *objKey* και η αλλαγή στον δημιουργό ώστε να το χειριστεί.

Τώρα, ας εφαρμόσουμε τη συνταγή μας:

- **ΑΝ**: Ας ξεκινήσουμε με την αναλλοίωτη της κλάσης:

$$0 \texttt{ <= } sIdNum \texttt{ \&\& } strlen(sSurname) < sNameSz \texttt{ \&\& } strlen(sFirstname) < sNameSz \texttt{ \&\& } sWH \texttt{ >= } 0 \texttt{ \&\&}$$
$$(\forall k: [0..sNoOfCourses) \bullet (strlen(\text{sCourses}[k]) \texttt{ == } Course\texttt{::}cCodeSz\text{-}1)) \texttt{ \&\&}$$
$$(\forall j, k: [0 .. sNoOfCourses) \bullet (j \neq k \Rightarrow \text{sCourses}[j] \neq \text{sCourses}[k])) \texttt{ \&\&}$$
$$(0 \leq sNoOfCourses < sReserved) \texttt{ \&\& } (sReserved \texttt{ \% } sIncr \texttt{ == } 0)$$

Να παρατηρήσουμε τα εξής:

1) Το *sIdNum* μπορεί να πάρει τιμή "`0`" μόνον από τον ερήμην δημιουργό σε δήλωση της μορφής:

```
   Student s1;
```

2) Για το μέλος *sIdNum* χρειαζόμαστε πιο «περιοριστική» συνθήκη· με τη "0 `<=` *sIdNum*" γίνεται δεκτός ο αριθμός μητρώου "`1000000`" που είναι μάλλον απίθανο να υπάρχει για κάποιο τμήμα.

3) Όπως στην περίπτωση του *cTitle* της *Course* έτσι και για τα *sSurname* και *sFirstname*, δεν θα ρίχνουμε εξαίρεση αν μας έλθει τιμή με μήκος *sNameSz* ή μεγαλύτερο. Απλώς θα αποθηκεύουμε τους πρώτους *sNameSz*-1 χαρακτήρες.

- **ΔΕ**: Θα χρειαστούμε φυσικά έναν ερήμην δημιουργό.

```
Student:: Student( int aIdNum )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
   sWH = 0;
   try  {  sCourses = new Course::CourseKey[ sIncr ];  }
   catch( bad_alloc )
   {  throw StudentXptn( sIdNum, "Student", StudentXptn::allocFailed );  }
   sReserved = sIncr;
   sNoOfCourses = 0;
}; // Student()
```

Πρόσεξε τη διαφορά των δύο εξαιρέσεων:

- Η πρώτη ρίχνεται αν η *aIdNum* έχει παράνομη τιμή οπότε και δεν μπορούμε να βάλουμε τιμή στο *sIdNum* που είναι κλειδί.
- Στη δεύτερη περίπτωση το *sIdNum* έχει τιμή (που πήρε από την *aIdNum*.)

Prj04.5.1 Ο «Κανόνας των Τριών»

- **ΔΑ**: Θα χρειαστούμε και έναν δημιουργό αντιγραφής; Ο πίνακας μαθημάτων θα είναι δυναμικός. Έτσι, τα αντικείμενα της κλάσης μας δεσμεύουν πόρους του συστήματος (μνήμη) και θα χρειαστεί να γράψουμε δημιουργό αντιγραφής αφού αυτός που μας δίνει αυτομάτως ο μεταγλωττιστής δεν είναι σωστός.
- **ΤΕ**, **ΚΑ**: Για τον ίδιο λόγο (δυναμικός πίνακας) θα χρειαστεί να γράψουμε τελεστή εκχώρησης και καταστροφέα.

  Ο καταστροφέας είναι απλός:

```
   ~Student() {  delete[] sCourses;  };
```

  Και ο δημιουργός αντιγραφής:

```
Student::Student( const Student& rhs )
```

```
{
   sIdNum = rhs.sIdNum;
   strcpy( sSurname, rhs.sSurname );
   strcpy( sFirstname, rhs.sFirstname );
   sWH = rhs.sWH;
   try  {  sCourses = new Course::CourseKey[ rhs.sReserved ];  }
   catch( bad_alloc )
   {  throw StudentXptn( sIdNum, "Student", StudentXptn::allocFailed );  }
   sReserved = rhs.sReserved;
   for ( int k(0); k < rhs.sNoOfCourses; ++k )
      sCourses[k] = rhs.sCourses[k];
   sNoOfCourses = rhs.sNoOfCourses;
}; // Student( const Student& rhs )
```

Και ο (αντιγραφικός) τελεστής εκχώρησης:

```
Student& Student::operator=( const Student& rhs )
{
   if ( &rhs != this )
   {
      try {  Student tmp( rhs );
             swap( tmp );  }
      catch( StudentXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
}; // Student( const Student& rhs )
```

όπου

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );

   std::swap( sWH, rhs.sWH );
   std::swap( sNoOfCourses, rhs.sNoOfCourses );

   Course::CourseKey* svck( sCourses );
   sCourses = rhs.sCourses;  rhs.sCourses = svck;

   std::swap( sReserved, rhs.sReserved );
} // Student::swap
```

Prj04.5.2 Μέθοδοι "*get*" και "*set*"

GE: Θα χρειαστούμε μεθόδους *"get"* για όλα τα μέλη εκτός από το *sReserved* («μυστικό» της υλοποίησης):

```
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
   unsigned int getWH() const {  return sWH;  }
   unsigned int getNoOfCourses() const { return sNoOfCourses; }
   const Course::CourseKey* getCourses() const { return sCourses;}
```

SE: Θα χρειαστούμε μεθόδους *"set"* για όλα τα μέλη –κατ' αρχήν– εκτός από το *sReserved*.

Πρώτα η

```
void Student::setIdNum( int aIdNum )
{
```

```
   if ( aIdNum <= 0 )
      throw StudentXptn( sIdNum, "setIdNum", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
} // Student::setIdNum
```

Πρόσεξε ότι εδώ δεν επιτρέπουμε τιμή "0", που επιτρέπεται (μόνο) στον ερήμην δημιουργό.

Άλλες δύο, οι *setSurname()* και *setFirstname()*, είναι απλές:

```
void Student::setSurname( string aSurname )
{
   strncpy( sSurname, aSurname.c_str(), sNameSz-1 );
   sSurname[sNameSz-1] = '\0';
} // Student::setSurname

void Student::setFirstname( string aFirstname )
{
   strncpy( sFirstname, aFirstname.c_str(), sNameSz-1 );
   sFirstname[sNameSz-1] = '\0';
} // Student::setSurname
```

Στην περίπτωση (μάλλον απίθανη) που θα έλθει κάποιο επώνυμο ή όνομα με περισσότερους από 19 χαρακτήρες θα αντιγραφούν μόνον οι πρώτοι 19 και φυσικά δεν θα ρίξουμε εξαίρεση. Πάντως, καλό θα είναι να μάθει ο χρήστης του προγράμματος-πελάτη, με κάποιον τρόπο, ότι μια μέθοδος της κλάσης αλλοίωσε τα δεδομένα του!

Θα μπορούσαμε να γράψουμε και *μια* μέθοδο (*setCourses*) που θα δίνει τιμή στο *sNoOfCourses* και τον πίνακα sCourses. Δεν θα το κάνουμε· αν θέλεις κάνε το σαν άσκηση. Πάντως, μια τέτοια μέθοδος δεν θα ήταν και τόσο χρήσιμη αφού στη συνέχεια θα γράψουμε τις *insert1Course()* και *add1Course()*.

Στις μεθόδους "*set*" θα περιλάβουμε και την

```
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
```

που είχαμε στο Project 3. Αυτή παραμένει όπως ήταν! Και ο πίνακας; Ο πίνακας παραμένει όπως είναι, η τιμή του *sReserved* δεν αλλάζει, αλλά η τιμή του *sNoOfCourses* (== 0) μας λέει ότι δεν υπάρχουν στοιχεία για επεξεργασία. Θα μπορούσαμε βεβαίως να επιστρέψουμε τη μνήμη του δυναμικού πίνακα και να πάρουμε νέα με *sIncr* στοιχεία μόνον. Δεν αξίζει τον κόπο...

Prj04.5.3 Μέθοδοι για τα Στοιχεία του Πίνακα

Κατ' αρχήν μπορούμε να χρησιμοποιήσουμε ως οδηγό τις αντίστοιχες μεθόδους της *CourseCollection* –και αυτό θα κάνουμε– εστιάζοντας την προσοχή μας στις διαφορές.

Η βασική μέθοδος θα είναι η *findNdx()* που τροφοδοτείται με έναν κωδικό μαθήματος και μας επιστρέφει τη θέση του στον πίνακα ή, αν δεν υπάρχει, "–1". Την αντιγράφουμε (σχεδόν) από την *CourseCollection*:

```
int Student::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( sCourses, sNoOfCourses, 0, sNoOfCourses-1,
                       Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Η *find1Course()* θα είναι:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
```

Φυσικά, κάθε πρόγραμμα-πελάτης μπορεί να χρησιμοποιεί μόνον τη δεύτερη ενώ η πρώτη θα είναι «κρυμμένη» στην περιοχή `private`.

Θα γράψουμε *get1Course()*; Δεν απαγορεύεται αλλά δεν έχει και νόημα, αφού θα είναι μια συνάρτηση που θα τροφοδοτείται με έναν κωδικό και θα επιστρέφει τον ίδιο κωδικό.

Θα υλοποιήσουμε τώρα την *insert1Course()*:

```
void Student::insert1Course( const Course::CourseKey& aCode )
{
   if ( sReserved <= sNoOfCourses+1 )
   {
      try {  renew( sCourses, sNoOfCourses, sReserved+sIncr );
             ccReserved += ccIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw StudentXptn( sIdNum, "insert1Course",
                                    StudentXptn::allocFailed );
      }
   }
   sCourses[sNoOfCourses] = aCode;
   ++sNoOfCourses;
} // Student::insert1Course
```

Πρόσεξε όμως ότι η μέθοδος αυτή

- Δεν εξασφαλίζει τη μοναδικότητα των κωδικών στον πίνακα.
- Δεν ενημερώνει το μέλος *sWH*.
- Δεν ελέγχει αν υπάρχει στον πίνακα μαθημάτων μάθημα με τέτοιον κωδικό (*ακεραιότητα αναφοράς*).

Για να αποτρέψουμε απρόσεκτη –και εν δυνάμει καταστροφική– χρήση της μεθόδου θα την κρύψουμε, όπως κάναμε και με τη *insert1Course()* της *CourseCollection*· για εξωτερική χρήση θα ξαναγράψουμε την *add1Course()* έτσι που να αντιμετωπίζει τα πρώτα δύο προβλήματα. Αργότερα θα αντιμετωπίσουμε και το τρίτο πρόβλημα.

```
void Student::add1Course( const Course& oneCourse )
{
   if ( findNdx(oneCourse.getCode()) < 0 )
   {
      insert1Course( Course::CourseKey(oneCourse.getCode()) );
      sWH += oneCourse.getWH();
   }
} // Student::add1Course
```

Πρόσεξε ότι, παρ' όλο που στην *insert1Course()* θα περάσουμε μόνον τον κωδικό, τροφοδοτούμε τη μέθοδο με ολόκληρο το αντικείμενο κλάσης *Course*· και αυτό διότι στην επόμενη γραμμή παίρνουμε από αυτό και τις ώρες εβδομαδιαίας διδασκαλίας (*getWH()*).

Με παρόμοια λογική γράφουμε και τις:

```
void Student::delete1Course( const Course& oneCourse )
{
   int ndx( findNdx(oneCourse.getCode()) );
   if ( ndx >= 0 )  // υπάρχει
   {
      erase1Course( ndx );
      sWH -= oneCourse.getWH();
   }
} // Student::delete1Course

void Student::erase1Course( int ndx )
{
   sCourses[ndx] = sCourses[sNoOfCourses-1];
   --sNoOfCourses;
} // Student::erase1Course
```

Prj04.5.4 Φύλαξη και Φόρτωση

Για τη φύλαξη ενός αντικειμένου *Student* γράφουμε τη μέθοδο:

```
void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
   bout.write( sFirstname, sizeof(sFirstname) );
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
   bout.write( reinterpret_cast<const char*>(&sNoOfCourses),
               sizeof(sNoOfCourses) );
   for ( int k(0); k < sNoOfCourses; ++k )
      bout.write( sCourses[k].s, Course::cCodeSz );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save
```

που διαφέρει από αυτήν της αρχικής εκδοχής στο ότι εδώ υπάρχει η **for** που φυλάγει τους κωδικούς των μαθημάτων.

Η *load()* διαφέρει πιο πολύ από την αρχική.

```
void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      bin.read( reinterpret_cast<char*>(&tmp.sWH), sizeof(sWH) );
      bin.read( reinterpret_cast<char*>(&tmp.sNoOfCourses),
                sizeof(sNoOfCourses) );
      if ( tmp.sNoOfCourses >= tmp.sReserved )
      {
         delete[] tmp.sCourses;
         try
         {
            tmp.sCourses =
                    new Course::CourseKey[((tmp.sNoOfCourses/sIncr)+1)*sIncr];
            tmp.sReserved = ((tmp.sNoOfCourses/sIncr)+1)*sIncr;
         }
         catch( bad_alloc )
         {
            throw StudentXptn( tmp.sIdNum, "load", StudentXptn::allocFailed );
         }
      }
      for ( int k(0); k < tmp.sNoOfCourses; ++k )
         bin.read( tmp.sCourses[k].s, Course::cCodeSz );
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load
```

Εδώ κάνουμε τη φόρτωση με κάπως διαφορετικό τρόπο από αυτόν που χρησιμοποιήσαμε στην *CourseCollection*. Και στις δύο περιπτώσεις αποθηκεύουμε αυτά που διαβάζουμε σε ένα τοπικό αντικείμενο (*tmp*) και τελικώς ανταλλάσσουμε την τιμή του με το `*this`:

- Εκεί, διαβάζαμε τα αντικείμενα τύπου *Course* και τα βάζαμε στον πίνακα με την *insert1Course()*. Η μέθοδος αυτή μετράει τα στοιχεία του πίνακα (αυξάνει τον *sNoOfCourses*). Το πλήθος των στοιχείων που διαβάστηκε από το αρχείο αποθηκεύεται σε μια τοπική μεταβλητή *n*. Ακόμη, η *insert1Course()*, παίρνει και τη δυναμική μνήμη όταν χρειαστεί.
- Εδώ, το πλήθος των στοιχείων που διαβάζεται από το αρχείο αποθηκεύεται στο μέλος του αντικειμένου *tmp*.*sNoOfCourses*. Αν χρειάζεται, παίρνουμε την απαραίτητη δυναμι-

κή μνήμη, αφού πρώτα ανακυκλώσουμε όση μνήμη είχε ήδη δεσμεύσει το αντικείμενο. Στη συνέχεια αυτά που διαβάζουμε απόοθηκεύονται κατ' αυθείαν στα στοιχεία του πίνακα.

Εκτός από τις *save()* και *load()*, στην πρώτη εκδοχή της κλάσης, είχαμε και μια άλλη μέθοδο για ανάγνωση από αρχείο, τη *readFromText()*. Θα την κρατήσουμε και εδώ αλλά θα τις δώσουμε πιο «ρεαλιστικό» όνομα: *readPartFromText()*. Θα της κάνουμε όμως και μια άλλη αλλαγή: δεν θα αλλάζει την τιμή του αντικειμένου σε περίπτωση που θα ρίξει εξαίρεση:

```
void Student::readPartFromText( istream& tin )
{
   string line;
   getline( tin, line, '\n' );
   if ( !tin.eof() )
   {
      Student tmp;
      size_t t1Pos( line.find("\t") );
      if ( t1Pos >= line.length() )
         throw StudentXptn( sIdNum, "readPartFromText",
                            StudentXptn::incomplete );
      string str1( line.substr(0, t1Pos) );
      int    iStr1( atoi(str1.c_str()) );
'     try {  tmp.setIdNum( iStr1 );  }
      catch( StudentXptn& x )
      {  strcpy( x.funcName, "readPartFromText" );
         throw;  }
      size_t t2Pos( line.find("\t", t1Pos+1) );
      if ( t2Pos >= line.length() )
         throw StudentXptn( tmp.sIdNum, "readPartFromText",
                            StudentXptn::incomplete );
      tmp.setSurname( line.substr(t1Pos+1, t2Pos-t1Pos-1) );
      tmp.setFirstname( line.substr(t2Pos+1) );
      swap( tmp );
   } // if ( !tin.eof . . .
} // Student::readPartFromText
```

Prj04.5.5 Η Κλάση *Student*

Να πώς έγινε τώρα η κλάση *Student*:

```
class Student
{ // version 2
public:
// constructors, destructor
   explicit Student( int aIdNum=0 );
   Student( const Student& rhs );
   ~Student() {  delete[] sCourses;  };
// copy assignement
   Student& operator=( const Student& rhs );
// getters
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
   unsigned int getWH() const {  return sWH;  }
   unsigned int getNoOfCourses() const { return sNoOfCourses; }
   const Course::CourseKey* getCourses() const
                                          {  return sCourses;  }
// setters
   void setIdNum( int aIdNum );
   void setSurname( string aSurname );
   void setFirstname( string aFirstname );
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
// 1 Course methods
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
   void add1Course( const Course& oneCourse );
```

```
   void delete1Course( const Course& oneCourse );
// other methods
   void swap( Student& rhs );
   void readPartFromText( istream& tin );
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   enum { sNameSz = 20 };
   enum { sIncr = 3 };
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
   unsigned int       sReserved;

   unsigned int countTabs( string aLine );
   int findNdx( const string& code ) const;
   void insert1Course( const Course::CourseKey& aCode );
   void erase1Course( int ndx );
}; // Student
```

Από την προηγούμενη εκδοχή έχουμε και τις

```
bool operator!=( const Student& lhs, const Student& rhs )
{  return (lhs.getIdNum() != rhs.getIdNum());  }

bool operator==( const Student& lhs, const Student& rhs )
{  return !(lhs != rhs);  }
```

Η κλάση εξαιρέσεων είναι:

```
struct StudentXptn
{ // version 2
   enum { allocFailed, outOfRange, incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, const char* sv="" )
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentXptn( int obk, const char* mn, int ec, int ev )
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Prj04.6 Το «Μητρώο Φοιτητών»

Παίρνουμε τα αρχεία `CourseCollection.h` και `CourseCollection.cpp` και με τη *"find-and-replace"* του κειμενογράφου αντικαθιστούμε το "`Course`" με "`Student`". Αυτό που παίρνουμε είναι –σχεδόν– η κλάση *StudentCollection*. Κάθε αντικείμενό της είναι ένα σύνολο αντικειμενων *Student* υλοποιημένο με δυναμικό πίνακα. Μετά από μερικές διορθώσεις έχουμε:

```
class StudentCollection // version 1
{
public:
   StudentCollection();
   ~StudentCollection()  {  delete[] scArr;  };
// getters
   size_t getNOfStudents() const {  return scNOfStudents;  }
   const Student* getArr() const {  return scArr;  }
```

```
// 1 Student
   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) >= 0 );  };
   void delete1Student( int aIdNum );
   void add1Student( const Student& aStudent );
   const Student& get1Student( int aIdNum ) const;
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swap( StudentCollection& rhs );
private:
   enum { scIncr = 30 };
   Student* scArr;
   size_t   scNOfStudents;
   size_t   scReserved;

   StudentCollection( const StudentCollection& rhs ) { };
   StudentCollection& operator=( const StudentCollection& rhs ){};
   void erase1Student( int ndx );
   void insert1Student( const Student& aStudent );
   int findNdx( int aIdNum ) const;
}; // StudentCollection
```

Όπως στην *CourseCollection* έτσι και εδώ, προσπαθούμε να δυσκολέψουμε τη δημιουργία αντιγράφων (πόσα μητρώα φοιτητών θα υπάρχουν;) αχρηστεύοντας και «κρύβοντας» δημιουργό αντιγραφής και τελεστή εκχώρησης.

Θα εστιάσουμε την προσοχή μας στις δύο μεθόδους που έχουν διαφορές από τις αντίστοιχες της *CourseCollection*: τη *delete1Student()* και την *add1Student()*. Η πρώτη είναι απλή:

```
void StudentCollection::delete1Student( int aIdNum )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx >= 0 )  // υπάρχει
   {
      if ( scArr[ndx].getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student
```

αφού αρνούμαστε να συζητήσουμε διαγραφή στην περίπτωση που ο φοιτητής είναι γραμμένος σε μαθήματα· όπως στην *CourseCollection*::*delete1Course()*, αρνούμαστε να διαγράψουμε ένα μάθημα στο οποίο είναι γραμμένοι φοιτητές. Αργότερα θα δούμε πώς μπορείς να γράψεις μια πληρέστερη μέθοδο.

Η δεύτερη έχει κληρονομήσει ένα πρόβλημα από τη *Student*· το πρόβλημα της *insert1Course()* που δεν αντιμετώπισε ούτε η *add1Course()*: «*Δεν ελέγχει αν υπάρχει στον πίνακα μαθημάτων μάθημα με τέτοιον κωδικό (ακεραιότητα αναφοράς).*» Τώρα όμως θα πρέπει να το λύσουμε: δεν θα πρέπει να εισάγουμε στο «μητρώο φοιτητών» αντικείμενα αν δεν σιγουρεύουμε ότι όλοι οι κωδικοί μαθημάτων αντιστοιχούν σε μαθήματα του πίνακα μαθημάτων (αν μας δίδεται).

Τι θα κάνουμε; Να βάλουμε τον πίνακα μαθημάτων μέσα στη *StudentCollection*; Όχι! Ο πίνακας μαθημάτων είναι αυθύπαρκτος και χρήσιμος και για άλλες δουλειές. Μια σαφώς καλύτερη λύση είναι να βάλουμε ένα βέλος προς τον πίνακα μαθημάτων:

```
   CourseCollection* scPAllCourses;
```

που του δίνουμε αρχική τιμή στον δημιουργό:

```
StudentCollection::StudentCollection()
{
   try
   {
      scReserved = scIncr;
```

```
      scArr = new Student[ scReserved ];
      scNOfStudents = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentCollectionXptn( "StudentCollection",
                                   StudentCollectionXptn::allocFailed);
   }
   scPAllCourses = 0;
} // StudentCollection::StudentCollection
```

Για να μπορούμε να χειριστούμε την τιμή του βέλους γράφουμε μεθόδους:

```
CourseCollection* getPAllCourses() const {  return scPAllCourses;  }
void setPAllCourses( CourseCollection* pCourses )
{  scPAllCourses = pCourses;  }
```

Μετά απο'αυτά, αν στο πρόγραμμά μας έχουμε δηλώσει:

```
CourseCollection allCourses;
StudentCollection allStudents;
```

μπορούμε να δώσουμε:

```
allStudents.setPAllCourses( &allCourses );
```

Ας γράψουμε τώρα την *add1Student()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllCourses != 0 )
      {
         const Course::CourseKey* aStCourses( aStudent.getCourses() );
         for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
         {
           if ( !scPAllCourses->find1Course(aStCourses[k].s) )
              throw StudentCollectionXptn( "add1Student",
                                           StudentCollectionXptn::ref,
                                           aStCourses[k].s );
         }
      }
      insert1Student( aStudent );
   }
} // StudentCollection::add1Student
```

Τα βασικά σημεία:

- Ο έλεγχος για την μοναδικότητα του κλειδιού (αντικειμένου) γίνεται οπωσδήποτε (`findNdx(aStudent.getIdNum()) < 0`).
- Ο έλεγχος των κωδικών των μαθημάτων γίνεται μόνο στην περίπτωση που έχουμε «συνδέσει» πίνακα μαθημάτων (`scPAllCourses != 0`).
- Ενώ διασχίζουμε με τη **for** τον πίνακα κωδικών μαθημάτων aStCourses θα σταματήσουμε μόλις (αν) βρούμε τον πρώτο κωδικό που δεν υπάρχει στον πίνακα μαθημάτων.

Όπως στην *CourseCollection* έχουμε χάσει την πρόσβαση προς τις μεθόδους των στοιχείων (τύπου *Course*) του πίνακα έτσι και εδώ έχουμε χάσει την πρόσβαση προς τις μεθόδους των στοιχείων (τύπου *Student*) του πίνακα. Για να την ανακτήσουμε θα πρέπει να γράψουμε ενδιάμεσες συναρτήσεις. Τέτοιες συναρτήσεις θα πρέπει να γράψουμε τουλάχιστον για τις *add1Course()* και *delete1Course()*. Εδώ όμως χρειάζεται προσοχή: για κάθε στοιχείο του πίνακα sCourses υπάρχει ένα στοιχείο στον *Πίνακα Δηλώσεων Μαθημάτων*· έχουμε δηλαδή *πλεονασμό*. Και ένα πρόβλημα του πλεονασμού είναι το εξής: οι ενημερώσεις (εισαγωγές – διαγραφές) στους δύο πίνακες πρέπει να γίνονται με συνέπεια. Οι *add1Course()* και *delete1Course()* της *Student* δεν έχουν αυτην την ιδιότητα. Θα επανέλθουμε...

Prj04.6.1 Φύλαξη, Φόρτωση και Ευρετήριο

Κατ'αρχήν θα μπορούσαμε να πάρουμε τις *save()* και *load()* από αυτές της *CourseCollection* με «*find-and-replace*» αλλά μια νέα απαίτηση αυτής της δεύτερης εκδοχής του προβλήματος είναι:

> «*Για να ανακτήσουμε τη δυνατότητα κατ' ευθείαν πρόσβασης στα αντικείμενα του αρχείου θα χρειαστούμε ένα* **ευρετήριο** *(index) με στοιχεία τύπου*

```
struct IndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
}; // IndexEntry
```

> *Το πρόγραμμά μας θα πρέπει να οργανώνει αυτά τα στοιχεία σε έναν πίνακα που θα τον φυλάγει σε ένα (μη-μορφοποιημένο) αρχείο* `students.ndx`.»

Να υλοποιήσουμε τα παραπάνω *μέσα* στη *StudentCollection;* Όχι! Ένα αντικείμενο της *StudentCollection* είναι ένα σύνολο που υλοποιείται στη μνήμη. Ο index είναι εργαλείο για τον χειρισμό μιας *StudentCollection* που έχει φυλαχθεί σε αρχείο.

Ο index δεν θα ανήκει σε αντικείμενο και θα έρχεται ως παράμετρος στη *save()*:

```
void StudentCollection::save( ofstream& bout, IndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      index[k].sIdNum = scArr[k].getIdNum();
      index[k].loc = bout.tellp();
      scArr[k].save( bout );
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save
```

Όταν έρχεται η σειρά για φύλαξη στο αρχείο του `scArr[k]` φυλάγουμε στο αντίστοιχο στοιχείο του ευρετηρίου τον αριθμό μητρώου (*sIdNum*) και τη θέση στο αρχείο (*loc*) όπου θα αρχίσει η αποθήκευση του `scArr[k]`. Έτσι, όταν τελειώσει η φύλαξη του `scArr` ο `index` «ξέρει» που αρχίζει η αποθήκευση του κάθε αντικειμένου-στοιχείου του `scArr`.

Η υπόλοιπη διαχείριση του `index` είναι υποχρέωση του προγράμματος-πελάτη της κλάσης.

Για τη φόρτωση δεν χρειάζεται το ευρετήριο:

```
void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(scNOfStudents) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.insert1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
```

```
} // StudentCollection::load
```

Τι είναι η *swapArr()*; Μια μέθοδος που αντιμεταθέτει τους πίνακες δύο αντικειμένων *StudentCollection*:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   std::swap( scArr, rhs.scArr );
   std::swap( scNOfStudents, rhs.scNOfStudents );
   std::swap( scReserved, rhs.scReserved );
} // StudentCollection::swapArr
```

Και γιατί δεν γράφουμε μια *swap()* όπως όλοι οι «κανονικοί» άνθρωποι (προγραμματιστές); Διότι μια τέτοια μέθοδος θα πρέπει να αντιμεταθέτει και τις τιμές των *scPAllCourses* πράγμα όχι και τόσο βολικό.[6]

Prj04.7 Η Κλάση *StudentInCourse*

Θα ξαναγράψουμε τη *StudentInCourse* με βάση τη συνταγή μας και παίρνοντας υπόψη μας τις αλλαγές που κάναμε στην *Course*:

```
class StudentInCourse
{
public:
// . . .
private:
   unsigned int      sicSIdNum;  // αριθμός μητρώου
   Course::CourseKey sicCCode;   // κωδικός μαθήματος
   float             sicMark;    // βαθμός στο μάθημα
}; // StudentInCourse
```

Δύο αντικείμενα αυτής της κλάσης, *a* και *b*, είναι ίσα αν και μόνον αν αναφέρονται στον ίδιο φοιτητή (*a.sicSIdNum* == *b.sicSIdNum*) για το ίδιο μάθημα (*a.sicCCode* == *b.sicCCode*). Δηλαδή, το ζεύγος (*sicSIdNum, sicCCode*) είναι *κλειδί*. Για χρήση στην κλάση εξαιρέσεων (αλλά και αλλού) ορίζουμε στην περιοχή "**public**" της κλάσης τον τύπο:

```
   struct SICKey
   {
      unsigned int      sIdNum;
      Course::CourseKey CCode;
      explicit SICKey( int aIdNum=0, string aCCode="" )
      {  sIdum = aIdNum;  CCode = Course::CourseKey(aCCode);  }
   }; // SICKey
```

και τη

```
   SICKey getKey() const
   {  return SICKey( sicSIdNum, sicCCode.s );  }
```

Για τον τύπο αυτόν μπορούμε να επιφορτώσουμε τον "!=":

```
bool operator!=( const StudentInCourse::SICKey& lhs,
                 const StudentInCourse::SICKey& rhs )
{  return ( (lhs.sIdNum != rhs.sIdNum) || (lhs.CCode != rhs.CCode) );  }
```

και στη συνέχεια για τον *StudentInCourse*:

```
bool operator!=( const StudentInCourse& lhs, const StudentInCourse& rhs )
{  return ( lhs.getKey() != rhs.getKey() );  }
```

Ας εφαρμόσουμε τώρα τη συνταγή μας.

AN: Η αναλλοίωτη της κλάσης θα είναι:

$$0 \texttt{ <= } sicSIdNum \texttt{ \&\&}$$
$$((strlen(sicCCode.s) \texttt{ == } Course\texttt{::}cCodeSz\text{-}1) \texttt{ || } (sicCCode.s) \texttt{ == } 0)) \texttt{ \&\&}$$
$$0 \texttt{ <= } sicMark \texttt{ <= } 10$$

[6] Κάτι δεν πάει καλά με τη σχεδίαση…

ΔΕ: Ο ερήμην δημιουργός (με αρχική τιμή) δηλώνεται ως:

```
   StudentInCourse( int aIdNum=0, string aCCode="" );
```

και ορίζεται:

```
StudentInCourse::StudentInCourse( int aIdNum, string aCCode )
{
   if ( aIdNum < 0 )
      throw StudentInCourseXptn( SICKey(), "StudentInCourse",
                                 StudentInCourseXptn::negIdNum, aIdNum );
   sicSIdNum = aIdNum;
   if ( aCCode.length() != Course::cCodeSz-1 &&
        aCCode.length() != 0 )
      throw StudentInCourseXptn( SICKey(sicSIdNum), "StudentInCourse",
                                 StudentInCourseXptn::keyLen, aCCode.c_str());
   sicCCode = Course::CourseKey( aCCode );
   sicMark = 0;
} // StudentInCourse::StudentInCourse
```

ΔΑ, ΤΕ, ΚΑ: Ο δημιουργός αντιγραφής και ο τελεστής εκχώρησης που μας δίνει ο μεταγλωττιστής κάνουν σωστά τις δουλειές τους. Πάντως, κατά τη συνήθειά μας, θα γράψουμε έναν κενό καταστροφέα:

```
   ~StudentInCourse() { };
```

GE: Οι μέθοδοι "*get*", εκτός από τη *getKey()* που είδαμε παραπάνω:

```
   unsigned int getSIdNum() const {  return sicSIdNum;  }
   const char* getCCode() const {  return sicCCode.s;  }
   float getMark() const {  return sicMark;  }
```

SE: Και οι μέθοδοι "*set*":

```
void StudentInCourse::setSIdNum( int aIdNum )
{
   if ( aIdNum <= 0 )
      throw StudentInCourseXptn( getKey(), "setSIdNum",
                                 StudentInCourseXptn::negIdNum, aIdNum );
   sicSIdNum = aIdNum;
} // StudentInCourse::setSIdNum

void StudentInCourse::setCCode( string aCCode)
{
   if ( aCCode.length() != Course::cCodeSz-1 )
      throw StudentInCourseXptn( getKey(), "setCCode",
                                 StudentInCourseXptn::keyLen, aCCode.c_str());
   sicCCode = Course::CourseKey( aCCode );
} // StudentInCourse::setCCode

void StudentInCourse::setMark( float aMark )
{
   if ( aMark < 0 || 10 < aMark )
      throw StudentInCourseXptn( getKey(), "setMark",
                                 StudentInCourseXptn::rangeError, aMark );
   sicMark = aMark;
} // StudentInCourse::setMark
```

Τη *setIdNumCCode()* που είχαμε θα την κρατήσουμε; Όχι, διότι, όπως πιθανότατα μαντεύεις, στη συνέχεια θα γράψουμε και μια κλάση *StudentInCourseCollection* όπου και θα γίνονται αυτά που έκανε η *setIdNumCCode()*.

Άλλες Μέθοδοι: Η *save()* είναι αυτή της αρχικής εκδοχής με μικρές προσαρμογές στις εξαιρέσεις και στη φύλαξη του κωδικού μαθήματος:

```
void StudentInCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentInCourseXptn( getKey(), "save",
                                 StudentInCourseXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sicSIdNum), sizeof(sicSIdNum) );
   bout.write( sicCCode.s, Course::cCodeSz ); // κωδικός μαθήματος
```

```
   bout.write( reinterpret_cast<const char*>(&sicMark), sizeof(sicMark) );
   if ( bout.fail() )
      throw StudentInCourseXptn( getKey(), "save",
                                 StudentInCourseXptn::cannotWrite );
} // StudentInCourse::save
```

Για τη δεύτερη εφαρμογή θα χρειαστούμε και μια *load()*:

```
void StudentInCourse::load( istream& bin )
{
   StudentInCourse tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sicSIdNum), sizeof(sicSIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sicCCode.s, Course::cCodeSz );        // κωδικός μαθήματος
      bin.read( reinterpret_cast<char*>(&tmp.sicMark), sizeof(sicMark) );
      if ( bin.fail() )
         throw StudentInCourseXptn( getKey(), "load",
                                    StudentInCourseXptn::cannotWrite );
      *this = tmp;
   }
} // StudentInCourse::load
```

Η κλάση θα είναι:

```
class StudentInCourse
{
public:
   struct SICKey
   {
      unsigned int      sIdNum;
      Course::CourseKey CCode;
      explicit SICKey( int aIdNum=0, string aCCode="" )
      {  sIdNum = aIdNum;  CCode = Course::CourseKey(aCCode);  }
   }; // SICKey
   explicit StudentInCourse( int aIdNum=0, string aCCode="" );
   ~StudentInCourse() { };
// getters
   unsigned int getSIdNum() const {  return sicSIdNum;  }
   const char* getCCode() const {  return sicCCode.s;  }
   float getMark() const {  return sicMark;  }
   SICKey getKey() const
   {  return SICKey( sicSIdNum, sicCCode.s );  }
// setters
   void setSIdNum( int aIdNum );
   void setCCode( string aCode);
   void setMark( float aMark );
// other
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   unsigned int      sicSIdNum;  // αριθμός μητρώου
   Course::CourseKey sicCCode;   // κωδικός μαθήματος
   float             sicMark;    // βαθμός στο μάθημα
}; // StudentInCourse
```

Ακόμη, έχουμε επιφορτώσει τον "`!=`":

```
bool operator!=( const StudentInCourse::SICKey& lhs,
                 const StudentInCourse::SICKey& rhs )
{  return ( (lhs.sIdNum != rhs.sIdNum) || (lhs.CCode != rhs.CCode) );  }

bool operator!=( const StudentInCourse& lhs, const StudentInCourse& rhs )
{  return ( lhs.getKey() != rhs.getKey() );  }
```

Και η κλάση εξαιρέσεων:

```
struct StudentInCourseXptn
{
   enum { negIdNum, keyLen, rangeError, unknownCCode,
          fileNotOpen, cannotWrite };
   StudentInCourse::SICKey objKey;
```

```
   char                    funcName[100];
   int                     errorCode;
   char                    errStrVal[100];
   float                   errFltVal;
   StudentInCourseXptn( const StudentInCourse::SICKey obk,
                        const char* mn, int ec, const char* sv="" )
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentInCourseXptn( const StudentInCourse::SICKey& obk,
                        const char* mn, int ec, float fv )
      :  objKey( obk ),  errorCode( ec ),  errFltVal( fv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentInCourseXptn
```

Prj04.8 Η Κλάση *StudentInCourseCollection*

Αν θυμηθούμε πώς γράψαμε τις *CourseCollection* και *StudentCollection* και πάρουμε υπόψη μας ότι «*το ζεύγος (sicSIdNum, sicCCode) είναι κλειδί*» γράφουμε μια πρώτη μορφή της

```
class StudentInCourseCollection
{
public:
   StudentInCourseCollection();
   ~StudentInCourseCollection()  {  delete[] siccArr;  };
// getters
   size_t getNOfStudentInCourses() const
   {  return siccNOfStudentInCourses;  }
   const StudentInCourse* getArr() const {  return siccArr;  }
// 1 StudentInCourse
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) >= 0 );  }
   const StudentInCourse& get1StudentInCourse( int aIdNum,
                                               string code ) const;
   void delete1StudentInCourse( int aIdNum, string code );
   void add1StudentInCourse( const StudentInCourse& aStdnInCrs );
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swapArr( StudentInCourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { siccIncr = 50 };
   StudentInCourse* siccArr;
   size_t           siccNOfStudentInCourses;
   size_t           siccReserved;

   StudentInCourseCollection( const StudentInCourseCollection& rhs ) { };
   StudentInCourseCollection& operator=(
                                  const StudentInCourseCollection& rhs ) { };

   void erase1StudentInCourse( int ndx );
   void insert1StudentInCourse( const StudentInCourse& aStdnInCrs );
   int findNdx( int aIdNum, string code ) const;
}; // StudentInCourseCollection
```

Ενδεικτικώς ας δούμε τη

```
int StudentInCourseCollection::findNdx( int aIdNum, string code ) const
{
   int ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( siccArr, siccNOfStudentInCourses,
                       0, siccNOfStudentInCourses-1,
                       StudentInCourse(aIdNum,code) );
```

```
   return ndx;
} // StudentInCourseCollection::findNdx
```

Εδώ πρόσεξε τα εξής:

- Αφού το κλειδί για τα αντικείμενα τύπου *StudentInCourse* είναι (*sicSIdNum, sicCCode*) η μέθοδος πρέπει να τροφοδοτηθεί με τα αντίστοιχα *aIdNum* και *code*. Τις ίδιες παραμέτρους έχουν και οι *find1StudentInCourse()*, *delete1StudentInCourse()*, *get1StudentInCourse()* και *delete1StudentInCourse()*.
- Πριν καλέσουμε τη *linSearch()* ελέγχουμε και τα δύο τμήματα του κλειδιού· θα πρέπει να έχουμε: *aIdNum* `> 0 &&` *code*`.`*length*`() ==` *Course*`::`*cCodeSz*`-1`.

Φυσικά, το ενδιαφέρον βρίσκεται στις *delete1StudentInCourse()* και *add1StudentInCourse()*. Η δεύτερη θα παίξει (και) τον ρόλο της *setIdNumCCode()* του Project 3.

Για να μπορεί η *add1StudentInCourse()* να παίξει τον ρόλο της *setIdNumCCode()* θα πρέπει να έχει πρόσβαση στον πίνακα μαθημάτων και στο μητρώο φοιτητών· θα βάλουμε και εδώ παραμέτρους για τους πίνακες; Όχι! Όπως κάναμε στη *StudentCollection* θα βάλουμε:

```
private:
// . . .
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;
```

με αρχικές τιμές στον δημιουργό:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   try
   {
      siccReserved = siccIncr;
      siccArr = new StudentInCourse[ siccReserved ];
      siccNOfStudentInCourses = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentInCourseCollectionXptn( "StudentInCourseCollection",
                                 StudentInCourseCollectionXptn::allocFailed );
   }
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection
```

και –για τον χειρισμό τους– τις

```
public:
// . . .
   const StudentCollection* getPAllStudents() const
   {  return siccPAllStudents;  }
   const CourseCollection* getPAllCourses( ) const
   {  return siccPAllCourses;  }
// setters
   void setPAllStudents( const StudentCollection* pStudents )
   {  siccPAllStudents = pStudents;  }
   void setPAllCourses( const CourseCollection* pCourses )
   {  siccPAllCourses = pCourses;  }
```

Μετά απο'αυτά, αν στο πρόγραμμά μας έχουμε δηλώσει:

```
   CourseCollection allCourses;
   StudentCollection allStudents;
   StudentInCourseCollection allEnrollments;
```

μπορούμε να δώσουμε:

```
   allEnrollments.setPAllStudents( &allStudents );
   allEnrollments.setPAllCourses( &allCourses );
```

Η *delete1StudentInCourse()* και *add1StudentInCourse()* θα κάνουν ελέγχους –και ενημερώσεις– αν βρίσκουν `siccPAllStudents != 0` ή/και `siccPAllCourses != 0` για να ανταποκριθούν στην απαίτηση: «*οι ενημερώσεις (εισαγωγές – διαγραφές) στους δύο πίνακες*

[sCourses της *Student* και siccArr της *StudentInCourseCollection*] *πρέπει να γίνονται με συνέπεια.*»

Ας δούμε λοιπόν ένα σχέδιο για την *add1StudentInCourse()*:

- Πριν από οποιαδήποτε ενέργεια θα πρέπει να ελέγξουμε αν υπάρχει ήδη καταχωρισμένη η εγγραφή του φοιτητή στο συγκεκριμένο μάθημα. Αν υπάρχει δεν υπάρχει δουλειά για τη μέθοδο:

```
void add1StudentInCourse( const StudentInCourse& aStdnInCrs )
{
   if ( στον siccArr δεν υπάρχει στοιχείο με
                              (aStdnInCrs.getSIdNum(),aStdnInCrs.getCCode()) )
   {
   // . . .
   }
} // StudentInCourseCollection::add1StudentInCourse
```

- Για να προχωρήσουμε στην ενημέρωση του πίνακα (εισαγωγή στοιχείου) θα πρέπει να έχουμε σιγουρέψει ότι πρόκειται για εγγραφή υπαρκτού φοιτητή σε υπαρκτό μάθημα:

```
   if ( siccPAllCourses == 0 )
      throw …
   if ( δεν υπάρχει μάθημα με κωδικό aStdnInCrs.getCCode() )
      throw …
   if ( siccPAllStudents == 0 )
      throw …
   if ( δεν υπάρχει φοιτητής με α.μ. aStdnInCrs.getSIdNum() )
      throw …
```

- Αν βρούμε όλα όσα χρειαζόμαστε κάνουμε τις ενημερώσεις:

```
   Ενημέρωσε το μάθημα (αριθμός φοιτητών)
   Πάρε το αντικείμενο του μαθήματος
   Ενημέρωσε τον φοιτητή (εισαγωγή μαθήματος)
   Κάνε εισαγωγή του aStdnInCrs στον siccArr
```

Για να ενημερώσουμε το μάθημα πρέπει να καλέσουμε την *add1Student()* της συλλογής (*CourseCollection*) που δείχνει το *siccPAllCourses*:

```
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
```

Από την ίδια συλλογή παίρνουμε το αντικείμενο του μαθήματος για να το χρησιμοποιήσουμε στην ενημέρωση το αντικείμενο του φοιτητή:

```
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
```

Για να ενημερώσουμε το αντικείμενο του φοιτητή πρέπει να καλέσουμε την *add1Course* της συλλογής (*StudentCollection*) που δείχνει το *siccPAllStudents*:[7]

```
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
```

Μετά από αυτά εισάγουμε το *aStdnInCrs* στον πίνακα siccArr της συλλογής *StudentInCourseCollection*:

```
      insert1StudentInCourse( aStdnInCrs );
```

Να ολόκληρη η μέθοδος:

```
void StudentInCourseCollection::add1StudentInCourse(
                                          const StudentInCourse& aStdnInCrs )
{
   if ( findNdx(aStdnInCrs.getSIdNum(), aStdnInCrs.getCCode()) < 0 )
   { // δεν υπάρχει στοιχείο
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                    StudentInCourseCollectionXptn::noCrs );
```

[7] Δεν είναι απαραίτητο να χρησιμοποιήσουμε την τοπική μεταβλητή *oneCourse*· θα μπορούσαμε να γράψουμε:

```
      pAllStudents->add1Course( aStdnInCrs.getSIdNum(),
                          pAllCourses->get1Course( aStdnInCrs.getCCode() );
```

```
      if ( !(siccPAllCourses->find1Course(aStdnInCrs.getCCode())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                  StudentInCourseCollectionXptn::unknownCrs,
                                              aStdnInCrs.getCCode() );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                   StudentInCourseCollectionXptn::noStdnt );
      if ( !(siccPAllStudents->find1Student(aStdnInCrs.getSIdNum())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                 StudentInCourseCollectionXptn::unknownStdn,
                                              aStdnInCrs.getSIdNum() );
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
      insert1StudentInCourse( aStdnInCrs );
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Η *delete1StudentInCourse()* είναι κάπως πιο απλή:

```
void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                        string code )
{
   int ndx( findNdx( aIdNum, code );
   if ( ndx >= 0 ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                     StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                   StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course(code) );
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      erase1StudentInCourse( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse
```

Η *StudentInCourseCollection* γίνεται τελικώς:

```
class CourseCollection;
class StudentCollection;

class StudentInCourseCollection
{ // version 1
public:
   StudentInCourseCollection();
   ~StudentInCourseCollection()  {  delete[] siccArr;  };
// getters
   size_t getNOfStudentInCourses() const
   {  return siccNOfStudentInCourses;  }
   const StudentInCourse* getArr() const {  return siccArr;  }
   const StudentCollection* getPAllStudents() const
   {  return siccPAllStudents;  }
   const CourseCollection* getPAllCourses( ) const
   {  return siccPAllCourses;  }
// setters
   void setPAllStudents( StudentCollection* pStudents )
   {  siccPAllStudents = pStudents;  }
   void setPAllCourses( CourseCollection* pCourses )
   {  siccPAllCourses = pCourses;  }
// 1 StudentInCourse
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) >= 0 );  }
   void delete1StudentInCourse( int aIdNum, string code );
   void add1StudentInCourse( const StudentInCourse& aStdnInCrs );
   const StudentInCourse& get1StudentInCourse( int aIdNum,
                                               string code ) const;
```

```
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swapArr( StudentInCourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { siccIncr = 50 };
   StudentInCourse*   siccArr;
   size_t             siccNOfStudentInCourses;
   size_t             siccReserved;
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;

   StudentInCourseCollection( const StudentInCourseCollection& rhs ) { };
   StudentInCourseCollection& operator=(
                                  const StudentInCourseCollection& rhs ) { };
   void erase1StudentInCourse( int ndx );
   void insert1StudentInCourse( const StudentInCourse& aStdnInCrs );
   int findNdx( int aIdNum, string code ) const;
}; // StudentInCourseCollection
```

Πρόσεξε τις δύο *προειδοποιήσεις δήλωσης* για τις κλάσεις *CourseCollection* και *StudentCollection*. Είναι απαραίτητες για να γίνουν δεκτές από τον μεταγλωττιστή οι δηλώσεις

```
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;
```

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentInCourseCollectionXptn
{ // version 1
   enum { allocFailed, notFound, noCrs, unknownCrs, noStdnt,
          unknownStdn, fileNotOpen, cannotWrite, cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   char errIntVal;
   StudentInCourse::SICKey errSICVal;
   StudentInCourseCollectionXptn( const char* mn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
   StudentInCourseCollectionXptn( const char* mn, int ec, int iv )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
   StudentInCourseCollectionXptn( const char* mn, int ec,
                                  const StudentInCourse::SICKey& sk )
      :  errorCode( ec ),  errSICVal( sk )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
}; // StudentInCourseCollectionXptn
```

Prj04.9 Πώς θα Γίνονται οι Ενημερώσεις

Ας πούμε ότι θέλουμε να κάνουμε εισαγωγή της εγγραφής ενός φοιτητή σε κάποιο μάθημα· θα πρέπει

- Να ενημερώσουμε ένα μέλος του πίνακα μαθημάτων (τύπου *Course Collection*), με την *add1Student()*.
- Να ενημερώσουμε ένα μέλος του μητρώου φοιτητών (τύπου *StudentCollection*), με την *add1Course()*.
- Να εισαγάγουμε ένα νέο στοιχείο στον πίνακα εγγραφών στα μαθήματα (τύπου *StudentInCourseCollection*) με την *add1StudentInCourse()*.

Ποιος διασφαλίζει ότι όλα αυτά θα γίνουν σωστά ώστε να είναι σωστό το περιεχόμενο των πινάκων μας; Το πρόγραμμα που τους χρησιμοποιεί; Ούτε για αστείο! Αυτό πρέπει να διασφαλίζεται από αυτόν που γράφει τις κλάσεις. *Εδώ* θα γίνει.

Η *StudentInCourseCollection*::*add1StudentInCourse()*, όπως τη γράψαμε, κάνει όλες τις ενημερώσεις που παραθέτουμε παραπάνω. Αυτό όμως δεν διασφαλίζει οτιδήποτε, αφού ο προγραμματιστής μπορεί να βάλει κλήσεις (και) προς τις άλλες μεθόδους. Πώς διορθώνεται αυτό; Έχουμε δύο επιλογές:

- Να γράψουμε τις (*Course Collection*::)*add1Student()*, (*StudentCollection*::)*add1Course()* έτσι ώστε να κάνουν όλες τις ενημερώσεις. Με αυτήν την επιλογή θα έχεις το δικαίωμα να κάνεις εισαγωγή εγγραφής σε ένα μάθημα με οποιαδήποτε από τις τρεις μεθόδους. Αλλά και κάθε φορά που θα αλλάζεις το λογισμικό σου θα πρέπει να αλλάζεις και τις τρεις μεθόδους με συνεπή τρόπο.
- Να κρύψουμε (σε περιοχή **private**) τις (*CourseCollection*::)*add1Student()*, (*StudentCollection*::)*add1Course()* ώστε η *add1StudentInCourse()* να είναι η μόνη υπεύθυνη για όλες τις ενημερώσεις.

Η δεύτερη επιλογή έχει αυτό το πολύ σημαντικό πλεονέκτημα: μια μεθοδος «*είναι η μόνη υπεύθυνη για όλες τις ενημερώσεις*.» Η πρώτη επιλογή, χωρίς αυτό το «προσόν», σου εγγυάται(!) ότι –συν τω χρόνω, με τις αλλαγές και προσαρμογές που θα πρέπει να γίνονται– θα έχεις λογισμικό και δεδομένα με πολλά προβλήματα.

Φυσικά, θα πρέπει να έχουμε και *μια μονον μέθοδο υπεύθυνη για τη όλες τις ενημερώσεις που αφορούν τη διαγραφή* ενός φοιτητή από ένα μάθημα.

Πώς υλοποιείται η επιλογή μας;

- Δηλώνουμε στην *CourseCollection* και στην *StudentCollection*:

```
friend void StudentInCourseCollection::add1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                               string code );
```

- «Κρύβουμε» στην περιοχή **private** της *CourseCollection* τις *add1Student* και *delete1Student*.
- «Κρύβουμε» στην περιοχή **private** της *StudentCollection* τις *add1Course* και *delete1Course*.

Prj04.10 Οι Άλλες Συλλογές Τελικώς

Τώρα μπορούμε να δώσουμε τις τελικές –αν και όχι πλήρεις– μορφές των κλάσεων *CourseCollection* και *StudentCollection*.

Prj04.10.1 Η Κλάση *CourseCollection*

Θα ξαναδούμε τώρα την *CourseCollection*::*delete1Course()*. Η λύση που δώσαμε στην §Prj04.3.1 φαίνεται κάπως «τεμπέλικη» διότι αυτό που λέμε στην πραγματικότητα είναι το εξής: Θέλεις να διαγράψεις ένα μάθημα στο οποίο είναι γραμμένοι φοιτητές (*cNoOfStudents* > 0); Τότε:

- Διάγραψε πρώτα μια προς μια τις εγγραφές των φοιτητών στο συγκεκριμένο μάθημα ενημερώνοντας παραλλήλως τα αντικείμενα των φοιτητών.
- Μετά από αυτό (αφού θα έχεις *cNoOfStudents* == 0) διάγραψε και το αντικείμενο του μαθήματος.

Παίρνοντας υπόψη μας και αυτά που λέγαμε στην Prj04.4 σου προτείνουμε (ως άσκηση) το εξής:

- Εφοδίασε το σύστημά σου με έναν πίνακα τύπου *DeletedCourseCollection* και έναν *DeletedStudentInCourseCollection*[8] με στοιχεία αντίστοιχων κλάσεων

```
class DeletedCourse
{
private:
// . . .
public:
   Course dcCourse;
   time_t dcDelTime; // ή DateTime dcDelTime (χρόνος διαγραφής)
}; // DeletedCourse
class DeletedStudentInCourse
{
private:
// . . .
public:
   StudentInCourse dsiccEnroll;
   time_t          dsiccDelTime; // ή DateTime dsiccDelTime
}; // DeletedStudentInCourse
```

- Μην πειράξεις την *CourseCollection*::*delete1Course()* αλλά γράψε μιαν άλλη μέθοδο, ας την πούμε *CourseCollection*::*moveOut1Course()*, που θα κάνει τα εξής:
  - Θα διαγράφει τις εγγραφές στο μάθημα από τον πίνακα δηλώσεων και αφού εξοπλίσει την κάθε μια με τον χρόνο διαγραφής (μετατροπή σε αντικείμενο *DeletedStudentInCourse*) θα τις εισάγει στον πίνακα τύπου *DeletedStudentInCourseCollection*. Προφανώς θα χρειαστείς και μια *StudentInCourseCollection*::*moveOut1StudentInCourse()*.
  - Θα διαγράφει το αντικείμενο του μαθήματος από τον πίνακα μαθημάτων και αφού το μετατρέψει σε αντικείμενο *DeletedCourse* θα το εισάγει στον πίνακα *DeletedCourseCollection*.
  - Φυσικά, θα πρέπει να ενημερώνει και τα αντικείμενα των φοιτητών που έχουν εγγραφεί στο μάθημα που διαγράφεται.

Όλες οι διαγραφές από τον πίνακα δηλώσεων θα πρέπει να γίνονται με τη *StudentInCourseCollection*::*delete1StudentInCourse()* και μόνον. Για να έχεις πρόσβαση από τον πίνακα μαθημάτων στον πίνακα των εγγραφών στα μαθήματα θα χρειαστείς ένα βέλος:

```
   StudentInCourseCollection* ccPAllEnrollments;
```

που θα παίρνει τιμή "`0`" όταν δημιουργείται η συλλογή (πίνακας μαθημάτων) και θα το διαχειρίζεσαι με τις:

```
const StudentInCourseCollection*
                        CourseCollection::getPAllEnrollments() const
{  return ccPAllEnrollments;  }

void CourseCollection::setPAllEnrollments(
                              StudentInCourseCollection* pEnrollments )
{  ccPAllEnrollments = pEnrollments;  }
```

Αφού σχεδιάσεις και υλοποιήσεις τους πίνακες διαγραφέντων μάλλον θα πρέπει να βάλεις αντίστοιχα εργαλεία και για αυτούς.

Ο ορισμός της κλάσης *CourseCollection* είναι:

```
class CourseCollection
{  // version 1
friend void StudentInCourseCollection::add1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                               string code );
public:
   CourseCollection();
```

[8] Φυσικά θα χρειαστεί και ένας πίνακας *DeletedStudentCollection*.

```
   ~CourseCollection()  {  delete[] ccArr;  };
// getters
   size_t getNOfCourses() const {  return ccNOfCourses;  }
   const Course* getArr() const {  return ccArr;  }
//   const StudentInCourseCollection* getPAllEnrollments() const
//   {  return ccPAllEnrollments;  }
// setters
//   void setPAllEnrollments( StudentInCourseCollection*
//                                                    pEnrollments)
//   {  ccPAllEnrollments = pEnrollments;  }
// 1 Course
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
   void delete1Course( string code );
   void add1Course( const Course& aCourse );
   const Course& get1Course( string code ) const;
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swap( CourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { ccIncr = 30 };
   Course*                   ccArr;
   size_t                    ccNOfCourses;
   size_t                    ccReserved;
//   StudentInCourseCollection* ccPAllEnrollments;

   CourseCollection( const CourseCollection& rhs ) { };
   CourseCollection& operator=( const CourseCollection& rhs ) { };
   void erase1Course( int ndx );
   void insert1Course( const Course& aCourse );
   void add1Student( string code );
   void delete1Student( string code );
   int findNdx( const string& code ) const;
}; // CourseCollection
```

Σε σχόλια έχουμε βάλει τα εργαλεία για την πρόσβαση στον πίνακα εγγραφών στα μαθήματα.[9]

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct CourseCollectionXptn
{  // version 1
   enum { allocFailed, notFound, entity, cannotDel, prereqRef,
          noEnroll, enrollRef, fileNotOpen, cannotWrite,
          cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   int  errIntVal;
   CourseCollectionXptn( const char* mn, int ec,
                         const char* sv="", int iv=0 )
```

[9] Και πώς θα δεχθεί ο μεταγλωττιστής τη δήλωση της *ccPAllEnrollments*; Δεν θα χρειαστεί μια προειδοποίηση δήλωσης της *StudentInCourseCollection* πριν από τη δήλωση της `CourseCollection`; Εδώ έχουμε και άλλες απαιτήσεις από τις δηλώσεις "`friend`" των `StudentInCourseCollection::add1StudentInCourse()` και `StudentInCourseCollection::delete1StudentInCourse()`. Εδώ το πρόβλημα λύνεται με τη δήλωση της *StudentInCourseCollection* πριν από τη δήλωση της *CourseCollection*. Αφού παρόμοια ισχύουν και για την *StudentCollection*, στο πρόγραμμά μας θα πρέπει να βάλουμε:

```
#include "StudentInCourseCollection.h"
#include "CourseCollection.h"
#include "StudentCollection.h"
```

Έτσι, όταν έρχεται η δήλωση της *CourseCollection* (και μετά, της *Student Collection*) ο μεταγλωττιστής έχει ήδη βρει τη *StudentInCourseCollection* και τις μεθόδους της *add1StudentInCourse()* και *delete1StudentInCourse()*.

```
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
}; // CourseCollectionXptn
```

Παρατήρηση: ▶

Κρύψαμε τον δημιουργό αντιγραφής και τον τελεστή εκχώρησης για να εμποδίσουμε την αντιγραφή αντικειμένων κλάσης *CourseCollection*. Τελικώς, απλώς τη δυσκολέψαμε: με τις *getNOfCourses()*, *getArr()* και *add1Course()* μπορείς να δημιουργήσεις αντίγραφα. Πάντως, έχουμε την εξής ελπίδα: ο προγραμματιστής που θα προσπαθήσει να δημιουργήσει αντίγραφα αντικειμένων κλάσης *CourseCollection*, θα δει τα αχρηστευμένα και θα το ξανασκεφτεί...◀

Prj04.10.2 Η Κλάση *StudentCollection*

Το πρώτο πράγμα που θα πρέπει να κάνουμε είναι η διόρθωση της *add1Student()*. Αφού είπαμε ότι η εισαγωγή εγγραφών σε μαθήματα θα γίνεται από «μια πόρτα», την *add1StudentInCourse()*, αυτή θα πρέπει να καλούμε στη **for** της *add1Student()*. Ο έλεγχος

```
if ( !scPAllCourses->find1Course(aStCourses[k].s) )
```

είναι περιττός αφού γίνεται από την *add1StudentInCourse()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      insert1Student( aStudent );
      const Course::CourseKey* aStCourses( aStudent.getCourses() );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                                 StudentInCourse(aStudent.getIdNum(),
                                                 aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Πρόσεξε τα εξής:

- Θα χρειαστούμε ένα βέλος –ας το πούμε *scPAllEnrollments*– προς τον *Πίνακα Δηλώσεων Μαθημάτων*. Η τιμή του ελέγχεται στη δεύτερη **if**. *Οι δηλώσεις και οι μέθοδοι χειρισμού του βέλους προς τον πίνακα μαθημάτων θα αντικατασταθούν από τα αντίστοιχα του scPAllEnrollments*.
- Με την *insert1Student()* εισάγεται και ο πίνακας με τους κωδικούς μαθημάτων. Στη συνέχεια, η *StudentInCourseCollection*::*add1StudentInCourse()* θα καλεί τη *StudentCollection*::*add1Course()* που καλεί τη *Student*::*add1Course()* για να εισαγάγει τον κάθε κωδικό στον πίνακα κωδικών μαθημάτων. Αλλά η τελευταία δεν κάνει εισαγωγή αν τον βρει ήδη στον πίνακα.
- Υπάρχει μια σοβαρή διαφορά της *StudentCollection*::*add1Student()* από την *CourseCollection*::*add1Course()*:
  - Στο μητρώο φοιτητών εισάγουμε οποιοδήποτε αντικείμενο κλάσης *Student* χωρίς οποιοδήποτε πρόβλημα. Και αυτό διότι το αντικείμενο *Student*, με τον πίνακα μαθημάτων που περιέχει, μας επιτρέπει να ενημερώσουμε σωστά τον πίνακα δηλώσεων μαθημάτων και τον πίνακα μαθημάτων.
  - Στον πίνακα μαθημάτων εισάγουμε αντικείμενα κλάσης *Course* με *μηδενισμένο το ocNoOfStudents*. Γιατί; Διότι ένα αντικείμενο *Course* έχει *μόνον το πλήθος* των

φοιτητών που το ζήτησαν και *όχι τους αριθμούς μητρώου*. Έτσι είναι αδύνατη η ενημέρωση του πίνακα δηλώσεων μαθημάτων και των αντικειμένων με τα στοιχεία των φοιτητών.

Ας έλθουμε τώρα στη *StudentCollection*::*delete1Student()*, που την κάναμε «τεμπέλικη», αφού σβήνει μόνον φοιτητές που δεν είναι γραμμένοι σε μαθήματα. Παρ' όλο που η διαγραφή των εγγραφών ενός φοιτητή σε 4 ή 5 μαθήματα από τον *Πίνακα Δηλώσεων Μαθημάτων* δεν είναι τόσο «τραγική» όσο η διαγραφή των εγγραφών μερικών δεκάδων φοιτητών σε ένα μάθημα, θα σου προτείνουμε να χειριστείς τη διαγραφή φοιτητή με παρόμοιο τρόπο:

- Όρισε μια κλάση:

```
class DeletedStudent
{
private:
// . . .
public:
   Student dsStudent;
   time_t  dsDelTime; // ή DateTime dsDelTime (χρόνος διαγραφής)
}; // DeletedStudent
```

- Μην πειράξεις την («τεμπέλικη») *StudentCollection*::*delete1Student()* αλλά γράψε μιαν άλλη μέθοδο, ας την πούμε *StudentCollection*::*moveOut1Student()*, που θα κάνει τα εξής:
  - Θα διαγράφει τις εγγραφές του φοιτητή στα μαθήματα από τον πίνακα δηλώσεων και αφού εξοπλίσει την κάθε μια με τον χρόνο διαγραφής (μετατροπή σε αντικείμενο *DeletedStudentInCourse*) θα τις εισάγει στον πίνακα τύπου *DeletedStudentInCourseCollection* που είδαμε πιο πριν.
  - Θα διαγράφει το αντικείμενο του φοιτητή από το μητρώο φοιτητών και αφού το μετατρέψει σε αντικείμενο *DeletedStudent* θα το εισάγει στον πίνακα *DeletedStudentCollection*.
  - Θα πρέπει να ενημερώνει και τα αντικείμενα των μαθημάτων στα οποία είχε εγγραφεί ο φοιτητής που διαγράφεται.

Θα πρέπει ακόμη να γράψουμε τις «ενδιάμεσες» μεθόδους *add1Course()* και *delete1Course()*. Να θυμίσουμε ότι θα είναι *εσωτερικές* και θα καλούντσι μόνον από τις *φίλες* *StudentInCourseCollection*::*add1StudentInCourse()* και *StudentInCourseCollection*::*delete1StudentInCourse()*.

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   scArr[ndx].add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   scArr[ndx].delete1Course( aCourse );
} // StudentCollection::add1Course
```

Να πώς δηλώνεται η κλάση *StudentCollection*:

```
class StudentCollection
{  // version 1
friend void StudentInCourseCollection::add1StudentInCourse(
                                 const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse(
```

```
                                        int aIdNum, string code );
public:
   StudentCollection();
   ~StudentCollection()  {  delete[] scArr;  delete[] scIndex;  };
// getters
   size_t getNOfStudents() const {  return scNOfStudents;  }
   const Student* getArr() const {  return scArr;  }
   const StudentInCourseCollection* getPAllEnrollments() const
   {  return scPAllEnrollments;  }
// setters
   void setPAllEnrollments( StudentInCourseCollection*
                                                    pEnrollments )
   {  scPAllEnrollments = pEnrollments;  }
// 1 Student
   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) >= 0 );  };
   const Student& get1Student( int aIdNum ) const;
   void add1Student( const Student& aStudent );
   void delete1Student( int aIdNum );
// other
   void save( ofstream& bout, IndexEntry* index );
   void load( ifstream& bin );
   void swapArr( StudentCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { scIncr = 30 };
   Student*                   scArr;
   size_t                     scNOfStudents;
   size_t                     scReserved;
   StudentInCourseCollection* scPAllEnrollments;

   StudentCollection( const StudentCollection& rhs ) { };
   StudentCollection& operator=( const StudentCollection& rhs ) { };
   void erase1Student( int ndx );
   void insert1Student( const Student& aStudent );
   void add1Course( int aIdNum, const Course& aCourse );
   void delete1Course( int aIdNum, const Course& aCourse );
   int findNdx( int aIdNum ) const;
}; // StudentCollection
```

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentCollectionXptn
{  // version 1
   enum { allocFailed, notFound, noEnroll, enrollRef,
          fileNotOpen, cannotWrite, cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   char errIntVal;
   StudentCollectionXptn( const char* mn, int ec,
                          const char* sv="", int iv=0  )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
   StudentCollectionXptn( const char* mn, int ec, int iv )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
}; // StudentCollectionXptn
```

Prj04.11 Το 1ο Πρόγραμμα – Δημιουργία

Κρατούμε κατά βάση το σχέδιο που χρησιμοποιήσαμε στην πρώτη εκδοχή του προγράμματος (Prj03.2)· η υλοποίηση όμως θα είναι διαφορετική και μάλλον πιο απλή αφού αρκετή πολυπλοκότητα «κρύφτηκε» μέσα στις κλάσεις και στις μεθόδους τους.

Ας ξεκινήσουμε με τη

```
void loadCourses( string flNm, CourseCollection& allCourses )
{
// Άνοιξε το αρχείο
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "loadCourses", ProgXptn::cannotOpen,
                      flNm.c_str() );
// Τώρα διάβαζε
   Course oneCourse;
   loadSylCourse( bin, oneCourse );
   while ( !bin.eof() )
   {
      oneCourse.clearStudents();
      allCourses.add1Course( oneCourse );
      loadSylCourse( bin, oneCourse );
   } // while
   bin.close();
} // loadCourses
```

(όπου *loadSylCourse()* είναι –χωρίς αλλαγές– αυτή που είδαμε στο πρόγραμμα του Project 3.)

Εδώ βλέπεις τι εννοούμε: όλη η διαχείριση δυναμικής μνήμης κρύφτηκε στις μεθόδους της *allCourses*.

Prj04.11.1 Αρχείο Φοιτητών και Δηλώσεων Μαθημάτων

Ας δούμε τώρα πώς θα διαβάσουμε το αρχείο `enrllmnt.txt`:

- Μπορούμε να διαβάσουμε *όλα* τα στοιχεία (μαζί με τους κωδικούς μαθημάτων που δήλωσε) που αφορούν έναν φοιτητή και να τα εισαγάγουμε στο Μητρώο Φοιτητών αν δεν υπάρχει ήδη μέσα εγγραφή για τον συγκεκριμένο φοιτητή.
- Μπορούμε να διαβάσουμε τα στοιχεία του φοιτητή –*αλλά όχι τους κωδικούς των μαθημάτων*– και να τα εισαγάγουμε στο Μητρώο Φοιτητών. Στη συνέχεια διαβάζουμε και εισάγουμε –στο αρχείο δηλώσεων– έναν προς έναν τους κωδικούς μαθημάτων μαζί με τον αριθμό μητρώου του φοιτητή.

  Ας δούμε τι θα γίνεται με τα λάθη στην *πρώτη περίπτωση*:
- Αν στο Μητρώο Φοιτητών βρεθεί ο αριθμός μητρώου δεν γίνεται η εισαγωγή και προχωρούμε στον επόμενο φοιτητή.
- Αν στο αντικείμενο του φοιτητή βρεθεί κωδικός μαθήματος που δεν υπάρχει στον πίνακα μαθημάτων θα πάρουμε από την *add1StudentInCourse()* –μέσω της *add1Student()*– εξαίρεση. Στην περίπτωση αυτή θα πρέπει να διαγράψουμε τον κωδικό από το αντικείμενο του φοιτητή και να δοκιμάσουμε πάλι την εισαγωγή. Αν πάρουμε και πάλι *unknownCrs* διαγράφουμε τον κωδικό και ξαναδοκιμάζουμε. Αυτή η διαδικασία επαναλαμβάνεται μέχρις ότου γίνει η εισαγωγή. Για να καταλάβεις καλύτερα αυτό που γίνεται, ας πούμε ότι έχουμε δηλώσεις 6 (0..5) μαθημάτων και έχουμε λάθος στους κωδικούς 2, 3 και 5.
  - Την πρώτη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1 και 2. Διαγράφουμε τον κωδικό στη θέση 2 και εκεί έρχεται αυτός που ήταν στη θέση 5. Ο πίνακας μένει με 5 (0..4) στοιχεία.
  - Τη δεύτερη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1 και 2. Διαγράφουμε τον κωδικό στη θέση 2 και εκεί έρχεται αυτός που ήταν στη θέση 4. Ο πίνακας μένει με 4 (0..3) στοιχεία.
  - Την τρίτη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1, 2 και 3. Διαγράφουμε τον κωδικό στη θέση 3.
  - Την τέταρη φορά: γίνονται (επιτυχείς) αναζητήσεις για τους κωδικούς 0, 1 και 2.

Μπορούμε να αποφύγουμε τα παραπάνω αν ελέγχουμε τους κωδικούς των μαθημάτων μόλις τους διαβάσουμε και εισάγουμε στον πίνακα του αντικειμένου μόνον τους σωστούς. Έτσι το αντικείμενο εισάγεται στο μητρώο φοιτητών με την πρώτη. Στο παραπάνω παράδειγμα εισάγουμε μόνον τους κωδικούς 0, 1 και 3. Πάντως θα πρέπει να πάρεις υπόψη σου ότι αυτοί οι έλεγχοι θα (ξανα)γίνουν όταν καλέσουμε την *add1Student()*.

Ερχόμαστε τώρα στη διαχείριση των σφαλμάτων στη *δεύτερη περίπτωση*:

- Αν δεν βρεθεί ο αριθμός μητώου η εισαγωγή στο Μητρώο Φοιτητών γίνεται αφού δεν υπάρχουν άλλα προβλήματα. Αν βρεθεί ο αριθμός μητρώου δεν γίνεται η εισαγωγή αλλά έχουμε μια διαφορά από την πρώτη περίπτωση: πριν προχωρήσουμε στον επόμενο φοιτητή θα πρέπει να διαβάσουμε και να αγνοήσουμε τους κωδικούς των μαθημάτων.
- Αν εισαχθεί ο φοιτητής είσαγουμε και τις δηλώσεις μαθημάτων, καλώντας την *add1StudentInCourse()*, για το καθένα. Αν για κάποιο ακυρωθεί η εισαγωγή πιάνουμε την εξαίρεση και γράφουμε στο *log*. Εδώ οι έλεγχοι που θα γίνουν είναι μόνον αυτοί της *add1StudentInCourse()* και των συναρτήσεων που αυτή καλεί.

Όπως καταλαβαίνεις ο δεύτερος τρόπος είναι προτιμότερος· αυτόν θα υλοποιήσουμε.

Πριν από αυτό όμως θα πρέπει να δούμε τις εξαιρέσεις που μας ενδιαφέρουν, δηλαδή αυτές για τις οποίες θα γράψουμε καταχώριση στο *log*:

- Στην *add1StudentInCourse()* το πρώτο πράγμα που κάνουμε είναι να αναζητήσουμε την εγγραφή σε ένα αντικείμενο κλάσης *StudentInCourseCollection*. Για την αναζήτηση (*findNdx()*) θα πρέπει να δημιουργηθεί –από τον αριθμό μητρώου του φοιτητή και τον κωδικό του μαθήματος– ένα αντικείμενο κλάσης *StudentInCourse*. Αν η δομή του κωδικού μαθήματος δεν συμμορφώνεται με τους κανόνες (το μήκος να είναι 7)[10] *ο δημιουργός της StudentInCourse* θα ρίξει εξαίρεση *StudentInCourseXptn* με κωδικό *keyLen*.
- Αν δεν υπάρχει πρόβλημα με τη δομή του κωδικού μαθήματος γίνεται αναζήτηση στον πίνακα μαθημάτων· αν δεν βρεθεί μάθημα με τέτοιον κωδικό ρίχνεται εξαίρεση *StudentInCourseCollectionXptn* με κωδικό *unknownCrs* από την *add1StudentInCourse*.

Θα διαβάσουμε το αρχείο με τα στοιχεία των φοιτητών με την:

```
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log )
{
   ifstream tin( flNm.c_str() ); //"enrllmnt.txt" );
   if ( tin.fail() )
      throw ProgXptn( "readStudentData", ProgXptn::cannotOpen, flNm.c_str() );
   do {
      readAStudent( tin, allStudents, allEnrollments, allCourses, log );
   } while( !tin.eof() );
   tin.close();
} // readStudentData
```

που καλεί την παρακάτω συνάρτηση για να διαβάσει τα στοιχεία του κάθε φοιτητή και να ενημερώσει καταλλήλως τους πίνακες:

```
void readAStudent( istream& tin,
                   StudentCollection& allStudents,
                   StudentInCourseCollection& allEnrollments,
                   CourseCollection& allCourses,
                   ofstream& log )
{
   Student      aStudent;
   unsigned int sIdNum;

// Διάβασε τα στοιχεία ενός φοιτητή
```

[10] ή κάτι πιο πολύπλοκο, σαν αυτά που έχουμε στην άσκ. Prj03-1.

```
   aStudent.readPartFromText( tin );
   if ( !tin.eof() )
   {
      sIdNum = aStudent.getIdNum();
      if ( allStudents.find1Student(aStudent.getIdNum()) )
      { // υπάρχει στον πίνακα φοιτητών
      // Αγνόησε τη δήλωση
         ignoreStudentData( tin );
         log << " multiple entry for student with id num "
             << sIdNum << endl;
      }
      else
      {
      // Βάλε τον φοιτητή στον πίνακα φοιτητών
         allStudents.add1Student( aStudent );
         string str1;
         getline( tin, str1, '\n' );
         if ( !tin.eof() )
         {
            int noc( atoi(str1.c_str()) );
            for ( int k(0); k < noc && !tin.eof(); ++k )
            {
               getline( tin, str1, '\n' );
               try
               {
                  StudentInCourse aStdnInCrs( sIdNum, str1 );
                  allEnrollments.add1StudentInCourse( aStdnInCrs);
               }
               catch( StudentInCourseXptn& x )
               {
                  if ( x.errorCode == StudentInCourseXptn::keyLen )
                     log << " student with id num "
                         << aStudent.getIdNum()
                         << " asking course " << str1 << endl;
                  else
                     throw;
               } // catch( StudentInCourseXptn...
               catch( StudentInCourseCollectionXptn& x )
               {
                  if ( x.errorCode ==
                                  StudentInCourseCollectionXptn::unknownCrs )
                     log << " student with id num " << aStudent.getIdNum()
                         << " asking course " << str1 << endl;
                  else
                     throw;
               } // catch( StudentInCourseCollectionXptn...
            } // for
         } // if ( !tin.eof() )
         if ( !tin.eof() ) getline( tin, str1, '\n' );//blank line
      } // if ( allStudents.find1Student(aStudent.getIdNum()) )
   } // if ( !tin.eof() )
} // readAStudent
```

Η κάθε μια `catch` «μεταφράζει» σε C++ αυτά που είπαμε παραπάνω για τις εξαιρέσεις. Πρόσεξε ότι:

- Οι εξαιρέσεις άλλων τύπων –εκτός από τους δύο που μας ενδιαφέρουν– περνούν «ανενόχλητες».
- Ακόμη περνούν –αλλά με τη "`throw;`"– οι εξαιρέσεις που πιάνουμε αλλά έχουν κωδικό άλλον από αυτόν που μας ενδιαφέρει.

Prj04.11.2 Έλεγχος Δηλώσεων

Η πιο «φυσική» θέση της συνάρτησης για τον έλεγχο των δηλώσεων μαθημάτων (εβδομαδιαίου φόρτου) είναι ως μια μέθοδος της *StudentCollection*:

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      if ( scArr[k].getWH() > maxWH )
         log << "student with id num " << scArr[k].getIdNum() << ": "
             << scArr[k].getWH() << " hours/week" << endl;
   } // for
} // StudentCollection::checkWH
```

Αυτή καλείται από τη **main**, μετά την ανάγνωση των δεδομένων όλων των φοιτητών:

```
      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );
```

Prj04.11.3 Φύλαξη

Η φύλαξη των συλλογών (και του ευρετηρίου) γίνεται με κλήση της

```
void saveCollections( CourseCollection& allCourses,
                      StudentCollection& allStudents,
                      SIndexEntry* index,
                      StudentInCourseCollection& allEnrollments )
{
   ofstream bout( "Courses.dta", ios_base::binary );
   allCourses.save( bout );
   bout.close();
   bout.open( "students.dta", ios_base::binary );
   allStudents.save( bout, index );
   bout.close();
   bout.open( "students.ndx", ios_base::binary );
   for ( int k(0); k < allStudents.getNOfStudents(); ++k )
      bout.write( reinterpret_cast<const char*>(&index[k]),
                  sizeof(SIndexEntry) );
   bout.close();
   bout.open( "enrllmnt.dta", ios_base::binary );
   allEnrollments.save( bout );
   bout.close();
} // saveCollections
```

Αφού το ευρετήριο είναι πίνακας μπορεί να φυλαχθεί με πιο απλό τρόπο:

```
   bout.write( reinterpret_cast<const char*>(index),
               allStudents.getNOfStudents()*sizeof(IndexEntry) );
```

Prj04.11.4 ...Και το Πρόγραμμα

Το πρόγραμμα θα είναι ως εξής:

```
#include <string>
#include <fstream>
#include <new>
#include <iostream>

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "StudentInCourse.cpp"
#include "StudentInCourseCollection.h"
#include "CourseCollection.h"
#include "SIndexEntry.h"
```

```
#include "StudentCollection.h"
#include "CourseCollection.cpp"
#include "StudentCollection.cpp"
#include "StudentInCourseCollection.cpp"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadCourses( string flNm, CourseCollection& allCourses );
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log );
void saveCollections( CourseCollection& allCourses,
                      StudentCollection& allStudents,
                      IndexEntry* index,
                      StudentInCourseCollection& allEnrollments );

int main()
{
   try
   {
      CourseCollection allCourses;
      StudentCollection allStudents;
      StudentInCourseCollection allEnrollments;

      allEnrollments.setPAllStudents( &allStudents );
      allEnrollments.setPAllCourses( &allCourses );

      allCourses.setPAllEnrollments( &allEnrollments );
      allStudents.setPAllEnrollments( &allEnrollments );

      loadCourses( "Courses.dta", allCourses );

      ofstream log( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt");

      readStudentData( "enrllmnt.txt", allStudents, allEnrollments,
                       allCourses, log );

      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );

      log.close();

      SIndexEntry* index;
      try
      {  index = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }

      saveCollections( allCourses, allStudents, index, allEnrollments );

      delete[] index;
   } // try
```

```
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( CourseXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )
   {
      switch ( x.errorCode )
      {
         case StudentXptn::allocFailed:
            cout << "Student " << x.objKey << ": no dynamic memory in "
                 << x.funcName << endl;
            break;
         case StudentXptn::negIdNum:
            cout << "Student " << x.objKey << ": illegal id num ("
                 << x.errIntVal << ") in " << x.funcName << endl;
            break;
         case StudentXptn::incomplete:
            cout << "incomplete data for Student " << x.objKey << " in "
                 << x.funcName << endl;
            break;
         case StudentXptn::fileNotOpen:
            cout << "file " << x.errStrVal << " not open in " << x.funcName
                 << endl;
            break;
         case StudentXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case StudentXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected StudentXptn for Student " << x.objKey
                 << " from " << x.funcName << endl;
      } // switch
   } // catch( StudentXptn
   catch( StudentInCourseXptn& x )  { /* . . . */ }
   catch( CourseCollectionXptn& x )  { /* . . . */ }
   catch( StudentCollectionXptn& x )  { /* . . . */ }
   catch( StudentInCourseCollectionXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Να παρατηρήσουμε τα εξής:

- Αν αλλάξεις τη σειρά των οδηγιών "`include`" μπορεί να αντιμετωπίσεις προβλήματα.
- Η "`delete[] index`" είναι απαραίτητη; Σε κάθε περίπτωση, αφού εκεί τελειώνει το πρόγραμμα, η δυναμική μνήμη θα ελευθερωθεί! Σωστό! Τη βάλαμε μόνο και μόνο για να δεις ότι ενώ για την ανακύκλωση των δυναμικών πινάκων που είναι μέσα σε αντικείμενα θα δράσουν οι καταστροφείς για τον index θα πρέπει να ζητήσουμε ρητώς την ανακύκλωση.
- Όπως καταλαβαίνεις, δεν υπάρχει λόγος να παραθέσουμε το μακροσκελέστατο τμήμα διαχείρισης εξαιρέσεων –δηλαδή όλες τις `catch`– αφού οι μόνες που έχουν πραγματικό ενδιαφέρον είναι αυτές της *readAStudent* και όχι αυτές της **main**.
- Τέλος, πρόσεξε πόσο απλή είναι η **main** αφού όλη η πολυπλοκότητα έχει κρυφτεί μέσα στις μεθόδους.

Prj04.12 Το 2ο Πρόγραμμα – Εκμετάλλευση

Το δεύτερο πρόγραμμα είναι πολύ απλό:

```
#include <fstream>
```

```
#include <iostream>
#include <new>

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "SIndexEntry.h"

struct ProgXptn
// ΟΠΩΣ ΣΤΟ 1ο ΠΡΟΓΡΑΜΜΑ

void loadIndex( string flNm, PIndexEntry& index, unsigned int& ndxSz );
void retrieve( string flNm, PIndexEntry index, int ndxSz );

int main()
{
   IndexEntry*  index;
   unsigned int ndxSz;
   try
   {
      loadIndex( "students.ndx", index, ndxSz );
      retrieve( "students.dta", index, ndxSz );
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
}
```

Στο `SIndexEntry.h` έχουμε τα εξής:

```
struct SIndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
   explicit IndexEntry( int aIdNum=0, int aLoc=0 )
   {  sIdNum = aIdNum;  loc = aLoc;  }
}; // SIndexEntry

typedef SIndexEntry* PIndexEntry;

bool operator!=( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return ( lhs.sIdNum != rhs.sIdNum );  }

bool operator==( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return !(lhs != rhs);  }
```

Επιφορτώνουμε τον "`!=`" (και τον "`==`") και γράφουμε τον ερήμην δημιουργό με αρχικές τιμές διότι θα μας χρειαστούν για να χρησιμοποιήσουμε τη *linSearch()*. Δύο αντικείμενα *SIndexEntry* θεωρούνται ίσα αν και μόνον αν έχουν ίδια τιμή στο *sIdNum*. Φυσικά, θα παρατηρήσεις ότι ο δημιουργός χρειάζεται κάποιους ελέγχους. Σωστό! Γράψε τους (μαζί και μια κλάση εξαιρέσεων)!

Ας δούμε τώρα τις δύο συναρτήσεις. Η *loadIndex()* βασίζεται σε αυτά που μάθαμε στην §15.12.1:

```
void loadIndex( string flNm, PIndexEntry& index, unsigned int& ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "loadIndex", ProgXptn::cannotOpen, flNm.c_str() );

   bin.seekg( 0, ios_base::end );
   unsigned int flSize( bin.tellg() );
```

```
   ndxSz = flSize/sizeof(IndexEntry);

   try {  index = new IndexEntry[ndxSz+1];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadIndex", ProgXptn::allocFailed );  }

   bin.seekg( 0 );
   for ( int k(0); k < ndxSz; ++k )
      bin.read( reinterpret_cast<char*>(&index[k]), sizeof(IndexEntry) );
   bin.close();
} // loadIndex
```

Με τις:

```
   bin.seekg( 0, ios_base::end );
   unsigned int flSize( bin.tellg() );
   ndxSz = flSize/sizeof(IndexEntry);
```

βρίσκουμε το μέγεθος του αρχείου και υπολογίζουμε το πλήθος των στοιχείων που περιέχονται σε αυτό.

Στη συνέχεια παίρνουμε την απαιτούμενη δυναμική μνήμη. Το "**+1**" μας δίνει μια θέση στον πίνακα για τον φρουρό που χρησιμοποιεί η *linSearch()*.

Τέλος, με τη **for** φορτώνουμε το περιεχόμενο του αρχείου στον πίνακα.

Η ανάκτηση και η επίδειξη των στοιχείων των φοιτητών γίνεται με τη:

```
void retrieve( string flNm, PIndexEntry index, int ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "retrieve", ProgXptn::cannotOpen, flNm.c_str() );

   string line;
   cout << "Student Id Number: ";  getline( cin, line, '\n' );
   while ( line != "ΤΕΛΟΣ" )
   {
      int idNum( atoi(line.c_str()) );
      int ndx( linSearch(index, ndxSz, 0, ndxSz-1, IndexEntry(idNum)) );
      if ( ndx < 0 )
         cout << "unknown Student Id Number" << endl;
      else
      {
         Student oneStudent;
         bin.seekg( index[ndx].loc );
         oneStudent.load( bin );
         oneStudent.display( cout );
      }
      cout << "Student Id Number: ";  getline( cin, line, '\n' );
   } // while
   bin.close();
} // retrieve
```

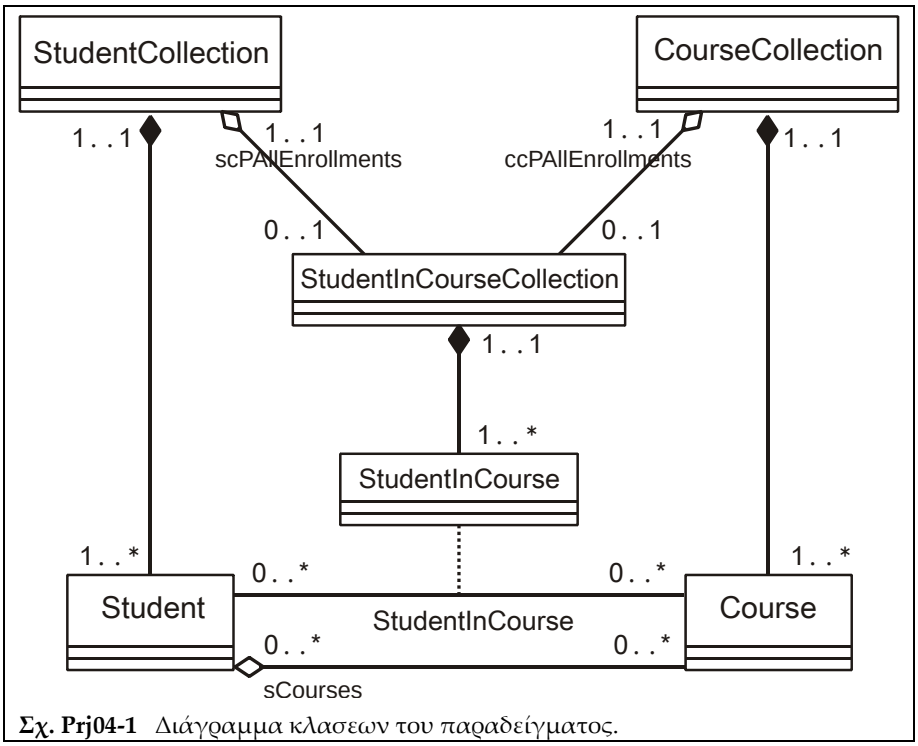
Η *retrieve()* τροφοδοτείται με τον index και το όνομα του αρχείου –με τα στοιχεία των φοιτητών– το οποίο και ανοίγει.

Στη συνέχεια ζητάει από τον χρήστη τον αριθμό μητρώου ενός φοιτητή και ψάχνει στον index να βρει στοιχείο με αυτόν. Αν το βρει –στη θέση *ndx*– φορτώνει τα στοιχεία του φοιτητή με τις:

```
         Student oneStudent;
         bin.seekg( index[ndx].loc );
         oneStudent.load( bin );
```

Στην §15.14.3 είχαμε υποσχεθεί ότι «*Αργότερα θα δούμε μια πολύ απλή μορφή ευρετηρίου.*» Εδώ πραγματοποιήσαμε αυτό που υποσχεθήκαμε. Αλλά... το παράδειγμά μας έχει πολύ περιορισμένη χρησιμότητα: Μόνο ανάκτηση πληροφοριών.

Δεν μπορούμε να κάνουμε και ενημέρωση; Μπορούμε, αρκεί να περιοριστούμε στα γενικά στοιχεία του φοιτητή και να μην αλλάξουμε το πλήθος μαθημάτων. Φυσικά δεν μπο-



Σχ. Prj04-1 Διάγραμμα κλασεων του παραδείγματος.

ρούμε να διαγράφουμε ή να προσθέτουμε νέους φοιτητές. Με τέτοιες ενημερώσεις χάνουμε τη δυνατότητα της σειριακής διαχείρισης του αρχείου. Για να τα έχουμε όλα χρειάζονται πέρα από το ευρετήριο και ειδικοί αλγόριθμοι που δεν έχουν σχέση με τους στόχους αυτού του βιβλίου.

Prj04.13 Για το Παράδειγμά μας

Με αυτό το παράδειγμα είχαμε την ευκαιρία να δούμε πώς χρησιμοποιούμε αυτά που μάθαμε μέχρι τώρα για κλάσεις και αντικείμενα. Στο Σχ. Prj04-1 βλέπεις και το σχετικό διάγραμμα κλάσεων.

Πόσο «πραγματικό» είναι το παράδειγμα; Όχι πολύ. Και αυτό δεν έχει σχέση μόνο με τα στοιχεία που θα πρέπει να κρατάει ένα αρχείο φοιτητών και λείπουν από το δικό μας. Ας δούμε τα άλλα προβλήματά του.

- Μια «πραγματική» εφαρμογή αυτού του είδους στήνεται με ένα *Σύστημα Διαχείρισης Βάσεων Στοιχείων* (Data Base Management Systems, DBMS) και όχι με απλά αρχεία. Πολλά από τα προβλήματα που αντιμετωπίσαμε[11] (και μερικά από αυτά που θα δούμε στη συνέχεια) αντιμετωπίζονται από το ΣΔΒΔ και η δουλειά του προγραμματιστή διευκολύνεται. Φυσικά, το ΣΔΒΔ κρατάει τους πίνακας στη ΒΔ, στον δίσκο, και όχι στη μνήμη.
- Το «σωστό» σχέδιο ήταν αυτό που είχαμε στο Project 3 και όχι το τωρινό.[12] Η αλήθεια είναι ότι στις «πραγματικές» εφαρμογές γίνονται τέτοιες «αποκανονικοποιήσεις» για να αυξήσουμε την ταχύτητα εκτέλεσης. Αλλά αυτές εισάγουν πλεονασμό και είδες πώς πληρώνεται ο πλεονασμός: οι μεγαλύτερες περιπλοκές σε αυτά που γράψαμε έχουν σχέση με τη συνεπή ενημέρωση του πίνακα δηλώσεων μαθημάτων και του πίνακα μαθημάτων του κάθε φοιτητή.

---

[11] Για παράδειγμα οι έλεγχοι ακεραιότητας οντότητας και αναφοράς.

[12] Στη γλώσσα του Σχεσιακού μοντέλου ΒΔ λέμε ότι ο πίνακας μαθημάτων μέσα στο αντικείμενο κλάσης *Student* παραβιάζει την *1η Κανονική Μορφή*.

- Θα μπορούσαμε να έχουμε τον πλεονασμό μόνο στην κύρια μνήμη αλλά όχι στα αρχεία: Φυλάγουμε στο αρχείο φοιτητών για κάθε φοιτητή μόνον επώνυμο, όνομα και αριθμό μητρώου. Κατα τη φόρτωση των στοιχείων, τα υπόλοιπα συμπληρώνονται όταν φορτώνουμε το αρχείο δηλώσεων μαθημάτων. Έτσι δεν έχουμε εγγραφές μεταβλητού μήκους και δεν χρειαζόμαστε ευρετήριο (τουλάχιστον για τον λόγο αυτόν).
- Και κάτι ακόμη: Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
```

- Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)

- Σε ένα τέτοιο σύστημα προγραμμάτων και δεδομένων, σε πραγματικές συνθήκες, θα πρέπει να έχουν ταυτόχρονη πρόσβαση πολλοί χρήστες –για παράδειγμα υπάλληλοι της γραμματείας του τμήματος και πιθανότατα οι ίδιοι οι ενδιαφερόμενοι φοιτητές– χωρίς όμως να καταστρέφουν ο ένας τη δουλειά του άλλου. Οι κίνδυνοι προέρχονται από τον τρόπο λειτουργίας των πολυχρηστικών ΛΣ που δίνουν εκ περιτροπής χρόνο εξυπηρέτησης σε όλους τους χρήστες αλλά με πολύ μεγάλη –για τον άνθρωπο– ταχύτητα ώστε να μην ενοχλείται ο χρήστης. Σκέψου λοιπόν την εξής περίπτωση:
  - Υπάλληλος της γραμματείας ζητάει φύλαξη των πινάκων (με τη συνάρτηση *saveCollections()*) του πρώτου προγράμματος· φυλάγεται ο πίνακας μαθημάτων, φυλάγεται το μητρώο των φοιτητών και…
  - Στο σημείο εκείνο το ΛΣ διακόπτει την εκτέλεση του προγράμματος και εξυπηρετεί την απαίτηση ενός φοιτητή που αλλάζει τις δηλώσεις μαθημάτων.
  - Όταν θα επιστρέψει για να συνεχίσει τη φύλαξη θα φυλάξει τον πίνακα δηλώσεων που όμως δεν είναι συνεπής με αυτούς που είχαν φυλαχθεί πριν τη διακοπή.

  Ένα ΣΔΒΔ, με τη δυνατότητα **επεξεργασίας συναλλαγών** (transaction processing) που σου παρέχει, σου επιτρέπει να απαλλαγείς με σχετικώς απλό τρόπο από τέτοιες επι-

πλοκές. Αλλιώς θα πρέπει να καταφύγεις στη χρήση εργαλείων που δίνει το ΛΣ (σημαφόροι, κλειδώματα κ.ά.) που κάνουν το πρόγραμμά σου πολύ πιο πολύπλοκο.

project

5

Σύνολα Γραμμάτων

Περιεχόμενα:

Prj05.1 Το Πρόβλημα

Το Project αυτό είναι ένα μεγάλο παράδειγμα επιφόρτωσης τελεστών.

Α. Θέλουμε να υλοποιήσουμε έναν τύπο συνόλων που στοιχεία τους θα είναι Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου. Για την υλοποίηση επιλέγουμε τη χρήση ψηφιοχάρτη (bitmap). Δηλαδή ένα σύνολο θα παριστάνεται με τα πρώτα 26 bits μιας τιμής τύπου `long int` *και:*

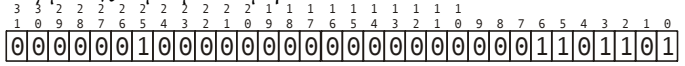
– το bit 0 έχει τιμή 1 αν και μόνον αν το `'A'` *ανήκει στο σύνολο,*

– το bit 1 έχει τιμή 1 αν και μόνον αν το `'B'` *ανήκει στο σύνολο,*

. . .

– το bit 25 έχει τιμή 1 αν και μόνον αν το `'Z'` *ανήκει στο σύνολο,*

Αν για παράδειγμα έχουμε μια τιμή:



αυτή παριστάνει το σύνολο { `'A'`, `'C'`, `'D'`, `'F'`, `'G'`, `'Z'` }. *Προφανώς το* { } *παριστάνεται με την τιμή 0.*

Γράψε μια κλάση, ας την πούμε SetOfUCL,[1] *που θα υλοποιεί το σύνολό μας με τον παραπάνω τρόπο. Η κλάση θα περιέχει μεθόδους για:*

(x, y σύνολα, u άτομο)

– πληθάριθμο: **#***x,*

– εισαγωγή στοιχείου σε σύνολο: x **+=** *u και x* **+** *u,*

– διαγραφή στοιχείου από σύνολο: x **~=** *u και x* **~** *u,*

[1] *Set of U*pper Case *L*etters

– ένωση συνόλων: x ∪= *y και x* ∪ *y,*
– τομή συνόλων: x ∩= *y και x* ∩ *y,*
– διαφορά συνόλων: x \= *y και x* \ *y,*
Φυσικά θα πρέπει να γράψεις και όποιες άλλες μεθόδους ή καθολικές συναρτήσεις θεωρείς αναγκαίες.
Υπόδ.: Για ευκολία μπορείς να υποθέσεις ότι όλα τα Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου βρίσκονται σε συναπτές θέσεις του πίνακα χαρακτήρων κατά αλφαβητική σειρά: αν το `'A'` *βρίσκεται στη θέση p, τότε το* `'B'` *βρίσκεται στη θέση p+1, το* `'C'` *βρίσκεται στη θέση p+2 κ.ο.κ.*
Δίνεται αρχείο text, με όνομα στον δίσκο `mstrpc.txt`*, που περιέχει κείμενο στην αγγλική γλώσσα. Τα γράμματα είναι κεφαλαία και πεζά αλλά για μας είναι ίδια:* `'A'` *και* `'a'` *είναι το ίδιο πράγμα (μια καλή ιδέα είναι η εξής: όταν τα διαβάζουμε, πριν από οποιαδήποτε άλλη δουλειά, τα κάνουμε κεφαλαία). Το κείμενο είναι χωρισμένο σε παραγράφους. Κάθε παράγραφος (εκτός από την τελευταία) χωρίζεται από την επόμενή της με μια κενή γραμμή (δηλαδή: αν διαβάσουμε* `'\n''\n'` *αλλάζει παράγραφος).*
Θέλουμε ένα πρόγραμμα που θα δημιουργεί ένα άλλο αρχείο, μη μορφοποιημένο, με όνομα στο δίσκο `paragltr.dta`*, που για κάθε παράγραφο θα έχει τρεις τιμές κλάσης setOfUCL:*
– το σύνολο των γραμμάτων της παραγράφου (τα βλέπουμε όλα ως κεφαλαία),
– το σύνολο των γραμμάτων της παραγράφου που υπάρχουν και στην επόμενη παράγραφο,
– το σύνολο των γραμμάτων της παραγράφου που δεν υπάρχουν στην επόμενη παράγραφο.

Prj05.2 Παίρνοντας Ιδέες από την Pascal

Πριν προχωρήσουμε στην υλοποίηση της κλάσης, ας ρίξουμε μια ματιά στη διαχείριση συνόλων που επιτρέπει η Pascal που επιτρέπει δηλώσεις της μορφής:

```
x, y, z: set of T;
```

Μετά από αυτό οι *x*, *y*, *z* είναι σύνολα με στοιχεία τύπου *T* (τύπος βάσης). Οι πράξεις συνόλων γίνονται με τους τελεστές:

- "`+`" για την ένωση. Το "`x + y`" παριστάνει το "*x* ∪ *y*"
- "`*`" για την τομή. Το "`x * y`" παριστάνει το "*x* ∩ *y*"
- "`-`" για τη διαφορά. Το "`x - y`" παριστάνει το "*x* \ *y*"

Με τις "`x = y`" και "`x <> y`" συγκρίνονται τα *x*, *y* για ισότητα και ανισότητα αντιστοίχως.

Αν *u* τιμή τύπου *T*, το "`u in x`" είναι συνθήκη που παίρνει τιμή `true` αν η *u ανήκει* στο *x* και `false` αν δεν ανήκει.

Συνθήκη είναι και η "`x <= y`" που παίρνει τιμή `true` αν το *x* είναι *υποσύνολο* του *y* και `false` αν δεν είναι.[2] Η σύγκριση μπορεί να γίνει και με την "`y >= x`" (το *y* είναι *υπερσύνολο* του *x*).

Στην Extended Pascal παρέχεται και η συνάρτηση *card*: Με το "`card(x)`" παίρνουμε τον **πληθάριθμο** (cardinality) του *x* (#*x*).

Δεν είναι κακή ιδέα να επιφορτώσουμε τους αντίστοιχους τελεστές με το ίδιο νόημα για την κλάση που έχουμε να γράψουμε.

---

[2] *Μην* υποθέσεις ότι με το "`x < y`" ελέγχουμε για *γνήσιο υποσύνολο*. Αν τα *x*, *y* είναι σύνολα η Pascal δεν επιτρέπει να γράψεις τέτοια παράσταση.

Prj05.3 Η Κλάση και οι Μέθοδοι

Ξεκινούμε με την αναλλοίωτη της κλάσης η οποία είναι τετριμμένη: `true`. Σύμφωνα με τους κανόνες μας θα πρέπει να γράψουμε `struct` (όλα ανοικτά) και όχι `class`:

```
struct SetOfUCL
{
   unsigned long int bitmap;
// . . .
}; // SetOfUCL
```

Με το πρόβλημα αυτό ασχοληθήκαμε στο Παράδ. 3 της §19.5 όπου λέγαμε «*Αλλά αν επιλέξουμε* `struct` *έχουμε το εξής πρόβλημα: Δίνουμε τη δυνατότητα στο πρόγραμμα που τη χρησιμοποιεί να χειρίζεται την τιμή του (μοναδικού) μέλους ενώ η κλάση γράφεται για να του δώσει τη δυνατότητα να χειρίζεται σύνολα και τα μέλη τους. Πώς αποφεύγεται κάτι τέτοιο;*

- *Με το να την κάνουμε* `class`,
- *να βάλουμε το "*`long int bitmap`*" («μυστικό» της υλοποίησης) σε περιοχή* `private` *και*
- *να μην γράψουμε μεθόδους getBitmap ούτε setBitmap.*»:

```
class SetOfUCL
{
public:
// . . .
private:
   unsigned long int bitmap;
}; // SetOfUCL
```

Όπως λέει και το όνομά της, θα χειριζόμαστε τη *bitmap* ως ψηφιοπίνακα με τις συναρτήσεις (*bitValue(), setBit(), clearBit()* κλπ) που μάθαμε στην §17.6.

Η πιο απλή μορφή της κλάσης, όπως την υποδεικνύει η διατύπωση των απαιτήσεων:

```
class SetOfUCL
{
public:
   SetOfUCL() {  bitmap = 0UL;  };
   ~SetOfUCL() { };
private:
   long int bitmap;
}; // SetOfUCL
```

- Όπως βλέπεις, ορίσαμε και τον ερήμην δημιουργό που μας δίνει το κενό σύνολο. Αν δηλαδή δηλώσουμε:

```
   SetOfUCL x;
// x == { }
```

το *x* είναι κενό.

- Ο «κενός» καταστροφέας μας θυμίζει ότι δεν χρειάζεται να ορίσουμε δημιουργό αντιγραφής, τελεστή εκχώρησης και καταστροφέα μια και αυτοί που θα ορίσει αυτομάτως ο μεταγλωττιστής κάνουν τη δουλειά μας.

Είναι χρήσιμο να έχουμε και έναν δημιουργό (με αρχική τιμή) για δημιουργία *μονοσυνόλων*. Τον δηλώνουμε:

```
   explicit SetOfUCL( char c );
```

και τον γράφουμε αντιγράφοντας από τη *setBit()*.

Σε ποιο δυαδικό ψηφίο θα βάλουμε την τιμή "1"; Αφού το `'A'` αντιστοιχεί στο δυαδικό ψηφίο 0, ένα τυχόν κεφαλαίο γράμμα *c* θα αντιστοιχεί στο δυαδικό ψηφίο:

```
static_cast<int>(c) - static_cast<int>('A')
```

Και αν ο *c* δεν είναι κεφαλαίο γράμμα τί κάνουμε; Τότε *πρέπει να ρίξουμε εξαίρεση* αφού δεν μπορούμε να ανταποκριθούμε:

```
SetOfUCL::SetOfUCL( char c )
{
   if ( !isupper( c ) )
      throw SetOfUCLXptn( "SetOfUCL", SetOfUCLXptn::nonUCL, c );
```

```
   bitmap = 1;
   bitmap <<= ( static_cast<int>(c) - static_cast<int>('A') );
} // SetOfUCL::SetOfUCL
```

Η

```
   SetOfUCL oneSet( 'S' );
```

δημιουργεί ένα (μονο)σύνολο με μοναδικό στοιχείο το `'S'`.

Παρατηρήσεις: ▶

1. Προφανώς θα μπορούσαμε να γράψουμε:

```
   bitmap <<= ( c - 'A' );
```

αλλά προτιμούμε τη γραφή με την τυποθεώρηση για να σου υπενθυμίζουμε τι γίνεται. Είτε με τη μια γραφή είτε με την άλλη, βασιζόμαστε στα εξής:

- *«Για ευκολία μπορείς να υποθέσεις ότι όλα τα Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου βρίσκονται σε συναπτές θέσεις του πίνακα χαρακτήρων κατά αλφαβητική σειρά: αν το* `'A'` *βρίσκεται στη θέση p, τότε το* `'B'` *βρίσκεται στη θέση p+1, το* `'C'` *βρίσκεται στη θέση p+2 κ.ο.κ.»*
- *«Το bit 0 έχει τιμή 1 αν και μόνον αν το* `'A'` *ανήκει στο σύνολο.»*

2. Αν δεν θέλεις να αντιγράψεις τη *setBit()* μπορείς απλώς να την καλέσεις:

```
SetOfUCL::SetOfUCL( char c )
{
   if ( !isupper( c ) )
      throw SetOfUCLXptn( "SetOfUCL", SetOfUCLXptn::nonUCL, c );

   bitmap = 0;
   setBit( bitmap, static_cast<int>(c) - static_cast<int>('A') );
} // SetOfUCL::SetOfUCL
```

Φυσικά, στην περίπτωση αυτή θα πρέπει να φέρεις στο πρόγραμμά σου το περίγραμμα *setBit()*.

Το περίγραμμα της *setBit()* μπορεί να ρίξει και κάποια εξαίρεση· μήπως πρέπει να την πιάσουμε και να ρίξουμε μια *SetOfUCLXptn*; Όχι! Δεν υπάρχει περίπτωση να τη ρίξει! ◀

Μεθόδους "*get*" και "*set*" θα γράψουμε; Για τη *bitmap* ούτε λόγος. Πάντως τον ρόλο "*get*" θα παίξει το κατηγόρημα «*ανήκειν*» (∈) και ρόλο "*set*" θα παίξουν οι μέθοδοι *εισαγωγής στοιχείου σε σύνολο: x* `+=` *u* και *διαγραφής στοιχείου από σύνολο: x* `-=` *u*.

Όπως συνηθίζουμε, για να μπορούμε να κάνουμε τις δοκιμές μας με την κλάση, την εξοπλίζουμε με μια μέθοδο:

```
void SetOfUCL::display( ostream& tout ) const
{
   unsigned long int x( 1 );

   for ( int k(0); k <= 25; ++k )
   {
      if ( (bitmap & x) != 0 )
         tout << static_cast<char>( k + static_cast<int>('A') );
      x <<= 1;
   } // for
   tout << endl;
} // SetOfUCL::display
```

Prj05.3.1 Πληθάριθμος: #*x*

Η πρώτη μέθοδος που μας ζητείται είναι αυτή που δίνει τον *πληθάριθμο* ενός συνόλου *x* (`#`*x*). Ακολουθώντας την Extended Pascal την ονομάζουμε *card*:

x`.`*card*`() == #`*x*

Αφού για κάθε στοιχείο που περιέχεται στο σύνολο θα βάζουμε "1" στο αντίστοιχο δυαδικό ψηφίο του *bitmap*, αρκεί να μετρήσουμε πόσα "1" υπάρχουν στο *bitmap*. Επομένως, μπορούμε να χρησιμοποιήσουμε το περίγραμμα συνάρτησης *count1*:

```
unsigned int SetOfUCL::card() const
{  return count1( bitmap );  } // SetOfUCL::card
```

ή να το αντιγράψουμε με την κατάλληλη προσαρμογή:

```
unsigned int SetOfUCL::card() const
{
   unsigned long int x( 1 );
   int fv( 0 ), lastb( 8*sizeof(unsigned long int)-1 );

   for ( int k(0); k <= lastb; ++k )
   {
      if ( (bitmap & x) != 0 )  ++fv;
      x <<= 1;
   } // for
   return fv;
} // SetOfUCL::card
```

Prj05.3.2Ένωση Συνόλων – Εισαγωγή Στοιχείου σε Σύνολο

Τι σχέση έχει η ένωση συνόλων με την εισαγωγή στοιχείου σε σύνολο; Η εισαγωγή είναι ειδική περίπτωση της ένωσης. Πράγματι, οι προδιαγραφές για την εισαγωγή είναι οι εξής:

true { εισαγωγή του u στο x } $u \in x$

Αλλά τις ίδιες προδιαγραφές έχει και η "$x = x \cup \{ u \}$" ($x \cup= \{ u \}$):

true { $x = x \cup \{ u \}$ } $u \in x$

Αν πάρουμε υπόψη μας ακόμη ότι από την "**operator@=**" μπορούμε να πάρουμε την υλοποίηση της "**operator@**" (επιφόρτωση του "**@**") καταλαβαίνεις ότι το πρώτο που έχουμε να κάνουμε είναι η υλοπόιηση του "$\cup=$". Και –αφού είπαμε ότι θα ακολουθήσουμε τους συμβολισμούς της Pascal– θα επιφορτώσουμε τον "**operator+=**". Όπως είπαμε στην §22.6 η επιφόρτωση θα πρέπει να γίνει με μέθοδο της κλάσης:

```
SetOfUCL& SetOfUCL::operator+=( const SetOfUCL& rhs )
{
   bitmap = bitmap | rhs.bitmap;
   return *this;
} // SetOfUCL::operator+=
```

Αν θέλεις, μπορείς να υλοποιήσεις και μια εκδοχή με όνομα αλλά πρόσεχε: *το "**union**" δεν επιτρέπεται*! Ας πούμε λοιπόν:

```
SetOfUCL& SetOfUCL::setUnion( const SetOfUCL& rhs )
{
   return ( *this += rhs );
} // SetOfUCL::setUnion
```

Πώς λύθηκε το πρόβλημα της εισαγωγής; Αν το x είναι αντικείμενο κλάσης *SetOfUCL* τότε με την

```
   x += SetOfUCL( 'S' );
```

υλοποιείς την $x = x \cup \{$ **'S'** $\}$.

Πώς λύθηκε το πρόβλημα της καθολικής συνάρτησης; Όπως μάθαμε στην §22.7.1:

```
SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv += rhs;
   return fv;
} // SetOfUCL operator+
```

Πρόσεξε ότι αυτή η συνάρτηση *δεν χρειάζεται να δηλωθεί ως* **friend**.

Η διατύπωση του προβλήματος όμως απαιτεί να κάνουμε εισαγωγή γράφοντας "`x += 'S'`". Μπορούμε να έχουμε αυτήν τη δυνατότητα; Ναι, με δύο τρόπους:

- Να βγάλουμε το "`explicit`" από τον δημιουργό μονοσυνόλου.
- Να κάνουμε δεύτερη επιφόρτωση του "`+=`".

Αν επιλέξουμε την πρώτη λύση θα είναι δυνατές (μετά την επιφόρτωση του "`==`") σύγκρισεις σαν την "`x == 'S'`" που είναι απαράδεκτη. Έτσι, θα πάμε στη δεύτερη λύση:

```
SetOfUCL& SetOfUCL::operator+=( char c )
{
   if ( !isupper(c) )
      throw SetOfUCLXptn( "operator+=", SetOfUCLXptn::nonUCL, c );

   *this += SetOfUCL( c );
   return *this;
} // SetOfUCL::operator+=
```

Και εδώ μπορούμε να δώσουμε και μια εκδοχή με όνομα:

```
SetOfUCL& SetOfUCL::insert( char c )
{
   return ( *this += c );
} // SetOfUCL::insert
```

Από τη μέθοδο εισαγωγής χαρακτήρα μπορείς να πάρεις –με την πάγια τεχνική– και καθολική συνάρτηση εισαγωγής χαρακτήρα:

```
SetOfUCL operator+( const SetOfUCL& lhs, char rhs )
```

Στα παραπάνω, πρόσεξε το εξής ουσιώδες: *Το νόημα του τελεστή "`+`" δίνεται μόνο μια φορά, στην επιφόρτωση του τελεστή "`+=`"*. Όλοι οι άλλοι ορισμοί ανάγονται αμέσως ή εμμέσως στον "`+=`". Θα πει κανείς: Αν τα ορίσουμε όλα εξ αρχής, γράφοντας πράξεις με ψηφιοχάρτες, θα κάναμε τις συναρτήσεις μας πιο γρήγορες· και μόνο το ότι θα αποφεύγαμε τόσες κλήσεις συναρτήσεων θα ήταν σημαντικό κέρδος. Ναι, αλλά θα είχαμε περισσότερες πιθανότητες για λάθη και η νοηματική συνέπεια όλων αυτών δεν θα ήταν (σχεδόν) αυταπόδεικτη, όπως είναι τώρα.

Παράδειγμα

Οι εντολές

```
   SetOfUCL x( 'A' ), y( x ), z;

   x += 'B';  x.insert( 'C' );
   x.display( cout );
   y += 'C';  y.insert( 'D' );  y += 'E';
   y.display( cout );
   z = x + y;
   z.display( cout );
   x.setUnion( y );
   x.display( cout );
   cout << x.card() << endl;
```

δίνουν:

```
ABC
ACDE
ABCDE
ABCDE
5
```

Prj05.3.3 Διαφορά Συνόλων – Διαγραφή Στοιχείου Συνόλου

Αφου η διαφορά συνόλων $x \setminus y$ είναι το σύνολο των στοιχείων του x που δεν ανήκουν στο y, η διαφορά σχετίζεται με τη διαγραφή στοιχείου συνόλου όπως σχετίζεται η ένωση με την εισαγωγή:

`true` { διαγραφή του u από το x } $u \notin x$

`true` { $x = x \setminus \{ u \}$ } $u \notin x$

Θα πρέπει λοιπόν να ξεκινήσουμε και εδώ με την επιφόρτωση του "`-=`" με μια μέθοδο που θα υλοποιεί την "`\=`". Να σημειώσουμε ότι καταφεύγουμε στον "`-`" που χρησιμοποιεί η Pascal και αγνοούμε τον "`~`" που χρησιμοποιείται στη διατύπωση του προβλήματος διότι ο "`~`" δεν είναι κατάλληλος: είναι *ενικός*!

Μπορούμε να δούμε τη διαφορά $x \setminus y$ ως την τομή, με το x, του συνόλου των στοιχείων που ανήκουν στο x ή στο y (αλλά όχι και στα δύο):

```
SetOfUCL& SetOfUCL::operator-=( const SetOfUCL& rhs )
{
   bitmap = bitmap & ( bitmap ^ rhs.bitmap );
   return *this;
} // SetOfUCL::operator-=
```

και για όποιον προτιμάει συνάρτηση με όνομα:

```
SetOfUCL& SetOfUCL::setDifference( const SetOfUCL& rhs )
{
   return ( *this -= rhs );
} // SetOfUCL::setDifference
```

Από αυτήν παίρνουμε την καθολική συνάρτηση με τη πάγια τεχνική:

```
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv -= rhs;
   return fv;
} // operator-( SetOfUCL, char )
```

και τη μέθοδο διαγραφής:

```
SetOfUCL& SetOfUCL::operator-=( char c )
{
   if ( isupper(c) )
   {
      *this -= SetOfUCL( c );
   }
   return *this;
} // SetOfUCL::operator-=
```

Πρόσεξε ότι εδώ *δεν* ρίχνουμε εξαίρεση αν το c δεν είναι κεφαλαίο λατινικό γράμμα· είναι σίγουρο ότι –και στην περίπτωση αυτή– μετά την εκτέλεση της πράξης $c \notin$ `*this`.

Η μέθοδος

```
SetOfUCL& SetOfUCL::remove( char c )
{
   return ( *this -= c );
} // SetOfUCL::remove
```

κάνει την ίδια δουλειά.

Παράδειγμα

Οι εντολές

```
   x.display( cout );  y.display( cout );
   y -= x;
   y.display( cout );
   y -= 'D';
   y.display( cout );
   y -= 'Q';
   y.display( cout );
   z = x - y;
   z.display( cout );
```

δίνουν:

```
ABC
ACDE
DE
```

```
E
E
ABC
```

Prj05.3.4 Τομή Συνόλων: $x \cap= y$

Ακολουθώντας την Pascal, επιφορτώνουμε τον "`*=`" για την υλοποίηση της πράξης "$\cap=$":

```
SetOfUCL& SetOfUCL::operator*=( const SetOfUCL& rhs )
{
   bitmap = bitmap & rhs.bitmap;
   return *this;
} // SetOfUCL::operator*=
```

Από αυτήν παίρνουμε την καθολική συνάρτηση:

```
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv *= rhs;
   return fv;
} // operator*( SetOfUCL, SetOfUCL )
```

Prj05.4 Το Πρόγραμμα

Ας ξεκινήσουμε με την παρατήρηση ότι: *από κάθε παράγραφο χρειαζόμαστε μόνον το σύνολο των γραμμάτων της* (αφού τα μετατρέψουμε σε κεφαλαία). Έχει νόημα λοιπόν να γράψουμε μια συνάρτηση

```
void readParagraph( ifstream& tin, SetOfUCL& letSet )
```

που θα διαβάζει μέσω του ρεύματος *tin* μια παράγραφο από ένα αρχείο text και θα μας δίνει το σύνολο *letSet* των γραμμάτων που έχει.

Πώς θα δουλεύει η συνάρτηση; Έτσι περίπου:

```
"Άδειασε" το letSet
Διάβασε έναν χαρακτήρα c
while ( δεν τελείωσε η παράγραφος )
{
   if ( ο c είναι γράμμα )
      Βάλε στο letSet το αντίστοιχο κεφαλαίο
   Διάβασε έναν χαρακτήρα c
}
```

Πώς αδειάζουμε το *letSet*; Θα πρέπει να μηδενίσουμε το *bitmap* της *letSet*. Θα πρέπει να εφοδιάσουμε την κλάση μας με μια κατάλληλη μέθοδο[3]:

```
   void clear()  {  bitmap = 0L; };
```

οπότε η "*«Άδειασε» το letSet*" γίνεται:[4]

```
   letSet.clear();
```

Διαβάζουμε τον *c* με την:

```
tin.get(c);
```

Η "*Βάλε στο letSet το αντίστοιχο κεφαλαίο*" γίνεται:

```
letSet.insert( toupper(c) );
```

Ας έλθουμε τώρα στις συνθήκες. Η "*ο c είναι γράμμα*" είναι απλή: `isalpha(c)`.

---

[3] Για να μπορούμε να ελέγχουμε αν ένα σύνολο είναι *κενό* γράφουμε μια:

```
bool isEmpty() const {  return ( bitmap == 0UL );  }
```

[4] Η *clear()* δεν είναι απαραίτητη. Αφού ο ερήμην δημιουργός δημιουργεί ένα κενό σύνολο, μπορούμε να «αδειάσουμε» το *letSet* γράφοντας απλώς: "`letSet = SetOfUCL()`".

Η *"δεν τελείωσε η παράγραφος"* θέλει λίγη σκέψη. Πότε τελειώνει η παράγραφος;

- Όταν βρούμε "`'\n''\n'`" ή
- όταν βρούμε τέλος αρχείου.

Για να μπορούμε να ανιχνεύουμε το "`'\n''\n'`" θα πρέπει να θυμόμαστε τον προηγούμενο χαρακτήρα ή να διαβάζουμε προκαταβολικά (*peek*) τον επόμενο. Ας προτιμήσουμε την πρώτη λύση που δεν αλλάζει και την κανονική ροή της ανάγνωσης. Δηλώνουμε:

```
char c, lastC;
```

και στη *lastC* θα κρατούμε την προηγούμενη τιμή της *c*:

```
lastC = c;  tin.get(c);
```

Έτσι, θα μπορούμε να ελέγξουμε:

```
if ( c == '\n' )
{
   eopar = ( lastC == '\n' );
}
```

Προηγουμένως έχουμε δηλώσει:

```
bool eopar;     // end of paragraph
```

Με βάση τα παραπάνω η *"δεν τελείωσε η παράγραφος"* γίνεται:

```
!tin.eof() && !eopar
```

Αρχικώς βάζουμε:

```
eopar = false;  lastC = '\0';
```

Να λοιπόν η συνάρτηση:

```
void readParagraph( ifstream& tin, SetOfUCL& letSet )
{
   bool eopar;     // end of paragraph
   char c, lastC;

   letSet.clear();
   eopar = false;  lastC = '\0';
   tin.get( c );
   while ( !tin.eof() && !eopar )
   {
      if ( isalpha( c ) )
      {
         letSet.insert( toupper( c ) );
      }
      else if ( c == '\n' )
      {
         eopar = ( lastC == '\n' );
      }
      lastC = c;  tin.get( c );
   } // while
} // readParagraph
```

Τώρα το πρόγραμμά μας γίνεται πιο εύκολο. Ας προσπαθήσουμε να κάνουμε ένα σχέδιο:

```
Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της
Διάβασε τη δεύτερη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

Τώρα θα πρέπει να ξανακάνουμε τα ίδια αλλά το ρόλο της πρώτης παραγράφου θα τον παίζει η δεύτερη και της δεύτερης η τρίτη. Έχουμε όμως μια διαφορά: τη δεύτερη παράγραφο την έχουμε διαβάσει ήδη και τα γράμματά της υπάρχουν στο *nextSet*. Θα δουλέψουμε λοιπόν ως εξής:

```
Βάλε curSet = nextSet
Διάβασε την τρίτη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

```
Βάλε curSet = nextSet
Διάβασε την τέταρτη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
. . .
```

Όταν τελειώσει αυτή η διαδικασία, διότι θα βρούμε τέλος αρχείου, θα έχουμε στο *curSet* το σύνολο των γραμμάτων της τελευταίας παραγράφου. Τι θα κάνουμε με αυτό; Το πιο φυσιολογικό είναι να θεωρήσουμε ότι υπάρχει και μια «κενή παράγραφος» (με κενό σύνολο γραμμάτων) μετά από αυτήν.

Να λοιπόν τι θα πρέπει να κάνουμε:

```
Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της
while ( !tin.eof() )
{
   Διάβασε την επόμενη παράγραφο και πάρε το σύνολο nextSet των
   γραμμάτων της
   Υπολόγισε τα σύνολα
   common = curSet ∩ nextSet και notInNext = curSet \ nextSet
   Γράψε στο αρχείο τα curSet, common, notInNext
   Βάλε curSet = nextSet
}
Βάλε nextSet = { }
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

Ας γράψουμε το πρόγραμμά μας. Στην αρχή δηλώνουμε:

```
   SetOfUCL curSet, nextSet, common, notInNext;
   ifstream tin( "mstrpc.txt" );
   ofstream bout( "paragltr.dta", ios::binary );
```

και αρχίζουμε τη μετάφραση:

Η «*Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της*» μεταφράζεται στην:

```
   readParagraph( tin, curSet );
```

Παρομοίως η «*Διάβασε την επόμενη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της*» γίνεται:

```
   readParagraph( tin, nextSet );
```

Η «*Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet*« γίνεται:

```
      common = curSet * nextSet;
      notInNext = curSet - nextSet;
```

Και με την «*Γράψε στο αρχείο τα curSet, common, notInNext*» τι κάνουμε;

Ένας τρόπος να την υλοποιήσουμε είναι να γράψουμε:

```
      bout.write( reinterpret_cast<char*>( &curSet ),
                  sizeof( curSet ) );
```

και στη συνέχεια τα παρόμοια για τα άλλα δύο σύνολα.

Ένας άλλος είναι να εφοδιάσουμε την κλάση μας με μια μέθοδο:

```
void SetOfUCL::save( ostream& bout ) const
{
   bout.write( reinterpret_cast<const char*>(&bitmap), sizeof(bitmap) );
} // SetOfUCL::save
```

και να έχουμε το πρόγραμμά μας πιο απλό:

```
      curSet.save( bout );
      common.save( bout );  notInNext.save( bout );
```

Να λοιπόν το πρόγραμμά μας:

```
#include <fstream>
#include <iostream>
#include <cctype>
```

```
#include "SetOfUCL.cpp"

using namespace std;

void readParagraph( ifstream& tin, SetOfUCL& letSet );

int main()
{
   SetOfUCL curSet, nextSet, common, notInNext;
   ifstream tin( "mstrpc.txt" );
   ofstream bout( "paragltr.dta", ios::binary );

   readParagraph( tin, curSet );
   while ( !tin.eof() )
   {
      readParagraph( tin, nextSet );
      common = curSet * nextSet;
      notInNext = curSet - nextSet;
      curSet.save( bout );
      common.save( bout );  notInNext.save( bout );
      curSet = nextSet;
   } // while
   tin.close();
   nextSet.clear();
   common = curSet * nextSet;
   notInNext = curSet - nextSet;
   curSet.save( bout );
   common.save( bout );  notInNext.save( bout );
   bout.close();
} // main

void readParagraph( ifstream& tin, SetOfUCL& letSet )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Prj05.4.1 Και η *SetOfUCL* Μέχρι Τώρα

Λύσαμε το πρόβλημά μας έχοντας την εξής κλάση:

```
class SetOfUCL
{
public:
   SetOfUCL() {  bitmap = 0UL;  };
   explicit SetOfUCL( char c );
   ~SetOfUCL() { };
   unsigned int card() const;
   SetOfUCL& operator+=( const SetOfUCL& rhs );
   SetOfUCL& setUnion( const SetOfUCL& rhs );
   SetOfUCL& operator+=( char c );
   SetOfUCL& insert( char c );
   SetOfUCL& operator-=( const SetOfUCL& rhs );
   SetOfUCL& setDifference( const SetOfUCL& rhs );
   SetOfUCL& operator-=( char c );
   SetOfUCL& remove( char c );
   SetOfUCL& operator*=( const SetOfUCL& rhs );
   SetOfUCL& setIntersection( const SetOfUCL& rhs );
   void clear()  {  bitmap = 0UL; };
   bool isEmpty() const  {  return ( bitmap == 0UL );  };
   void save( ostream& bout ) const;
private:
   unsigned long int bitmap;
}; // SetOfUCL

SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator+( const SetOfUCL& lhs, char rhs );
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs );
```

```
SetOfUCL operator-( const SetOfUCL& lhs, char rhs );
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs );
```

Prj05.5 Εμπλουτίζοντας την Κλάση

Συγκρίνοντας τα παραπάνω με αυτά που ξέρουμε από τη Θεωρία Συνόλων βλέπουμε ότι μας λείπουν δύο πάγιες διμελείς σχέσεις των συνόλων: η σχέση "*ανήκειν*":[5]

$\_ \in \_ : X \leftrightarrow \mathbb{P}\, X$

και η σχέση "*περιέχεσθαι*":

$\_ \subseteq \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$

όπου:

$\forall x, y: \mathbb{P}\, X \bullet (x \subseteq y \Leftrightarrow \forall u{:}X \bullet u \in x \Rightarrow u \in y)$

Ας ξεκινήσουμε από την "$\in$". Υπάρχει κάποιος τελεστής για να επιφορτώσουμε; Η C++ δεν έχει "`in`" και δεν φαίνεται να ταιριάζει κάποιος άλλος τελεστής. Θα πρέπει να χρησιμοποιήσουμε κάποιο όνομα, π.χ.:

"u `isMemberOf` x" ή "x `hasMember` u"

Αν θέλουμε να γράψουμε μια μέθοδο καλύτερα να χρησιμοποιήσουμε το δεύτερο αφού για την "$u \in x$" θα γράφουμε "`x.hasMember(u)`".

Θα υλοποιήσουμε τη μέθοδο που θα επιστρέφει τιμή τύπου *bool* –αφού είναι κατηγόρημα– και θα γράψουμε προσαρμόζοντας την *bitValue()*:

```
bool SetOfUCL::hasMember( char c ) const
{
   bool fv( false );

   if ( isupper( c ) )
   {
      unsigned long int x( 1UL );
      x <<= static_cast<int>(c) - static_cast<int>('A');
      fv = ( ( bitmap & x ) != 0 );
   }
   return fv;
} // SetOfUCL::hasMember
```

Πρόσεξε ότι όταν ο *c* δεν είναι κεφαλαίο γράμμα δεν ρίχνουμε εξαίρεση· λέμε απλώς ότι δεν ανήκει στο σύνολο.

Αν προτιμάς το πρώτο όνομα, καλύτερα γράψε μια καθολική συνάρτηση:

```
bool isMemberOf( char c, const SetOfUCL& rhs )
```

που θα την καλείς ως εξής: "`isMemberOf( u, x )`". Φυσικά, αυτήν θα πρέπει να δηλώσεις ως **friend** της κλάσης.[6]

Για την υλοποίηση της δεύτερης δεν μας διευκολύνει και τόσο το κατηγόρημα που δίνουμε παραπάνω. Η συνάρτηση που θα προκύψει θα είναι πολύ αργή. Ένας άλλος τρόπος είναι να στηριχθούμε στην ιδιότητα:

$$(x \subseteq y) \Leftrightarrow (x = x \cap y)$$

Έχουμε λοιπόν:

```
bool SetOfUCL::isSubset( SetOfUCL rhs ) const
{  return ( bitmap == ( bitmap & rhs.bitmap ) );  } // isSubset
```

Αν *x*, *y* είναι *SetOfUCL* τότε η συνθήκη "$x \subseteq y$" θα «μεταφράζεται» στο πρόγραμμά μας σε "`x.isSubset(y)`".

---

[5] Θυμίσου ότι κάθε σύνολο με στοιχεία από το (*σύνολο βάσης*) *X* είναι μέλος του *δυναμοσυνόλου* $\mathbb{P}\, X$ του *X*.

[6] Βέβαια, αν δεν φοβάσαι μπερδέματα, μπορείς να έχεις και τις δύο, αρκεί να ορίσεις:

```
bool isMemberOf( char c, const SetOfUCL& rhs )  {  return rhs.hasMember( c );  }
```

Η Pascal χρησιμοποιεί τον "<=" για το υποσύνολο. Μπορούμε λοιπόν να τον επιφορτώσουμε ως καθολική συνάρτηση:

```
bool operator<=( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return lhs.isSubset( rhs );
} // operator<=( SetOfUCL...
```

και αντί για "x.isSubset(y)" να γράφουμε "x <= y".

Αφού η Pascal χρησιμοποιεί και τον ">=" για το υπερσύνολο, μπορούμε να τον επιφορτώσουμε χρησιμοποιώντας την *isSubset()*:

```
bool operator>=( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return rhs.isSubset( lhs );
} // operator<=( SetOfUCL...
```

Δεν πρέπει να παραλείψουμε να επιφορτώσουμε τον "==" για να μπορούμε να συγκρίνουμε δύο σύνολα για ισότητα:

```
bool operator==( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return ( lhs.bitmap == rhs.bitmap );
} // operator==( const SetOfUCL
```

και

```
bool operator!=( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return ( !(lhs==rhs) );
} // operator==
```

Πρόσεξε ότι η πρώτη θα πρέπει να δηλωθεί στην κλάση ως **friend**.[7]

Αν θέλουμε να έχουμε τη δυνατότητα να ελέγχουμε για γνήσιο *υποσύνολο* (proper subset) γράφουμε:

```
bool SetOfUCL::isProperSubset( const SetOfUCL& rhs ) const
{
   return ( this->isSubset(rhs) && (bitmap != rhs.bitmap) );
} // isProperSubset
```

και επιφορτώνουμε:

```
bool operator<( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return lhs.isProperSubset( rhs );
} // SetOfUCL::operator<
```

Παρατήρηση: ▶
Εδώ προσοχή: *Η "x < y" και η "x >= y" δεν έχουν την παραμικρή σχέση!* Το ότι το *x* δεν είναι γνήσιο υποσύνολο του *y* –δηλαδή δεν ισχύει η "x < y"– δεν σημαίνει ότι το *x* είναι υπερσύνολο του *y*. Μάλλον η ιδέα να επιφορτώσουμε τον "<" δεν είναι και τόσο καλή! ◀

Τελικώς, το περιεχόμενο του SetOfUCL.h έχει γίνει:

```
#ifndef _SETOFUCL_H
#define _SETOFUCL_H

#include <string>
#include <fstream>

using namespace std;

class SetOfUCL
{
friend bool operator==( const SetOfUCL& lhs, const SetOfUCL& rhs );
public:
   SetOfUCL() {  bitmap = 0UL;  };
```

[7] Να γράψουμε "return (lhs.isSubset(rhs) && rhs.isSubset(lhs))" για να αποφύγουμε το **friend**; Ε, όχι και έτσι...

```
   explicit SetOfUCL( char c );
   ~SetOfUCL() { };
   unsigned int card() const;
   SetOfUCL& operator+=( const SetOfUCL& rhs );
   SetOfUCL& setUnion( const SetOfUCL& rhs );
   SetOfUCL& operator+=( char c );
   SetOfUCL& insert( char c );
   SetOfUCL& operator-=( const SetOfUCL& rhs );
   SetOfUCL& setDifference( const SetOfUCL& rhs );
   SetOfUCL& operator-=( char c );
   SetOfUCL& remove( char c );
   SetOfUCL& operator*=( const SetOfUCL& rhs );
   SetOfUCL& setIntersection( const SetOfUCL& rhs );
   void clear()  {  bitmap = 0UL; };
   bool isEmpty() const  {  return ( bitmap == 0UL );  };
   void save( ostream& bout ) const;

   bool hasMember( char c ) const;
   bool isSubset( SetOfUCL rhs ) const;
   bool isProperSubset( const SetOfUCL& rhs ) const;

   void display( ostream& tout ) const;
private:
   unsigned long int bitmap;
}; // SetOfUCL

SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator+( const SetOfUCL& lhs, char rhs );
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator-( const SetOfUCL& lhs, char rhs );
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs );

bool operator<=( const SetOfUCL& lhs, const SetOfUCL& rhs );
bool operator<( const SetOfUCL& lhs, const SetOfUCL& rhs );
bool operator!=( const SetOfUCL& lhs, const SetOfUCL& rhs );

struct SetOfUCLXptn
{
   enum { nonUCL };
   char funcName[100];
   int  errorCode;
   char errCharVal;
   SetOfUCLXptn( const char* mn, int ec, char c=0 )
      :  errorCode( ec ),  errCharVal( c )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // SetOfUCLXptn
```

Prj05.6 Σχόλια και Παρατηρήσεις

Ας επαναλάβουμε και εδώ μερικές παρατηρήσεις σχετικώς με τις διάφορες επιλογές που κάναμε αναπτύσσοντας την κλάση.

- Για την ένωση και τη διαφορά ορίσαμε (επιφορτώσαμε) τους "`+=`" και "`-=`" και από αυτούς πήραμε όλους τους άλλους τελεστές (και σχετικές μεθόδους) σχεδόν με «μηχανικό τρόπο». Έτσι έχουμε εξασφαλισμένο ότι ο "`+`" έχει παντού το ίδιο νόημα (το ίδιο ισχύει και για τον "`-`"). Αν κάτι πρέπει να αλλάξει, αυτό θα γίνει σε ένα σημείο μόνο.
- Η τεχνική που μάθαμε στην §22.7.1 και εφαρμόσαμε για να πάμε από τον "`+=`" στον "`+`" και από τον "`-=`" στον "`-`" έχει έναν περιορισμό: δεν μπορεί να επιστρέψει τύπο αναφοράς. Πρόσεξε την περίπτωση με τη διαφορά:

```
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv -= rhs;
```

```
   return fv;
} // operator-( SetOfUCL, SetOfUCL )
```

Όταν κληθεί η `operator-()` δημιουργείται η *fv* που

- είναι τοπική στη συνάρτηση και
- ζει όσο εκτελείται η συνάρτηση.

Όταν έλθει η στιγμή να χρησιμοποιηθεί η τιμή που επιστρέφει –για παράδειγμα εκτελεσθεί η εκχώρηση στη *notInNext* στην "`notInNext = curSet - nextSet`"– η εκτέλεση της `operator-()` έχει τελειώσει, η *fv* δεν υπάρχει πια και η συνάρτηση επιστρέφει ένα βέλος προς μια ανύπαρκτη μεταβλητή.

Τι θα μπορούσαμε να κάνουμε; Να κάνουμε την *fv* δυναμική και να γράψουμε τη συνάρτηση έτσι ώστε να επιστρέφει βέλος προς αυτήν.

- Κατά τα άλλα η τεχνική αυτή δεν είναι μόνο για τελεστές. Ας πούμε ότι έχουμε την

```
SetOfUCL& SetOfUCL::setDifference( const SetOfUCL& rhs )
{
   return ( *this -= rhs );
} // SetOfUCL::setDifference
```

και θέλουμε να ορίσουμε και την καθολική συνάρτηση *setDifference()* που θα κάνει την ίδια δουλειά με την `operator-()`. Γράφουμε:

```
SetOfUCL setDifference( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv.setDifference( rhs );
   return fv;
} // setDifference( SetOfUCL, SetOfUCL )
```

κεφάλαιο

23

Κληρονομιές

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα γνωρίσουμε την έννοια της παραγωγής μιας κλάσης από μια άλλη και της κληρονομιάς που παίρνει η νέα από την παλιά.
Προσδοκώμενα αποτελέσματα:
Εξοικείωση με την κλήρονομικότητα και τις σχετικές προγραμματιστικές τεχνικές.
Έννοιες κλειδιά:

- *βασική κλάση*
- *παράγωγη κλάση*
- *σχέση* `is_a`
- *προστατευόμενα (*`protected`*) μέλη*
- *εικονικές (*`virtual`*) μέθοδοι*
- *πολυμορφισμός*
- *δυναμική τυποθεώρηση*
- *εξακρίβωση τύπου κατά την εκτέλεση -- RTTI*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Ένα βασικό χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού είναι η κληρονομικότητα: η δυνατότητα μιας κλάσης να δηλώνεται με βάση μιαν άλλη κλάση που προϋπάρχει και να έχει ως **κληρονομιά** (inheritance) τα μέλη και τις μεθόδους της αρχικής.

Στη συνέχεια θα αποφύγουμε τις πολλές θεωρίες και δούμε τα περί κληρονομιάς με παραδείγματα σε C++ (αφού, καλώς ή κακώς, οι τρόποι υλοποίησης των βασικών εννοιών στη C++ έχουν το ίδιο μεγάλο ενδιαφέρον με τις ίδιες τις έννοιες.)

23.1 Κτίζοντας Πάνω στα Υπάρχοντα

Η ημερομηνία ως χρονικός προσδιορισμός δεν είναι πάντοτε επαρκής. Οι περιπτώσεις που χρειαζόμαστε μεγαλύτερη ακρίβεια είναι πολλές. Αφού η *Date* δεν επαρκεί ας γράψουμε μια νέα κλάση, ας την πούμε *DateTime*, που θα πηγαίνει μέχρι δευτερόλεπτο:

```
class DateTime
{
public:
// . . .
private:
   unsigned int  dtYear;
   unsigned char dtMonth;
   unsigned char dtDay;
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Τι θα κάνουμε με τα *dtYear*, *dtMonth*, *dtDay*; Αυτό μπορούμε να το αντιγράψουμε από τη *Date*, αλλά δεν χρειάζεται! Δες έναν άλλον τρόπο να γράψουμε τα παραπάνω με πολύ «περισσότερο περιεχόμενο»:

```
class DateTime: public Date
{
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
// . . .
private:
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Αφήνοντας κατά μέρος τον δημιουργό, πού βρίσκεται το «περισσότερο περιεχόμενο»; Μας το δείχνει το παρακάτω προγραμματάκι:

```
   DateTime dt( 2007, 11, 26, 19, 30, 31 );
   cout << dt.getDay() << '.' << dt.getMonth() << '.' << dt.getYear() << endl;
   cout << dt << endl;
   dt += 3;          cout << dt << endl;
   dt.forward( 3 );  cout << dt << endl;
   ++dt;             cout << dt << endl;
   cout << (dt < Date(2008)) << endl;
```

που μας δίνουν:

```
26.11.2007
26.11.2007
26.11.2007
26.11.2007
27.11.2007
1
```

Η *dt* δηλώθηκε μεν κλάσης *DateTime* αλλά συμπεριφέρεται πλήρως ως αντικείμενο κλάσης *Date*! Ή αλλιώς:

- Η κλάση *DateTime* **κληρονομεί** (inherits) –για κάθε αντικείμενό της– όλα τα χαρακτηριστικά και τη συμπεριφορά που έχει ένα αντικείμενο της *Date*.
- Αλλά, κάθε αντικείμενό της *DateTime* έχει επιπλέον χαρακτηριστικά και μπορούμε να «εμπλουτίσουμε» τη συμπεριφορά του με επιπλέον μεθόδους. Αυτό και θα κάνουμε στη συνέχεια.

Αφού η *DateTime* κληρονομεί τη συμπεριφορά της *Date* θα κληρονομεί και την αναλλοίωτη. Πράγματι, ας δούμε την αναλλοίωτη της νέας κλάσης:

$$I_{DateTime}: (dYear > 0)\ \&\&\ (0 < dMonth <= 12)\ \&\&\ (0 < dDay <= lastDay(dYear, dMonth))$$
$$\&\&\ (0 < dtHour < 24)\ \&\&\ (0 < dtMin < 60)\ \&\&\ (0 < dtSec < 60)$$

Ή

$$I_{DateTime}: I_{Date}\ \&\&\ (0 < dtHour < 24)\ \&\&\ (0 < dtMin < 60)\ \&\&\ (0 < dtSec < 60)$$

Δηλαδή, αφού η I_{Date} συνδέεται με τα υπόλοιπα με "**&&**":

- ***Ό,τι ισχύει για τα αντικείμενα της βασικής κλάσης ισχύει και για τα αντικείμενα της παράγωγης κλάσης.***

Αυτή είναι η **αρχή της Liskov** (Liskov & Wing, 1993).

«*Βασική*»; «*Παράγωγη*»; Ήλθε η ώρα για λίγη ορολογία: Λέμε ότι η κλάση *DateTime* **παράγεται** (is derived) από την *Date* και παίρνει ως **κληρονομιά** (inheritance) όλα τα μέλη και τις μεθόδους της. Λέμε ότι η (κληρονομούμενη) κλάση *Date* είναι η

- **βασική κλάση** (base class) ή
- **υπερκλάση** (superclass) ή
- **γονική** (parent) κλάση

ενώ η κληρονόμος *DateTime* ονομάζεται

- **παράγωγη** (derived) κλάση ή
- **υποκλάση** (subclass) ή
- **κλάση-παιδί** (child class).

23.1.1 Τι ΔΕΝ Κληρονομείται

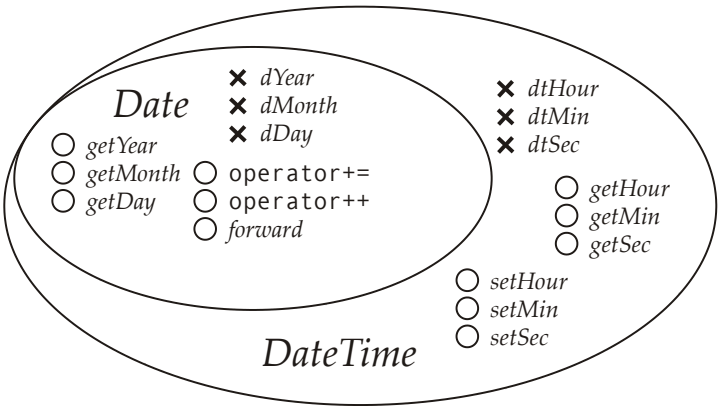
Γενικώς:

- ***Οι δημιουργοί δεν κληρονομούνται.***

με την έννοια που είδαμε στο παράδειγμα. Πέρα από αυτό όμως, υπάρχουν τέσσερις «ειδικές συναρτήσεις»:

- ο ερήμην δημιουργός,
- ο δημιουργός αντιγραφής,
- ο τελεστής εκχώρησης,
- ο καταστροφέας,

που όχι μόνο δεν κληρονομούνται με την έννοια που είδαμε πιο πάνω, αλλά αν δεν ορίσουμε κάποια από αυτές, όταν ορίζουμε μια κλάση, ο μεταγλωττιστής θα ορίσει αυτομάτως μια *συναγόμενη* (implicit). Αυτό, όπως ξέρουμε, δεν σημαίνει ότι η συναγόμενη συνάρτηση



Σχ. 23-1 Κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

θα συμπεριφέρεται όπως τη θέλουμε. Για τον λόγο αυτόν ακριβώς έχουμε βάλει κανόνες για το πότε πρέπει να παρεμβαίνουμε.

Τα παραπάνω ισχύουν και για τις παράγωγες κλάσεις: *Αν δεν ορίσουμε οποιαδήποτε από τις παραπάνω συναρτήσεις ο μεταγλωττιστής θα ορίσει αυτομάτως μια συναγόμενη.*

Αν όμως θελήσουμε να ορίσουμε δικές μας συναρτήσεις, μπορούμε να χρησιμοποιήσουμε τις αντίστοιχες της βασικής κλάσης όπως θα δούμε στη συνέχεια.

Τέλος, να πούμε ότι:

- ***Οι φιλίες δεν κληρονομούνται.***

Δηλαδή, το ότι μια συνάρτηση ή μια κλάση είναι φίλη της βασικής κλάσης δεν σημαίνει ότι αυτομάτως είναι και φίλη της παράγωγης. Αν θέλεις να είναι φίλη της παράγωγης θα πρέπει να το δηλώσεις.

23.2 Σχέσεις Αντικειμένων και Κλάσεων

Δες το Σχ. 23-1: να πώς περίπου είναι ένα αντικείμενο κλάσης *DateTime*. Όπως βλέπεις περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

Πώς ερμηνεύονται λοιπόν αυτά που είδαμε στο προηγούμενο παράδειγμα; Δηλαδή: γιατί ένα αντικείμενο κλάσης *DateTime* συμπεριφέρεται σαν αντικείμενο κλάσης *Date*; Διότι κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date* που ανταποκρίθηκε με τον τρόπο που είδαμε σε όλα τα μηνύματα που του στείλαμε.

Γενικεύοντας δίνουμε δύο κανόνες:

- ***Κάθε αντικείμενο της παράγωγης κλάσης D περιέχει ένα υποαντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

  Λόγω αυτού του γεγονότος:

- ***Κάθε αντικείμενο της παράγωγης κλάσης D συμπεριφέρεται σαν αντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

Και ακόμα γενικότερα:

- ***Οπουδήποτε, μέσα στο πρόγραμμά μας μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D.***

Αυτό λέγεται **δυνατότητα υποκατάστασης** (substitutability) και είναι μια άλλη θεώρηση της αρχής της Liskov (Liskov Substitution Principle - LSP).[1]

[1] Στο τέλος του κεφαλαίου θα δεις μια ακριβέστερη διατύπωση.

Για παράδειγμα, αν έχουμε δηλώσει:

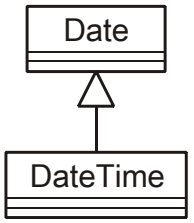
```
Date d;
DateTime dt;
Date* pD;
DateTime* pDt;
```

είναι νόμιμες οι εντολές:

```
d = dt;       // στη d εκχωρείται
              // το υποαντικείμενο κλάσης Date της dt
pD = new DateTime;
```

αλλά δεν είναι νόμιμες οι:[2]

```
dt = d;
pDt = new Date;
```



Σχ. 23-2 Παράσταση της συσχέτισης *DateTime* `is_a` *Date*.

Το ότι με τη "`d = dt`" εκχωρείται στη *d* το υποαντικείμενο κλάσης *Date* της *dt* ονομάζεται **τεμαχισμός** (slicing) του αντικειμένου. Αυτό θα το ξαναδούμε...

Πρόσεξε όμως και το εξής: Αν δούμε τις κλάσεις ως σύνολα αντικειμένων τότε *η βασική κλάση B περιέχει και τα αντικείμενα της παράγωγης κλάσης D*, δηλαδή η *D* είναι υποσύνολο της *B* (ενώ το αντικείμενο της *B* είναι «υποσύνολο» του αντικειμένου της *D*). Αντί για $D \subseteq B$ γράφουμε *D* `is_a` *B* και δίνουμε μια διαγραμματική παράσταση σαν αυτήν του Σχ. 23-2. Το βέλος δείχνει από την παράγωγη προς τη βασική. Το νόημά του είναι «*η D **αναφέρεται προς** (refers to) την B*».

Σημείωση:▶
Ορισμένοι αντί για *D* `is_a` *B* γράφουν *D* `a_kind_of` *B* και κρατούν την `is_a` ως *σχέση μεταξύ των αντικειμένων* των δύο κλάσεων. Εμείς χρησιμοποιούμε το `is_a` και για τις δύο περιπτώσεις.◀

Στη συνέχεια θα δούμε πώς μπορούμε να χειριζόμαστε το υποαντικείμενο της βασικής κλάσης μέσα στις μεθόδους της παράγωγης. Προς το παρόν να πούμε ότι μπορείς να φτάσεις σε αυτό ως εξής:

- Το `this` είναι βέλος προς αντικείμενο την παράγωγης κλάσης, στο παράδειγμά μας τύπου *DateTime*`*`.
- Η παράσταση "`static_cast<Date*>(this)`" μας δίνει ένα αντίγραφο του `this` αλλά τύπου *Date*`*`. Αυτό *είναι βέλος προς το υποαντικείμενο κλάσης Date*`*`.
- Το "`*(static_cast<Date*>(this))`" είναι *το υποαντικείμενο κλάσης Date*.

Να και ένα σχετικό

Παράδειγμα
Η "`dt = d`" απαγορεύεται με την έννοια ότι δεν υπάρχει παρόμοια *συναγόμενη* επιφόρτωση του "`=`". Φυσικά, αν την ορίσουμε δεν υπάρχει πρόβλημα.

Αν ορίσουμε, για παράδειγμα τον *δημιουργό μετατροπής*:

```
DateTime& DateTime::operator=( const Date& rhs )
{
   *(static_cast<Date*>(this)) = rhs:
   dtHour = 12;  dtMin = 0;  dtSec = 0;
} // DateTime::operator=
```

–που είναι κάτι λογικό– μια τέτοια εκχώρηση γίνεται νόμιμη.
&

Έξω από το αντικείμενο της παράγωγης κλάσης. μπορείς να πάρεις ένα αντίγραφο του υποαντικειμένου της βασικής με στατική τυποθεώρηση. Για παράδειγμα, το

"`static_cast<Date>(dt)`"

θα σου δώσει ένα αντίγραφο του υποαντικειμένου κλάσης *Date* του *dt*.

Η "`d = dt`", που είδαμε, εκτελείται ως "`d = static_cast<Date>(dt)`".

---

[2] Αργότερα θα δούμε πώς μπορούμε να κάνουμε νόμιμες και αυτές τις εντολές.

23.3 Ο Νέος Δημιουργός

Ας πούμε ότι, λόγω εκτέλεσης μιας δήλωσης, δημιουργείται αντικείμενο που αποτελείται από άλλα αντικείμενα είτε λόγω κληρονομιάς είτε λόγω δήλωσης μελών. Η σειρά δημιουργίας αντικειμένου που είδαμε στην §21.3 χρειάζεται συμπλήρωση[3]:

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται.
- Εκτελείται το σώμα του δημιουργού.

Παράδειγμα

Στο πρόγραμμα:

```
 0:   #include <iostream>
 1:   using namespace std;
 2:
 3:   class A
 4:   {
 5:      int ma;
 6:   public:
 7:      A( int ap = 0 )
 8:      {  ma = ap;  cout << "A object created" << endl;  }
 9:   }; // A
10:
11:   class B
12:   {
13:      int mb;
14:   public:
15:      B( int bp = 0 )
16:      {  mb = bp;  cout << "B object created" << endl;  }
17:   }; // B
18:
19:   class C
20:   {
21:      int mc;
22:   public:
23:      C( int cp = 0 )
24:      {  mc = cp;  cout << "C object created" << endl;  }
25:   }; // C
26:
27:   class D: public B
28:   {
29:      A da;
30:      C dc;
31:   public:
32:      D() {  cout << "D object created" << endl;  }
33:   }; // D
34:
35:   int main()
36:   {
37:      D d;
38:   }
```

βλέπουμε ότι η κλάση *D* (γρ. 27-33) είναι παράγωγη της *B* (γρ. 11-17) και έχει δύο μέλη, το πρώτο (γρ. 29) κλάσης *A* (γρ. 3-9), το δεύτερο (γρ. 30) κλάσης *C* (γρ. 19-25). Με βάση αυτά που είπαμε, για να δημιουργηθεί το αντικείμενο *d* (γρ. 37 στη `main`)

- Πρώτα θα κληθεί ο δημιουργός της *B*, για να δημιουργήσει το υποαντικείμενο κλάσης *B* που θα υπάρχει στο *d*.
- Μετά θα δημιουργηθούν τα μέλη *da* και *dc* από τους δημιουργούς των κλάσεων *A* και *C*.
- Τέλος, θα εκτελεσθεί το σώμα του δημιουργού της *D*.

  Αυτά επιβεβαιώνονται από τα αποτελέσματα:

[3] Ούτε τώρα είναι πλήρης ο κανόνας. Θα επανέλθουμε...

```
B object created
A object created
C object created
D object created
```

Να γράψουμε λοιπόν για την *DateTime* τον δημιουργό:

```
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
```

Σύμφωνα με τα παραπάνω, όταν θα εκτελείται, αρχικώς θα δημιουργηθεί το υποαντικείμενο της βασικής κλάσης, της *Date*, μετά θα δημιουργηθούν τα μέλη *dtHour*, *dtMin*, *dtSec* και τέλος θα εκτελεσθεί το σώμα όπου θα δώσουμε τιμές στα μέλη (αφού χρειάζονται και έλεγχοι):

```
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Καλά όλα αυτά, αλλά στα μέλη του υποαντικειμένου πώς θα δώσουμε τιμές; Αν προσπαθήσουμε να γράψουμε

```
   dYear = yp;  dMonth = mp;  dDay = dp;
```

μας περιμένει μια δυσάρεστη έκπληξη[4]:

```
Error E2247 dateTime01.cpp 128: 'Date::dYear' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dMonth' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dDay' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
```

Δηλαδή: ο δημιουργός της *DateTime* δεν έχει πρόσβαση στα *dYear*, *dMonth*, *dDay* του υποαντικειμένου *Date*! Για όλα φταίει εκείνο το "`private:`" που έχουμε στη *Date* και στη συνέχεια θα δούμε πώς διορθώνεται.

Εδώ θα δώσουμε άλλη λύση στο πρόβλημά μας, με τη λίστα εκκίνησης:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Τι λέμε εδώ; Δημιούργησε το υποαντικείμενο κλάσης *Date* με παραμέτρους *yp*, *mp*, *dp* και μετά συμπλήρωσε τις τιμές των «νέων» μελών. Με αυτόν τον τρόπο:

[4] Borland C++ v.5.5

- Δεν χρειάζεται να ξαναβάλουμε τους ελέγχους αφού θα τους κάνει ο δημιουργός της *Date*.
- Τα μέλη του υποαντικειμένου *Date* παίρνουν τιμή με τη δημιουργία του. Δεν χρειάζεται να πάρουν τις ερήμην τιμές και μετά να βάλουμε αυτές που πιθανόν έχουν δοθεί.

Βλέπουμε λοιπόν ότι *στη λίστα εκκίνησης μιας παράγωγης κλάσης μπορούμε να καλούμε τον δημιουργό της βασικής για να δημιουργήσει το αντίστοιχο υποαντικείμενο.*

23.3.1 Και Ένας Άλλος Τρόπος

Με βάση αυτά που είπαμε για εκχώρηση τιμής στο υποαντικείμενο της βασικής κλάσης μπορούμε να γράψουμε:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Φυσικά, δεν εννοούμε να προτιμήσεις αυτόν τον τρόπο. Ο πρώτος είναι σαφώς καλύτερος.

23.3.2 Ο Δημιουργός Αντιγραφής

Κατ' αρχάς να ξεκαθαρίσουμε ότι ο *εννοούμενος δημιουργός αντιγραφής της DateTime είναι σωστός*.

Θα γράψουμε όμως εδώ έναν τέτοιο δημιουργό μόνο και μόνο για παράδειγμα. Αλλά, πριν από αυτό θα δώσουμε ένα παράδειγμα της *δυνατότητας υποκατάστασης*. Αν έχουμε ορίσει:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
```

τότε οι

```
   Date d( dt );
   cout << d << endl;
```

θα δώσουν:

```
16.11.2007
```

Δηλαδή: Ένα αντικείμενο της παράγωγης κλάσης μπορεί να χρησιμοποιηθεί ως αρχική τιμή για ένα αντικείμενο της βασικής. Φυσικά, έχουμε τεμαχισμό και στο *d* εκχωρείται η τιμή του υποαντικειμένου κλάσης *Date* του *dt*.

Με βάση αυτό, να πώς μπορούμε να γράψουμε τον δημιουργό αντιγραφής της *DateTime*:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs )
{
   dtHour = rhs.dtHour;  dtMin = rhs.dtMin;  dtSec = rhs.dtSec;
}; // DateTime::DateTime
```

Με το "`Date(rhs)`" καλούμε τον δημιουργό αντιγραφής της *Date* για να δημιουργήσει αντίγραφο του υποαντικειμένου *Date* του *rhs*. Το σώμα του δημιουργού αναλαμβάνει την αντιγραφή των άλλων μελών.[5]

Από τον δημιουργό αντιγραφής παίρνουμε τον τελεστή εκχώρησης όπως μάθαμε στην §21.7, αφού προηγουμένως ορίσουμε την «ασφαλή *swap*». Το βλέπουμε στη συνέχεια.

23.4 Και ο Καταστροφέας ...

Η καταστροφή του αντικειμένου της παράγωγης κλάσης γίνεται με την αντίστροφη σειρά από αυτήν της δημιουργίας:

- Εκτελείται το σώμα του καταστροφέα.
- Καταστρέφονται τα μέλη του αντικειμένου.
- Καλείται (αυτομάτως) ο καταστροφέας της βασικής κλάσης και καταστρέφει το υποαντικείμενο της βασικής κλάσης.

Για παράδειγμα, αν στο πρόγραμμα της §23.3 εφοδιάσεις τις κλάσεις σου με κατάστροφείς του είδους:

```
   ~A() {  cout << "destroying A object" << endl;  }
   ~B() {  cout << "destroying B object" << endl;  }
 . . .
```

θα δεις στο τέλος της εκτέλεσης:

```
destroying D object    (εκτελείται το σώμα του καταστροφέα)
destroying C object    (καταστρέφονται τα μέλη του αντικειμένου)
destroying A object
destroying B object    (καταστρέφεται το υποαντικείμενο της βασικής)
```

Με τον καταστροφέα όμως δεν τελειώσαμε· θα επανέλθουμε...

23.5 Νέες και Παλιές Μέθοδοι

Να δούμε τι άλλο χρειαζόμαστε για τη νέα κλάση εκτός από τον δημιουργό και αυτά που κληρονομούνται από βασική.

Σίγουρα θα χρειαστούμε μεθόδους "*get*" και "*set*" για τα τρία μέλη:

```
   unsigned int getHour() const {  return dtHour;  };
   unsigned int getMin() const {  return dtMin;  };
   unsigned int getSec() const {  return dtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
```

όπου:

```
void DateTime::setHour( int hp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "setHour",
                          DateTimeXptn::hourRange, hp );
   dtHour = hp;
} // DateTime::setHour

void DateTime::setMin( int minp )
{
   if ( minp < 0 || 59 < minp )
```

[5] Αυτά γενικώς. Στην συγκεκριμένη περίπτωση θα ήταν καλύτερο να γράψουμε:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs ), dtHour( rhs.dtHour ), dtMin( rhs.dtMin ), dtSec( rhs.dtSec ) { }
```

```
      throw DateTimeXptn( "setMin",
                          DateTimeXptn::minRange, minp );
   dtMin = minp;
} // DateTime::setMin

void DateTime::setSec( int sp )
{
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "setSec",
                          DateTimeXptn::minRange, sp );
   dtSec = sp;
} // DateTime::setSec
```

Μια μέθοδος που μας ενδιαφέρει είναι η *forward()* –ή ο τελεστής "+="– αλλά τη θέλουμε *να αυξάνει τα δευτερόλεπτα και όχι τις ημέρες*. Γίνεται; Γίνεται:[6]

```
DateTime& DateTime::forward( long int ds )
{
   tm     currD = { dtSec, dtMin, dtHour, getDay(),
                    getMonth()-1, getYear()-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   if ( ds < 0 )
   {
      if ( currT < (-ds) )  // currT + ds < 0
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   else // ds >= 0
   {
      if ( ds > LONG_MAX - currT )  // currT + ds > ULONG_MAX
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   currT += ds;
   currD = *localtime( &currT );
   *(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                       currD.tm_mon+1,
                                       currD.tm_mday );
   dtHour = currD.tm_hour;
   dtMin = currD.tm_min;
   dtSec = currD.tm_sec;
   return *this;
} // Date::forward
```

Πρόσεξε δύο σημεία:

- Χρησιμοποιούμε και εδώ τον "=" της βασικής κλάσης (*Date*).
- Πρόσεξε τις επικεφαλίδες της μεθόδου στις δύο κλάσεις:

```
Date& forward( long int dd )
DateTime& forward( long int ds )
```

Διαφέρουν μόνο στον τύπο της επιστρεφόμενης τιμής. Δες όμως το αποτέλεσμα αυτής της ομοιότητας:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
   Date d( dt );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
   d.forward( 10 );
   dt.forward( 10 );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
```

[6] Αν δεν καταλαβαίνεις τις λεπτομέρειες της υλοποίησης ξαναδιάβασε την §22.5.1.4.

```
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
```

Αποτέλεσμα:

```
16.11.2007
16.11.2007 19:30:31
26.11.2007
16.11.2007 19:30:41
```

Αν ορίσουμε:

```
DateTime& operator+=( long int ds ) {  return forward( ds );  }
```

και αλλάξουμε τίς κλήσεις προς τις *forward()* σε:

```
d += 10;
dt += 10;
```

θα πάρουμε τα ίδια αποτελέσματα.

Και στις δύο περιπτώσεις, δράσαμε με την ίδια μέθοδο (τον ίδιο τελεστή) και το ίδιο όρισμα (10) και στα δύο αντικείμενα –*d* κλάσης *Date* και *dt* κλάσης *DateTime*– και η πρώτη προχώρησε κατά 10 ημέρες ενώ η δεύτερη προχώρησε κατά 10 *sec*. Τέτοιες μέθοδοι (τελεστές) λέγονται *πολυμορφικές*. Ο πολυμορφισμός από κληρονομιές βασίζεται στο ότι, για τα αντικείμενα της παράγωγης κλάσης η μέθοδος της παράγωγης κλάσης **υπερισχύει** (overrides) της μεθόδου της βασικής.

- ***Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.***

Δηλαδή, δεν υπάρχει περίπτωση να χρησιμοποιήσει ένα αντικείμενο της *DateTime* την *forward()* της *Date*; Ναι, αν ζητηθεί πιο συγκεκριμένα. Στο παραπάνω παράδειγμα δίνουμε:

```
dt.Date::operator+=( 10 );
```

ή

```
dt.Date::forward( 10 );
```

και η τιμή του *dt* γίνεται:

```
26.11.2007 19:30:41
```

Με τον πολυμορφισμό δεν τελειώσαμε. Στη συνέχεια θα δούμε μερικές πολύ ενδιαφέρουσες πτυχές του θέματος.

Παρατήρηση:▶

Με το "`this->Date::`" μπορούμε να καλούμε τις μεθόδους του υποαντικειμένου κλάσης *Date* και μέσα στις μεθόδους της παράγωγης κλάσης (*DateTime*). Για παράδειγμα αντί για

```
*(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                    currD.tm_mon+1,
                                    currD.tm_mday );
```

θα μπορούσαμε να γράψουμε:

```
this->Date::operator=( Date(currD.tm_year+1900, currD.tm_mon+1,
                            currD.tm_mday) );
```

Με αυτόν τον τρόπο δεν δημιουργείται αντίγραφο του `this` και έχεις ένα (πολύ μικρό) κέρδος.◀

Ας δούμε τώρα ένα άλλο πρόβλημα: Αν κάποιος γράψει `++dt` τότε η τιμή της *dt* θα προχωρήσει κατά μια ημέρα. Αυτό μπορεί να δημιουργήσει σύγχυση. Να την ξαναορίσουμε ώστε να προχωράει κατά 1 *sec*; Ας πούμε ότι αυτό δεν μας ενδιαφέρει· αν θέλουμε να προχωρήσουμε 1 *sec* θα χρησιμοποιούμε τη *forward()*. Τι κάνουμε;

Ορίζουμε στην περιοχή `private` της *DateTime*:

```
DateTime& operator++() { }
```

Αν τυχόν σου ξεφύγει κάποιο `++dt` μέσα στο πρόγραμμά σου θα το φροντίσει ο μεταγλωττιστής: «`'DateTime& operator++()' is private`.»

Φυσικά, αν κάποτε θελήσεις να χρησιμοποιήσεις τον "`++`" της *Date* μπορείς να δώσεις:

```
dt.Date::operator++();
```

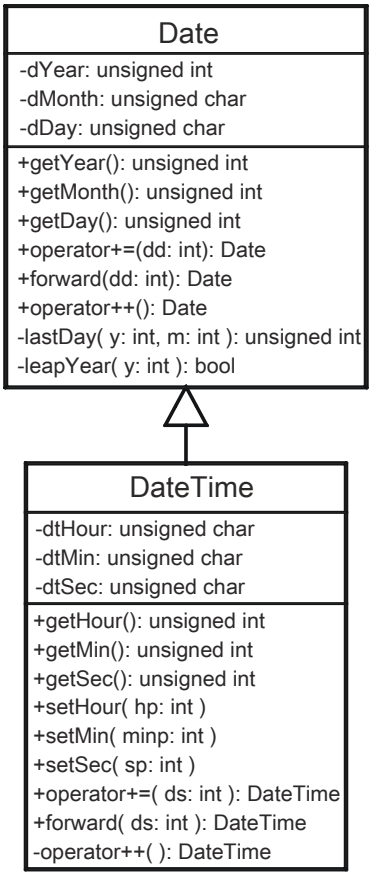
Παρατήρηση: ▶

Αν εσύ επιμένεις να επιφορτώσεις τον "`++`" ώστε η "`++dt`" να αυξάνει τον χρόνο κατά 1 *sec* θα πρέπει, σύμφωνα με αυτά που είπαμε στην §22.2, να την ορίσεις **inline** με βάση τη *forward*:

```
DateTime& operator++()
{  return forward( 1 );  }
```

Αυτός ο ορισμός διαφέρει από τον αντίστοιχο της *Date* (§22.5.1.6) μόνον στον τύπο του αποτελέσματος.◀

Στο Σχ. 23-3 βλέπεις ένα λεπτομερέστερο διάγραμμα κληρονομιάς.



Σχ. 23-3 Η κλάση *DateTime* είναι παράγωγη της κλάσης *Date*. Κληρονομεί από αυτήν τα *dYear*, *dMonth*, *dDay*, *getYear*, *getMonth*, *getDay*. Δεν κληρονομεί τις `operator+=`, *forward*, `operator++` (έχει δικές της).

23.5.1 Ο Τελεστής Εκχώρησης

Αν χρειάζεται να ορίσεις τον τελεστή εκχώρησης για την παράγωγη κλάση όρισέ τον με βάση τον δημιουργό αντιγραφής που, σύμφωνα με τον «κανόνα των τριών», μάλλον θα πρέπει να έχεις ορίσει. Χρησιμοποιώντας το πάγιο «πατρόν», δεν έχεις παρά να ορίσεις τη *swap()*.

Ορίζουμε τη *swap()* χρησιμοποιώντας τη *swap()* της βασικής. Στο παράδειγμά μας:

```
void DateTime::swap( DateTime& rhs )
{
   this->Date::swap( rhs );
   std::swap( dtHour, rhs.dtHour );
   std::swap( dtMin, rhs.dtMin );
   std::swap( dtSec, rhs.dtSec );
} // DateTime::swap
```

Η "`this->Date::swap(rhs)`" (ή, επί το απλούστερο: "`Date::swap(rhs)`") ασχολείται μόνο με το υποαντικείμενο *Date* του *rhs*.[7]

Μετά ο δρόμος είναι γνωστός:

```
DateTime& DateTime::operator=( const DateTime& rhs )
{
   if ( &rhs != this )
   {
      DateTime tmp( rhs );
      swap( tmp );
   }
   return *this;
} // DateTime::operator=
```

[7] Μα δεν έχουμε ορίσει *swap()* για τη *Date*! Ας την ορίσουμε:

```
void Date::swap( Date& rhs )
{
   std::swap( dYear, rhs.dYear );
   std::swap( dMonth, rhs.dMonth );
   std::swap( dDay, rhs.dDay );
} // Date::swap
```

23.6 Καθολικές Συναρτήσεις

Όπως στις μεθόδους, έτσι και στην υλοποίηση καθολικών συναρτήσεων για την παράγωγη κλάση μπορούμε να χρησιμοποιήσουμε συναρτήσεις που γράψαμε για τη βασική κλάση. Ας δούμε δύο παραδείγματα.

Επιφορτώνουμε τον "==" για τη *DateTime* ως εξής:

```
bool operator==( const DateTime& lhs, const DateTime& rhs )
{
   return ( static_cast<Date>(lhs) == static_cast<Date>(rhs) &&
            lhs.getHour() == rhs.getHour() &&
            lhs.getMin() == rhs.getMin() &&
            lhs.getSec() == rhs.getSec() );
} // operator==( const DateTime
```

και τον "<<":

```
ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return tout << static_cast<Date>( rhs ) << ' '
               << rhs.getHour() << ':' << rhs.getMin() << ':'
               << rhs.getSec();
} // operator<<( ostream& tout, const DateTime
```

23.7 Κλάση Εξαιρέσεων Παράγωγης Κλάσης

Θα τηρούμε τον εξής κανόνα:

- ***Η κλάση εξαιρέσεων της παράγωγης κλάσης είναι παράγωγη της κλάσης εξαιρέσεων της βασικής κλάσης.***

Θα έχουμε, για παράδειγμα:

```
struct DateTimeXptn :  public DateXptn
{
   enum { hourRange = 50, minRange, secRange  };
   DateTime errDateTimeVal;
   DateTimeXptn( const char* mn, int ec,
                 int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : DateXptn( mn, ec, ev1, ev2, ev3 ) {  }
   DateTimeXptn( const char* mn, int ec, int ev1, const DateTime& edt )
      : DateXptn( mn, ec, ev1, edt )
   {  errDateTimeVal = edt;  }
}; // DateTimeXptn
```

Πρόσεξε:

- Το "`hourRange = 50`": με αυτόν τον τρόπο επιδιώκουμε να διαχωρίσουμε τις περιοχές κωδικών σφάλματος των *DateXptn* (0 .. *DateXptn*::*cannotWrite*) και *DateTimeXptn* (50.. *DateTimeXptn*::*secRange*).
- Τη σύλληψη των εξαιρέσεων· αν κάπου πιάνεις εξαιρέσεις και των δύο τύπων θα πρέπει να βάλεις:

  ```
  catch( DateTimeXptn& x )
  { . . . }
  catch( DateXptn& x )
  { . . . }
  ```

  Αν βάλεις τις **catch** με την αντίστροφη σειρά η "`catch(DateXptn& x)`" δεν θα αφήνει να περάσει οποιαδήποτε εξαίρεση στην "`catch(DateTimeXptn& x)`", αφού κάθε `DateTimeXptn` είναι και `DateXptn`.

Και μια παρατήρηση για το συγκεκριμένο παράδειγμα: Ένα αντικείμενο κλάσης *DateTimeXptn* έχει μέλος κλάσης *DateTime* αλλά έχει κληρονομήσει και μέλος κλάσης *Date* (*errDateVal*). Στο κληρονομημένο μέλος δίνουμε τιμή με τη λίστα εκκίνησης του δεύτερου

δημιουργού, από το υποαντικείμενο κλάσης *Date* του *edt*, με τον σχετικό τεμαχισμό. Ολόκληρο το *edt* αντιγράφεται στο νέο μέλος *errDateTimeVal*.

23.8 "private" ή "protected"

Λέγαμε στην §23.3 ότι «*ο δημιουργός της DateTime δεν έχει πρόσβαση στα dYear, dMonth, dDay του υποαντικειμένου Date! Για όλα φταίει εκείνο το "*`private`*:" που έχουμε στη Date*». Αυτό βέβαια ισχύει και για οποιαδήποτε μέθοδο της *DateTime*.

Παρ' ολα αυτά, όπως φάνηκε από αυτά που είδαμε στις προηγούμενες παραγράφους:

- Μπορούμε να αλλάζουμε τις τιμές των μελών του υποαντικειμένου της βασικής με κάποιον δημιουργό ή άλλες μεθόδους (π.χ. *swap()*) και –αν χρειαστεί– με κάποια "*set*".
- Μπορούμε να παίρνουμε τις τιμές των μελών του υποαντικειμένου της βασικής με τις "*get*".

Σε πολλές περιπτώσεις πάντως, όταν έχουμε πολύπλοκους υπολογισμούς, τα παραπάνω δεν είναι ικανοποιητικά. Εκτός από αυτό, συχνά στις περιοχές `private` υπάρχουν «κρυφές» μέθοδοι ή βοηθητικές συναρτήσεις που μας είναι χρήσιμες για την υλοποίηση των μεθόδων της παράγωγης κλάσης.

Η C++ μας δίνει λύση στα προβλήματα αυτά με μια τρίτη κατηγορία δηλώσεων, τις δηλώσεις "`protected`".

- ***Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της.***

Σημείωση: ▶

Δες μια άλλη κάπως ακριβέστερη διατύπωση: *Ό,τι υπάρχει στην περιοχή "*`protected`*" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης*. Διάβασε την επόμενη υποπαράγραφο.◀

Ας δούμε ένα παράδειγμα· αλλάζουμε τη *Date* ως εξής:

```
class Date
{
// . . .
public:
// . . .
protected:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   static unsigned int lastDay( int y, int m );
private:
   static bool leapYear( int y );
}; // Date
```

Τώρα στον δημιουργό της *DateTime* μπορείς να γράψεις, αφού βάλεις τους ελέγχους που ξέρουμε:

```
   if ( yp <= 0 ) throw . . .
   dYear = yp;
   if ( mp <= 0 || 12 < mp ) throw . . .
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
```

αντί για

```
   :  Date( yp, mp, dp )
```

ή

```
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
```

Αλλά προσοχή: αυτό δεν σημαίνει ότι πρέπει να γράψεις έτσι τον δημιουργό. Όπως βλέπεις, ξαναγράφουμε και εδώ τους ελέγχους που έχουμε γράψει στον δημιουργό και στις "*set*" της *Date*. Οι δύο τρόποι, που είδαμε πιο πριν είναι προτιμότεροι διότι βασίζονται στον δημιουργό της *Date* που έχει τους ελέγχους.

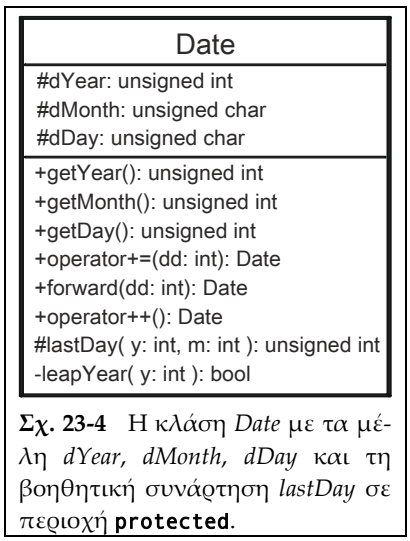
Αν πάμε όμως στη *forword()* τα πράγματα αλλάζουν: οι

```
dYear = currD.tm_year+1900;
dMonth = currD.tm_mon+1;
dDay = currD.tm_mday;
```

είναι προτιμότερες από την

```
*(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                    currD.tm_mon+1,
                                    currD.tm_mday );
```

διότι ο δημιουργός που καλείται θα κάνει ελέγχους που είναι άχρηστοι.



Σχ. 23-4 Η κλάση *Date* με τα μέλη *dYear*, *dMonth*, *dDay* και τη βοηθητική συνάρτηση *lastDay* σε περιοχή `protected`.

Όπως λέγαμε και στην §20.8.1, στα διαγράμματα της UML, ό,τι δηλώνεται σε περιοχή "`protected`" σημειώνεται με "`#`". Στο Σχ. 23-4 βλέπεις διαγραμματική παράσταση της *Date*, όπως την αλλάξαμε παραπάνω.

Πολλοί πεπειραμένοι προγραμματιστές βάζουν πάντοτε τα μέλη σε περιοχή "`private`" ενώ σε περιοχή "`protected`" μπορεί να βάλουν βοηθητικές συναρτήσεις ή μεθόδους.

Άλλοι, χωρίς συζήτηση, αντικαθιστούν το "`private`" με το "`protected`" σε κάθε κλάση που πρόκειται να κληρονομηθεί. Αν θέλεις να κάνεις και εσύ το ίδιο πάρε υπόψη σου τα εξής:

- ***Αν χρησιμοποιείς την αλλαγή περιορισμού πρόσβασης από "`private`" σε "`protected`" –για να μπορείς να αλλάζεις τις τιμές των μελών του υποαντικειμένου της βασικής κλάσης– θα πρέπει να βάζεις και τους απαραίτητους ελέγχους κατά περίπτωση.***

Φυσικά, με αυτόν τον τρόπο, πληθαίνουν τα σημεία των ελέγχων πράγμα που κάνει πιο πολύπλοκη τη κάθε απόπειρα ενημέρωσης/τροποποίησης της βασικής κλάσης. Για τον λόγο αυτόν, ακόμη και με τη αλλαγή σε "`protected`", καλό είναι

- ***Να προσπαθείς να ελαχιστοποιείς την κατ' ευθείαν πρόσβαση στα μέλη του υποαντικειμένου γράφοντας τις μεθόδους της παράγωγης κλάσης με χρήση των μεθόδων και των δημιουργών της βασικής.***

23.8.1 * "`protected`": Ψιλά Γράμματα

Επανερχόμαστε τώρα στον κανόνα-ορισμό της προηγούμενης παραγράφου: «*Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της*» και στη διευκρινιστική σημείωση «*Ό,τι υπάρχει στην περιοχή "`protected`" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης.*» Ας πούμε ότι έχουμε:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   void display( ostream& tout ) const {  tout << mb;  }
protected:
   int mb;
}; // B
```

```
class D2: public B
{
public:
   D2( int bp=2, int d2p=0 ): B(bp)  {  md2 = d2p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md2;  }
private:
   int md2;
}; // D2
```

Θέλουμε να εφοδιάσουμε τη *D2* με μια μέθοδο:

```
   void xchangeB( B& b );
```

που θα ανταλλάσσει τις τιμές των μελών *mb* του αντικειμένου και του *b*. Αν

```
   B b;
   D2 d2a( 2, 11 );
```

τότε οι:

```
   d2a.xchangeB( b );
   d2a.display( cout );  cout << endl;
   b.display( cout );  cout << endl;
```

θα πρέπει να δώσουν:

```
0 11
2
```

Θέλουμε να υλοποιήσουμε τη μέθοδο ως εξής:

```
   void xchangeB( B& b )  {  std::swap( b.mb, mb );  }
```

αλλά ο μεταγλωττιστής[8] έχει αντιρρήσεις: «`Error E2247 testDerived.cpp 31: 'B::mb' is not accessible in function D2::xchangeB(B &)`».

Εδώ τι έχουμε;

- Στη βασική κλάση (*B*) έχουμε δηλώσει "`protected: int mb`".
- Στην παράγωγη κλάση (*D1*), μια μέθοδος προσπαθεί να διαχειρισθεί το μέλος *mb* ενός αντικειμένου *b* κλάσης *B*.

Ο μεταγλωττιστής μας λέει ότι το "`B::mb`" δεν είναι προσβάσιμο από τη συνάρτηση-μέλος "`D2::xchangeB(B &)`".

Αν βάλουμε:

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

ο μεταγλωττιστής δεν έχει πρόβλημα με τη μέθοδο. Αν έχουμε δηλώσει

```
   D2 d2a( 2, 11 ), d2b( 3, 13 );
```

οι

```
   d2a.xchangeB( d2b );
   d2a.display( cout );  cout << endl;
   d2b.display( cout );  cout << endl;
```

θα δώσουν:

```
3 11
2 13
```

Τώρα όμως δεν είναι δεκτή η "`d2a.xchangeB( b )`".

Αν ορίσουμε

```
class DD2: public D2
{
public:
   DD2( int bp=1, int d2p=0, int dd2p=0 )
      : D2(bp, d2p)  {  mdd2 = dd2p;  }
   void display( ostream& tout ) const
   {  D2::display( tout );  tout << ' ' << mdd2;  }
```

[8] Borland BC++, v.5.5.

```
private:
   int mdd2;
}; // DD2
```

και δηλώσουμε:

```
   DD2 dd2( 5, 17, 23 );
```

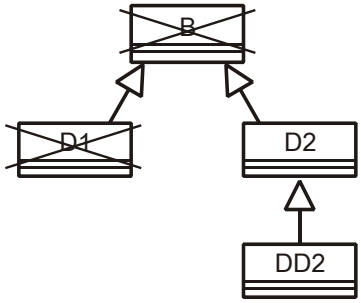
η

```
   d2a.xchangeB( dd2 );
```

είναι δεκτή και δουλεύει.

Αν όμως ορίσεις

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 )
      : B(bp)  {  md1 = d1p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
private:
   int md1;
}; // D1
```



Σχ. 23-5 Οι μέθοδοι της *D2* «βλέπουν» τα "`protected`" μέλη των υποαντικειμένων *B* των αντικειμενων της *D2* και της *DD2*· όχι της *B* και της *D1*.

Αλλάζουμε τώρα την *xchangeB()* σε

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

Και πάλι το ίδιο πρόβλημα: δεν επιτρέπεται πρόσβαση προς το "`b.mb`"!

Διατυπώνουμε λοιπόν τον κανόνα μας ως εξής (Σχ. 23-5):

- ***Αν έχουμε μια (βασική) κλάση B στην οποίαν υπάρχει περιοχή "`protected`", μια κλάση D, παράγωγη (αμέσως ή εμμέσως) της B και μια μέθοδο f της D, η f έχει πρόσβαση στα μέλη των περιοχών "`protected`" των υποαντικειμένων κλάσης B των αντικειμένων κλάσης D και των παραγώγων της D.***

23.9 Εικονικές Μέθοδοι

Στην §23.5 λέγανε: «*Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.*» Είναι όμως πάντα έτσι; Δες ένα πρόβλημα: Δηλώνουμε

```
   Date* pind[2];
```

και παίρνουμε μνήμη:

```
   pind[0] = new Date( 2002, 2, 2 );
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

Το δεύτερο είναι σωστό; Είναι και παραείναι! Αφού «*Οπουδήποτε … μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D*» όλα είναι εντάξει. Για δες όμως τη συνέχεια:

```
   pind[0]->forward( 10 );
   pind[1]->forward( 10 );
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
   ++(*pind[0]);
   ++(*pind[1]);
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
```

Αποτέλεσμα:

```
12
13
13
14
```

Δηλαδή; Και για το δεύτερο αντικείμενο χρησιμοποίησε τη *forward()* και τον "++" της βασικής. Η ζημιά έγινε από τον μεταγλωττιστή· όταν έκανε τη μεταγλώττιση σημείωσε ότι τα βέλη του pind δείχνουν αντικείμενα κλάσης *Date* και αυτό ήταν... Όταν έρχεται η ώρα του υπολογισμού χρησιμοποιεί τις μεθόδους της *Date*.

Τώρα σου έρχεται αγανάκτηση: «Ε, δεν θα χρησιμοποιούμε βέλη και τελειώσαμε!» Εύκολα το λες, δύσκολα το κάνεις. Ας πούμε ότι έχουμε:

```
   DateTime dt( 2003, 3, 3, 3, 33, 33 );
```

και τη συνάρτηση:

```
void plus10( Date& d )
{
   d.forward( 10 );
} // plus10
```

Οι εντολές

```
   cout << dt << endl;
   plus10( dt );
   cout << dt << endl;
```

θα δώσουν:

```
3.3.2003 3:33:33
13.3.2003 3:33:33
```

Δηλαδή, κλήθηκε η *forward()* της *Date* και όχι της *DateTime*. Γιατί; Αν το έχεις ξεχάσει, να υπενθυμίσουμε ότι

- οι παράμετροι αναφοράς είναι «μεταμφιεσμένα» βέλη και ότι
- όταν θέλουμε να περάσουμε σε συνάρτηση αντικείμενο θα χρησιμοποιήσουμε παράμετρο αναφοράς (με `const` ή χωρίς), διότι τα αντικείμενα είναι συνήθως μεγάλα.

Καταλαβαίνεις λοιπόν ότι δεν μπορείς να πεις τόσο εύκολα ότι δεν θα χρησιμοποιήσεις βέλη.

Πώς διορθώνεται αυτό; Ως εξής:

♦ ***Όποιες μεθόδους ξαναορίζουμε στην παράγωγη κλάση τις δηλώνουμε στη βασική κλάση ως "`virtual`" (εικονικές).***

```
class Date
{
// . . .
public:
   virtual const Date& operator+=( long int dd );
   virtual const Date& forward( long int dd );
   virtual Date& operator++();
// . . .
}; // Date
```

Μετά από αυτή τη διόρθωση οι εντολές του πρώτου παραδείγματος θα δώσουν:

```
12
3
13
3
```

ενώ από το δεύτερο θα πάρουμε:

```
3.3.2003 3:33:33
3.3.2003 3:33:43
```

Τι κάνει αυτή η «μαγική» λέξη;

- Ο χαρακτηρισμός "`virtual`" (εικονικός) μπαίνει σε μεθόδους της *βασικής* κλάσης όταν στην παράγωγη κλάση υπάρχει μέθοδος με το ίδιο όνομα (πολυμορφική). Στην περίπτωσή μας βάλαμε "`virtual`" τις `operator+=()`, *forward()*, `operator++()` της *Date* διότι αυτές ορίζονται ξανά και στην παράγωγή της *DateTime*.

- Το αποτέλεσμά του είναι το εξής: προειδοποιεί τον μεταγλωττιστή να μην βιαστεί να «δέσει» τη συγκεκριμένη μέθοδο με τα βέλη προς αντικείμενα της βασικής αλλά να αναβάλει την απόφασή του μέχρι τη στιγμή της εκτέλεσης της εντολής ώστε να μπορεί να «δει» *την κλάση του αντικειμένου που δείχνει το βέλος εκείνη τη στιγμή*.

Να βάλουμε "`virtual`" και στις αντίστοιχες μεθόδους της παράγωγης κλάσης; Πρόσεξε: Αν κατάλαβες τι ακριβώς συμβαίνει με το "`virtual`" θα έχεις καταλάβει ότι και αυτές είναι "`virtual`" είτε βάλουμε είτε δεν βάλουμε.[9] Καλύτερα λοιπόν να γράψουμε το "`virtual`" και στην παράγωγη κλάση για λόγους τεκμηρίωσης. Γιατί να μην δούμε την αλήθεια κατάματα;...

23.10 * Υλοποίηση Εικονικών Συναρτήσεων

Θα πούμε τώρα δυο λόγια για την υλοποίηση των εικονικών μεθόδων, παρ' όλο που αυτό είναι έξω από το αντικείμενο αυτού του βιβλίου.

Με τις κλάσεις του παραδείγματος της §23.3 (κάπως τροποποιημένες)

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual void display( ostream& tout ) const {  tout << mb;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B

class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
   virtual void g( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

κάνουμε το εξής πείραμα: Δηλώνουμε δύο αντικείμενα:

```
   B   bo;
   D1  d1o;
```

και ζητούμε:

```
   cout << "sizeof_bo: " << sizeof(bo) << endl;
   cout << "sizeof_d1o: " << sizeof(d1o) << endl;
```

Αποτέλεσμα:[10]

```
sizeof_bo: 8
sizeof_d1o: 12
```

Αλλά σε ένα αντικείμενο κλάσης *B* έχουμε ένα μέλος τύπου `int` ενώ τα αντικείμενα κλάσης *D1* έχουν δύο τέτοια μέλη. Ο μεταγλωττιστής που χρησιμοποιούμε παριστάνει τις τιμές τύπου `int` με 4 ψηφιολέξεις. Και στις δύο περιπτώσεις «περισσεύουν» 4 ψηφιολέξεις. Με αυτά υλοποιείται μια «κρυφή» μεταβλητή-βέλος, ας την πούμε *vfptr*, που δείχνει έναν πίνακα με τις εικονικές μεθόδους (για την ακρίβεια τις διευθύνσεις των εικονικών μεθό-

[9] «*Once virtual, always virtual!*»

[10] g++ (Dev-C++).

δων) που χρησιμοποιούν τα αντικείμενα της κλάσης και ονομάζεται **πίνακας εικονικών μεθόδων / συναρτήσεων** (virtual method / function table) ή απλώς **vtable**.

Κατά τη διάρκεια της μεταγλώττισης:

- Στον vtable της *B* θα εισαχθούν οι *B*::*f* και *B*::*display*.
- Στον vtable της *D1* θα εισαχθούν οι *B*::*f* (που κληρονομείται από τη *B*), *D1*::*display* και *D1*::*g*.
- Κάθε αντικείμενο κλάσης *B*, όπως το *bo*, εφοδιάζεται με το «κρυφό» βέλος *vfptr* και στους δημιουργούς της κλάσης μπαίνουν εντολές που το βάζουν να δείχνει τον vtable της *B*.
- Τα αντίστοιχα γίνονται και με τα αντικείμενα κλάσης *D1*.

Κατά τη διάρκεια της εκτέλεσης αν ζητηθεί η εκτέλεση κάποιας εικονικής μεθόδου ενός αντικειμένου αναζητείται η διεύθυνσή της στον πίνακα που δείχνει το *vfptr* και εκτελείται η σωστή μέθοδος.

23.10.1 "virtual" και Τεμαχισμός

Είναι δυνατόν να βάλουμε το "`virtual`" και να μην έχουμε τη σωστή συμπεριφορά του αντικειμένου; Και βέβαια, αν δεν το αφήνουμε να λειτουργήσει. Δες ένα

Παράδειγμα

Χρειαζόμαστε μια συνάρτηση, ας την πούμε *plus10Local()*, που θα τροφοδοτείται με ένα αντικείμενο *Date* ή *DateTime* –όπως η *plus10()*– και θα προχωρεί την τιμή του κατά 10 ημέρες ή *sec* αντιστοίχως αλλά μόνο για τις επεξεργασίες που γίνονται μέσα στη συνάρτηση· η αλλαγή δεν θα βγαίνει προς τη συνάρτηση που θα καλεί την *plus10Local()*.

Θα έλεγε κανείς –με αρκετή επιπολαιότητα– ότι εδώ μας δίνεται η ευκαιρία να «γλυτώσουμε» από τα βέλη και να γράψουμε:

```
void plus10Local( Date d )
{
   d.forward( 10 );
// . . .
} // plus10Local
```

Έτσι κάναμε ένα σοβαρό λάθος: όταν εκτελείται η "`plus10Local(dt)`" καλείται ο δημιουργός αντιγραφής της *Date* για να αντιγράψει το *dt* στο *d*. Το *dt*; Όχι! Το υποαντικείμενο κλάσης *Date* του *dt*. Έχουμε δηλαδή *τεμαχισμό* και το "`virtual`" δεν έχει ευκαιρία να δουλέψει...

Δυστυχώς, το λάθος αυτό γίνεται με τρόπο που είναι τελείως «νόμιμος» για τον μεταγλωττιστή και έτσι δεν θα μας δώσει κάποιο μήνυμα.

Καταλαβαίνεις τώρα ότι η συνήθεια να περνούμε τα αντικείμενα με παραμέτρους αναφοράς δεν έχει να κάνει μόνο με την αποφυγή μιας αντιγραφής.[11]

Παρατήρηση: ▶

Μια και συζητούμε για αντιγραφές, δες ένα πρόβλημα που προκύπτει: Έστω ότι βάζουμε παράμετρο "`const Date& d`". Τώρα η "`d.forward(10)`" είναι παράνομη λόγω του "`const`". Για να κάνουμε τη δουλειά μας θα πρέπει να αντιγράψουμε το *d* σε ένα αντικείμενο, τοπικό στην *plus10Local*. Αλλά, τι τύπου θα είναι αυτό το τοπικό αντικείμενο; Σε λίγο θα το μάθουμε και αυτό...◀

[11] Ο κανόνας OBJ33 του (CERT 2009) λέει: *Μην τεμαχίζεις πολυμορφικά αντικείμενα* (*Do not slice polymorphic objects*).

23.10.2 Κάποια Σχόλια...

- Ο όρος **virtual** αποδόθηκε στα ελληνικά ως «*εικονική*»· θα βρεις επίσης και το όρο «*νοητή*». Και οι δύο μεταφράσεις είναι εξ ίσου παραπλανητικές. Όπως είδαμε μέχρι τώρα, οι μέθοδοι που δηλώνονται ως "`virtual`" είναι πραγματικότατες. Μια (περιφραστική) μετάφραση «*δυναμικώς επιλεγόμενη*» θα ήταν πιο κοντά στην πραγματικότητα. Πάντως στη συνέχεια θα δούμε μεθόδους που είναι γνησίως εικονικές.
- Γιατί να μην είναι όλες οι μέθοδοι "**virtual**" να τελειώνουμε; Διότι ο μηχανισμός επιλογής της σωστής μεθόδου χρειάζεται χρόνο εκτέλεσης. Γιατί λοιπόν να φορτώνουμε το πρόγραμμά μας με άχρηστες δουλειές για μεθόδους που δεν πρόκειται να αλλάξουμε, π.χ. για τις "*get*" και "*set*";
- Το πρόβλημα που λύνεται με τις εικονικές μεθόδους υπάρχει και στις άλλες αντικειμενοστρεφείς γλώσσες και οι λύση είναι παρόμοια: «**καθυστερημένη**» **επιλογή** (**πρόσδεση**) (late ή deferred binding) μεθόδου. Οι «συγγενεις» γλώσσες –Java και C#– μπορεί να μην χρησιμοποιούν (φανερά) βέλη αλλά χρησιμοποιούν «κρυμμένα» (αναφορές). Έτσι
  - Στη Java κάθε μέθοδος είναι κατ' αρχήν "**virtual**" και ο προγραμματιστής θα πρέπει *να δηλώσει ποιες δεν θέλει* να είναι.
  - Η C# ακολουθεί τη C++: κάθε μέθοδος, κατ' αρχήν, *δεν είναι* "**virtual**" και ο προγραμματιστής θα πρέπει να δηλώσει ποιες θέλει να είναι.

23.11 Εικονικός Καταστροφέας

Επιστρέφουμε στο παράδειγμα της §23.3, όπως το εμπλουτίσαμε με τους καταστροφείς στην §23.4. Δίνουμε τις εντολές:

```
   B* pd( new D );
   delete pd;
```

και παίρνουμε:

```
B object created
A object created
C object created
D object created
destroying B object
```

Το ενδιαφέρον βρίσκεται στην τελευταία γραμμή, «`destroying B object`», που δείχνει ότι καλείται ο καταστροφέας της βασικής κλάσης για να καταστρέψει ένα αντικείμενο της παράγωγης κλάσης, το `*pd`. Πώς έγινε αυτό; Η απόφαση για το είδος του καταστροφέα πάρθηκε πολύ νωρίς, στη μεταγλωττιση.

Πώς διορθώνεται αυτό; Όπως είδαμε στην προηγούμενη παράγραφο:

♦ ***Ο καταστροφέας της βασικής κλάσης πρέπει να δηλώνεται "`virtual`".***

Πράγματι, αν ορίσουμε:

```
   virtual ~B() {  cout << "destroying B object" << endl;  }
```

θα πάρουμε:

```
B object created
A object created
C object created
D object created
destroying D object
destroying C object
destroying A object
destroying B object
```

Στην §21.5 λέγαμε «*Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα "`~K() {}`" στις κλάσεις που δεν χρειάζονται. Αργότερα θα δούμε ότι σε*

μερικές περιπτώσεις αυτό είναι απαραίτητο.» Τώρα βλέπεις ότι αυτό είναι απαραίτητο στις κλάσεις που πρόκειται να κληρονομηθούν· και μάλιστα θα πρέπει να ορίζεις:

```
   virtual ~K() { };
```

Παράδειγμα

Θα δούμε στη συνέχεια ότι η C++ έχει μια βασική κλάση εξαιρέσεων, την `std::exception`, από την οποία παράγονται όλες οι κλάσεις εξαιρέσεων που χρησιμοποιούν οι συναρτήσεις της βιβλιοθήκης της. Αυτή η κλάση είναι περίπου έτσι:

```
struct exception
{
   exception() throw() { }
   virtual ~exception() throw();
   virtual const char* what() const throw();
}; // exception
```

Όπως βλέπεις, ο καταστροφέας δεν ορίζεται αλλά έχει στη δήλωσή του το "`virtual`".

Έτσι, μπορούμε να βάλουμε στις δικές μας κλάσεις:

```
struct DateXptn :  public exception
{
// . . .
   virtual ~DateXptn() throw() { };
}; // DateXptn
```

και

```
struct DateTimeXptn :  public DateXptn
{
// . . .
   virtual ~DateTimeXptn() throw() { };
}; // DateTimeXptn
```

Αν θελήσεις να δοκιμάσεις τα παραπάνω θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <exception>`".

23.12 Περί Πολυμορφισμού

Αφού είδαμε διάφορες πλευρές –τεχνικές κυρίως– του πολυμορφισμού ας ολοκληρώσουμε –προς το παρόν– αυτό το θέμα, κυρίως για να ξεδιαλύνουμε μερικές παρεξηγήσεις της μορφής «πολυμορφισμός είναι επιφόρτωση ή υπερίσχυση ή μέθοδοι **virtual**».

Γενικώς, μπορούμε να ορίσουμε τον **πολυμορφισμό** (polymorphism) ως τη δυνατότητα να χειριζόμαστε (εν μέρει συνήθως) έναν τύπο *A* όπως τον τύπο *B*.

Στα πλαίσια του αντικειμενοστρεφούς προγραμματισμού αυτό εξειδικεύεται ως: η δυνατότητα αντικειμένων *διαφορετικών κλάσεων* να αποκρίνονται στο ίδιο μηνυμα με αποκρίσεις –όχι ταυτόσημες, αλλά– που εξαρτώνται από την κλάση του κάθε αντικειμένου.

Αυτά που μάθαμε μέχρι τώρα αφορούν την υλοποίηση στη C++ του λεγόμενου **πολυμορφισμού υποκλάσεων** (subclass ή subtype). Δηλαδή: αντικείμενα διαφορετικών τύπων που ανήκουν στην ίδια ιεραρχία κλάσεων μπορεί να δεχθούν κλήση μεθόδου ή τελεστή με το ίδιο όνομα (π.χ. `d.forward(10)` ή `d += 10`) και αποκρίνονται με τρόπο που εξαρτάται από την κλάση του αντικειμένου (π.χ. αν το *d* είναι κλάσης *Date* προχωρεί την ημερομηνία κατά 10 ημέρες ενώ αν είναι κλάσης *DateTime* προχωρεί τον χρόνο κατά 10 *sec*).

Πώς γίνονται αυτά; Η παράγωγη κλάση (υποκλάση) κληρονομεί από τη βασική (υπερκλάση) τη μέθοδο (τελεστή) που μας ενδιαφέρει (π.χ.: `const Date& forward(long int dd)`). Στην παράγωγη κλάση *επιφορτώνουμε*(;) μια μέθοδο με το ίδιο όνομα και *παρόμοια επικεφαλίδα* (π.χ.: `const DateTime& forward( long int dd)`). Η μέθοδος αυτή

υπερισχύει[12] αυτής που κληρονομήθηκε και *σχεδόν* κάθε φορά που στέλνουμε το σχετικό μήνυμα σε αντικείμενο της παράγωγης αυτό αποκρίνεται με την υπερισχύουσα μέθοδο. Για να γίνει εκείνο το «*σχεδόν* κάθε φορά» απλώς «κάθε φορά» δηλώνουμε τις μεθόδους μας "`virtual`".

Στην παραπάνω ιστορία έχουμε αφήσει κάποια πράγματα που χρειάζονται διευκρίνιση: ένα ερωτηματικό(;) και μια «*παρόμοια επικεφαλίδα*».

- Πρώτα το «*επιφορτώνουμε*(;)». ΔΕΝ επιφορτώνουμε, τουλάχιστον με την έννοια που ξέρουμε μέχρι τώρα. Στην επιφόρτωση συναρτήσεων μάθαμε ότι οι διάφορες συναρτήσεις με το ίδιο όνομα θα πρέπει να έχουν *διαφορετικές υπογραφές*. Δηλαδή θα πρέπει να έχουν *διαφορά* στις παραμέτρους (ώστε να μπορεί ο μεταγλωττιστής να επιλέξει). Οι πολυμορφικές μέθοδοι *πρέπει να έχουν τις ίδιες ακριβώς παραμέτρους*.
- Πόσο «*παρόμοια επικεφαλίδα*»; Αφού έχουμε ίδιο όνομα και ίδιες παραμέτρους, το «*παρόμοια*» πάει στον τύπο (επιστρεφόμενης τιμής) της συνάρτησης. Ο τύπος μπορεί να είναι ο ίδιος ή –αν δεν είναι– θα πρέπει να είναι *μετατρέψιμος* τύπος βέλους ή αναφοράς. Όπως θα δεις παρακάτω, ο τύπος βέλους της βασικής κλάσης (π.χ. `Date*` ή `Date&`) μπορεί να μετατραπεί σε τύπο βέλους προς την παράγωγη (π.χ. `DateTime*` ή `DateTime&`)· ο τύπος της μεθόδου της παράγωγης κλάσης είναι βέλος παράγωγης κλάσης.[13]

Για παράδειγμα, μπορείς να δεις αυτά τα πράγματα στη μέθοδο *forward()* ή στον "`+=`" των *Date* και *DateTime*:

```
const Date& Date::forward( long int dd )
const DateTime& DateTime::forward( long int ds )
```

ή

```
const Date& Date::operator+=( long int dd )
const DateTime& DateTime::operator+=( long int ds )
```

Και στις δύο περιπτώσεις μπορούσαμε να τις κάνουμε "`virtual`" και τις κάναμε.

Αυτός ο –δυναμικός– πολυμορφισμός ονομάζεται **πολυμορφισμός χρόνου εκτέλεσης** (run-time).

Παρατήρηση: ▶

Αν στη βασική κλάση, μια μέθοδος "`virtual`" έχει προδιαγραφή εξαιρέσεων, οι υπερισχύουσες μέθοδοι στις παράγωγες κλάσεις θα πρέπει να έχουν την ίδια ακριβώς προδιαγραφή εξαιρέσεων. Στο παράδειγμα της §23.11 μας δίνεται η βασική κλάση με υπογραφή καταστροφέα:

```
virtual ~exception() throw();
```

Στην παράγωγη κλάση (*DateXptn*) θα πρέπει να βάλουμε:

```
virtual ~DateTimeXptn() throw() { };
```

Αν παραλείψεις το "`throw()`" ο μεταγλωττιστής θα σου βγάλει λάθος.◀

23.12.1 Πολυμορφισμός Χρόνου Μεταγλώττισης(;)

Ας πούμε ότι μας ενδιαφέρει να κάνουμε πολυμορφικό τον μεταθεματικό "++". Όπως έχουμε ήδη πει, αυτός *δεν μπορεί να επιστρέψει βέλος ή αναφορά* και θα είναι:

```
Date Date::operator++( int )
DateTime DateTime::operator++( int )
```

Μπορούμε να το αφήσουμε έτσι· στην παράγωγη θα έχουμε απόκρυψη του τελεστή της βασικής. Αλλά

---

[12] Από την στιγμή που έχουμε *υπερίσχυση* παύουμε να μιλούμε για *επιφόρτωση*. Κρατούμε αυτόν τον όρο για την περίπτωση που η επικεφαλίδα δεν είναι *παρόμοια*. Το συζητούμε στη συνέχεια.

[13] Αυτό ονομάζεται *συμμεταβλητότητα* (covariance).

- Θα έχουμε διαφορετική συμπεριφορά των αντικειμένων της παράγωγης όταν τα χειριζόμαστε με βέλη και αναφορές της βασικής.
- Δεν μπορούμε να βάλουμε το "`virtual`" για να λύσουμε το πρόβλημα.

Αυτός ο πολυμορφισμός, που μας δίνει πιο γρήγορο πρόγραμμα, ονομάζεται πολυμορφισμός **χρόνου μεταγλώττισης** (compile time) σε αντιδιαστολή με τον «πλήρη» πολυμορφισμό που είναι **χρόνου εκτέλεσης** (run-time). Αλλά όπως καταλαβαίνεις μπορεί να δημιουργήσει πολύ σοβαρά προβλήματα στο πρόγραμμά σου· μην τον χρησιμοποιείς! Γενικώς, όπως μας συμβουλεύει το (CERT 2009):[14]

♦ ***Μην «κρύβεις» κληρονομούμενες μεθόδους που δεν είναι "virtual".***

23.12.2 * Υπερίσχυση ή Επιφόρτωση

Μέχρι τώρα ξέραμε την *επιφόρτωση* (overloading) συναρτήσεων· στο κεφάλαιο αυτό μάθαμε την *υπερίσχυση* (overriding) που είναι κάτι τελείως διαφορετικό. Αν τις μπερδέψεις μπορεί να έχεις προβλήματα, όπως φαίνεται στο παρακάτω παράδειγμα.

Επιστρέφουμε ξανά στο πρόγραμμα της §23.3 και κάνουμε μερικές αλλαγές, στη *B*:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B
```

και στη *D1*:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
private:
   int md1;
}; // D1
```

Η *f()* της *D1 υπερισχύει* –στα αντικείμενα κλάσης *D1*– της *f()* που ορίσαμε στη *B*. Αν βάλουμε στη `main`

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   B b( d );
   cout << b.f() << endl;
```

θα πάρουμε:

```
D1::f()
3
B::f()
2
```

Ας πούμε τώρα ότι αλλάζουμε τη *D1* ως εξής:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
```

[14] Η σύσταση *OBJ02* λέει: «*Do not hide inherited non-virtual member functions*».

```
private:
   int md1;
}; // D1
```

και στη `main` ζητούμε:

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Πρόσεξε τι έχουμε εδώ: Στη βασική κλάση υπάρχει μια μέθοδος

```
   virtual int f() const;
```

και στην παράγωγη μια:

```
   virtual void f( int& aV, int& aDV ) const;
```

Τι σχέση έχουν οι δύο μέθοδοι; Καμιά, απλώς έχουν το ίδιο όνομα. Δηλαδή, στην παράγωγη κλάση, που κληρονομεί την *f()* της *B*, έχουμε *επιφόρτωση*, Τα "`virtual`" δεν παίζουν οποιοδήποτε ρόλο, τουλάχιστον σε ό,τι βλέπουμε!

Δυστυχώς, ο μεταγλωττιστής θα βγάλει πρόβλημα στην:

```
   cout << d.f() << endl;
```

(«`Error E2193 tO.cpp 63: Too few parameters in call to 'D1::f(int &,int &) const' in function main()`»[15]) που, με απλά λόγια, οφείλεται στο εξής: ο μεταγλωττιστής θα αναζητήσει μέθοδο *f()* στη *D1*. Μόλις βρει αυτήν που δηλώνεται στην κλάση θα σταματήσει και δεν θα συνεχίσει την αναζήτηση στη βασική κλάση.

Το πρόβλημα λύνεται αν γράψουμε:

```
   cout << d.B::f() << endl;
```

οπότε και παίρνουμε:

```
B::f()
2
void D1::f()
1  2
```

Ας αλλάξουμε τώρα την *D1* ξαναβάζοντας και την *f()* που είχαμε αρχικώς.

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

Τώρα:

- Η "`virtual int f() const`" *υπερισχύει* της *f()* της βασικής αφού έχει ακριβώς την ίδια δήλωση με εκείνη. Τώρα, το "`virtual`" (στην *f()* της *B*) είναι απαραίτητο!
- Η `virtual void f( int& aV, int& aDV ) const` είναι *επιφόρτωση* της προηγούμενης. Το "`virtual`" είναι χρήσιμο μόνο στην περίπτωση που θα γράψουμε συνάρτηση που θα κληρονομήσει τη *D1* (για παράδειγμα τη *DD1*).

Οι

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
```

[15] Borland BC++, v.5.5.

```
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

δίνουν, χωρίς πρόβλημα:

```
D1::f()
3
void D1::f()
1  2
```

Η συμβουλή λοιπόν είναι: *Δώσε άλλο όνομα στη νέα συνάρτηση*· δεν τελείωσαν τα ονόματα!... Γενικώς:

- ***Απόφευγε να επιφορτώνεις κληρονομούμενες ή υπερισχύουσες (virtual) μεθόδους.***

Σημείωση: ▶

Τώρα, που τα κάναμε όλα σωστά, δοκιμάζουμε το εξής:

```
   B* pD( new D1(1,2) );
   cout << pD->f() << endl;
   int a1, a2;
   pD->f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Ο μεταγλωττιστής μας λέει ότι «η *B* δεν έχει συνάρτηση που να ταιριάζει με την `pD->f(a1, a2)`»!

Αυτό είναι ένα «χαλί» για να περάσουμε στην επόμενη παράγραφο.◀

23.13 Δυναμική Τυποθεώρηση – RTTI

Δες άλλο ένα πρόβλημα παρόμοιο με αυτό που είδαμε στη *Σημείωση* της προηγούμενης παραγράφου. Στην §23.9, όπου είχαμε:

```
   Date* pind[2];
// . . .
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

μετά την

```
   pind[1]->forward( 10 );
```

ζητήσαμε

```
   cout << pind[1]->getDay() << endl;
```

και –μετά τη διόρθωση με το "`virtual`"– είδαμε ότι η "`pind[1]->getDay()`" μας έδωσε "`3`". Δεν δοκιμάσαμε όμως να δούμε αν αυξήθηκαν τα δευτερόλεπτα σε "`43`". Τώρα που τα ρυθμίσαμε όλα ας σιγουρευτούμε ότι η *forward()* προχώρησε σωστά τα δευτερόλεπτα. Ζητούμε:

```
   cout << pind[1]->getSec() << endl;
```

και ο μεταγλωττιστής μας έχει μια νέα έκπληξη: «`'class Date' has no member named 'getSec'`». Σοκ για τους απρόσεκτους! "`protected`" βάλαμε, "`virtual`" βάλαμε, τι δεν βάλαμε; Όποιος(-α) μελετάει προσεκτικά ξέρει πως αυτά είναι άσχετα με το πρόβλημα:

- Το "`protected`" δίνει πρόσβαση στις μεθόδους της παράγωγης προς μέλη της βασικής.
- Το "`virtual`" δίνει δυνατότητα επιλογής της σωστής *πολυμορφικής* μεθόδου.

Εδώ τι συμβαίνει; Η *getSec()* δεν είναι πολυμορφική (**virtual**) ώστε να δημιουργήσει πρόβλημα επιλογής μεθόδου· είναι απλώς μέθοδος της παράγωγης κλάσης. Θα πρέπει εσύ να βάλεις στο πρόγραμμά σου τις εντολές που θα ερευνήσουν τον τύπο του αντικειμένου που δείχνει το βέλος και να οδηγήσουν στη σωστή εκτέλεση. Πώς διορθώνουμε τα πράγματα; Με **δυναμική τυποθεώρηση** (dynamic casting):

```
   cout << dynamic_cast<DateTime*>(pind[1])->getSec() << endl;
```

Η `dynamic_cast<`*τύπος βέλους*`>(`τιμή-βέλος`)` εξετάζει αν το όρισμα (τιμή-βέλος) δείχνει στην πραγματικότητα μια τιμή του τύπου βέλους που αναφέρεται ως παράμετρος. Αν είναι επιστρέφει βέλος αυτού του τύπου αλλιώς επιστρέφει "`0`" (`NULL`).

Στην περίπτωσή μας αφού βεβαιωθεί ότι το pind[1] δείχνει αντικείμενο κλάσης *DateTime* μας επιστρέφει βέλος τύπου *DateTime** προς το pind[1].

Εδώ εμείς ξέρουμε ότι η μετατροπή του βέλους pind[1] σε βέλος προς αντικείμενο *DateTime* γίνεται σίγουρα (δεν θα πάρουμε "0") και ύστερα από αυτό μπορούμε να ζητήσουμε τη *getSec()*. Λέμε ότι ο τύπος βέλους προς αντικείμενο της βασικής κλάσης είναι **μετατρέψιμος** (convertible) σε τύπο βέλους προς αντικείμενο παράγωγης κλάσης. Αν δεν είχαμε αυτήν τη σιγουριά θα έπρεπε να δουλέψουμε ως εξής:

```
DateTime* temp( dynamic_cast<DateTime*>(pind[1]) );
if ( temp != 0 )
   cout << temp->getSec() << endl;
```

Όπως βλέπεις, η δυναμική τυποθεώρηση βοηθάει να βρούμε απάντηση στο ερώτημα:

- *Κάποιο συγκεκριμένο βέλος δείχνει αντικείμενο κάποιας συγκεκριμένης κλάσης (ή γενικότερα μεταβλητή κάποιου τύπου);*

Αυτό το ερώτημα δεν μπορούμε να το αποφύγουμε όταν χρησιμοποιούμε αυτά που μάθαμε στις προηγούμενες παραγράφους: βέλη της βασικής κλάσης προς αντικείμενα παράγωγης (ή παραγώγων).

Στο παρακάτω κομμάτι προγράμματος βλέπουμε διάφορες περιπτώσεις χρήσης αυτής της τεχνικής:

```
Date* pd = new DateTime( 2004, 4, 4, 4, 44, 44 );
if ( dynamic_cast<Date*>(pd) == 0 )  cout << "OXI" << endl;
                                else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd) == 0 )  cout << "OXI"<<endl;
                                    else  cout << "NAI"<<endl;
Date* pd0 = new Date( 2005, 5, 5 );
if ( dynamic_cast<Date*>(pd0) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd0) == 0 )  cout << "OXI"<<endl;
                                     else  cout << "NAI"<<endl;
```

Αποτέλεσμα:

```
NAI
NAI
NAI
OXI
```

Το *pd*, παρ' όλο που έχει δηλωθεί ως βέλος προς αντικείμενο κλάσης *Date*, δείχνει προς αντικείμενο κλάσης *DateTime*. Έτσι,

- Στη δεύτερη περίπτωση, η `dynamic_cast<DateTime*>(pd)` επιστρέφει ως τιμή βέλος τύπου `DateTime*`.
- Στην πρώτη περίπτωση επιστρέφει το ίδιο το *pd*, που δείχνει προς το υποαντικείμενο κλάσης *Date* του αντικειμένου κλάσης `DateTime` που δείχνει το *pd*.
- Πρόσεξε την τελευταία περίπτωση: το *pd*0 δείχνει αντικείμενο κλάσης *Date*. Επομένως δεν μπορεί να μετατραπεί σε βέλος προς αντικείμενο της παράγωγης κλάσης και η `dynamic_cast<DateTime*>(pd0)` επιστρέφει 0.

Παρατήρηση: ▶

Αν έχουμε δηλώσει:

```
DateTime* pdt( new DateTime(2004, 4, 4, 4, 44, 44) );
```

η "`dynamic_cast<Date*>(pdt) != 0`" είναι η διατύπωση σε C++ της ερώτησης: «*δείχνει το pdt αντικείμενο κλάσης Date;*» Φυσικά η απάντηση είναι "`NAI`" και αυτό βγάζει η

```
if ( dynamic_cast<Date*>(pdt) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
```

Αν ξαναγυρίσεις στην §23.2 μπορείς να πεις ότι σε ένα αντικείμενο κλάσης *DateTime* το *υποαντικείμενο κλάσης Date** δίνεται από την παράσταση

"`*(dynamic_cast<Date*>(this))`" ◀

Εκτός από τη δυναμική τυποθεώρηση, **εξακρίβωση τύπου κατά την εκτέλεση** (Run Time Type Identification, RTTI) μπορεί να γίνει και με τον τελεστή `typeid`. Ας ξαναθυμηθούμε μερικά πράγματα που μάθαμε στο *Μέρος Α*, §2.8.1: «*μπορείς να δώσεις:*

`typeid(` *παράσταση* `).name()`

και να πάρεις τον τύπο του αποτελέσματος της παράστασης. Στο πρόγραμμά σου θα πρέπει να περιλάβεις (`#include`) *το "*`typeinfo`*".*» Μην αμφιβάλλεις λοιπόν ότι αν δώσεις:

```
cout << typeid(*pd).name() << endl;
cout << typeid(*pd0).name() << endl;
```

θα πάρεις:

```
DateTime
Date
```

Ο τελεστής "`typeid`" επιστρέφει ένα αντικείμενο κλάσης *type_info*, για την οποίαν έχουν επιφορτωθεί και οι τελεστές "`==`" και "`!=`". Έτσι, αν δώσουμε:

```
if ( typeid(*pd) == typeid(Date) )  cout << "NAI" << endl;
                                else  cout << "OXI" << endl;
if ( typeid(*pd) == typeid(DateTime) )  cout << "NAI" << endl;
                                   else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(Date) )  cout << "NAI" << endl;
                                 else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(DateTime) )  cout << "NAI" << endl;
                                    else  cout << "OXI" << endl;
```

θα πάρουμε:

```
OXI
NAI
NAI
OXI
```

23.13.1 Μετατροπή Αναφορών

Ας πούμε ότι έχουμε:

```
DateTime dt( 2014, 3, 3, 3, 33, 33 );
cout << "dt = " << dt << endl;
Date& rd( dt );
cout << "rd = " << rd << endl;
```

Αφού όπου μπορούμε να βάλουμε αντικείμενο κλάσης *Date* μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης *DateTime* γράψαμε την εντολή "`Date& rd(dt)`". Αυτές οι εντολές θα δώσουν:

```
dt = 3.3.2014 3:33:33
rd = 3.3.2014
```

«Ε, φυσικά έχουμε τεμαχισμό!» θα πεις. Όχι! Δεν έχουμε τεμαχισμό, αφού δεν έχουμε κάποια αντιγραφή τιμής. Να θυμίσουμε ότι στην §13.4 λέγαμε ότι τα δύο ονόματα –στην περίπτωσή μας *dt* και *rd*– *καθορίζουν την ίδια θέση της μνήμης*. Αυτό που βλέπουμε ως τιμή της *rd* οφείλεται στο ότι γράφεται με τον "`<<`" της *Date*. Πράγματι, αν βάλουμε:

```
DateTime* pRd( reinterpret_cast<DateTime*>(&rd) );
cout << "*pRd = " << *pRd << endl;
```

θα πάρουμε:

```
*pRd = 3.3.2014 3:33:33
```

Θα πεις τώρα «Θα μου τύχει ποτέ να γράψω τέτοια πράγματα στο πρόγραμμά μου;» Όχι βέβαια, αλλά μπορεί να σου τύχει να γράψεις μια συνάρτηση σαν την:

```
void aFunc( Date& aDate )
```

και να θέλεις

- αν κληθεί με όρισμα κλάσης *DateTime* να εκτελεσθούν οι εντολές *EDT* ενώ
- αν κληθει με όρισμα κλάσης *Date* να εκτελεσθούν οι εντολές *ED*.

Ο πάγιος τρόπος χειρισμού του προβλήματος δεν βασίζεται στην ερμηνευτική τυποθεώρηση αλλά στη δυναμική που μπορούμε να την κάνουμε όχι μόνο σε βέλη αλλά και σε τύπους αναφοράς:

```
void aFunc( Date& aDate )
{
// . . .
   try
   {
      DateTime& aDT( dynamic_cast<DateTime&>(aDate) );
   // εντολές EDT για την επεξεργασία του aDT
   }
   catch( bad_cast& )
   {
   // εντολές ED για την επεξεργασία του aDate
   }
// . . .
}
```

Εδώ προσπαθούμε να δούμε –μέσω του τοπικού αντικειμένου *aDT*– την παράμετρο αναφοράς *aDate* ως αντικείμενο κλάσης *DateTime*.

Αν αποτύχει η προσπάθεια ο τελεστής `dynamic_cast` θα ρίξει εξαίρεση κλάσης (*std*::) *bad_cast*. Στο πρόγραμμά σου θα πρέπει έχεις βάλει "`#include <typeinfo>`" όπου υπάρχει ο ορισμός της *bad_cast*.

Εδώ έχουμε μια περίπτωση που η δομή **try/catch** για έγερση και διαχείριση εξαιρέσεων χρησιμοποιείται σαν δομή **if-else**!

23.14 "is_a" ή "has_a"

Στη συνέχεια θα εξετάσουμε ακροθιγώς μερικά σημεία που έχουν σχέση, κατά κύριο λόγο, με αντικειμενοστρεφή σχεδίαση.

Ένας απλουστευτικός τρόπος να δούμε την κληρονομιά είναι και ο εξής: Περιλαμβάνουμε σε κάθε αντικείμενο της παράγωγης ένα αντικείμενο της βασικής. Αυτό όμως μπορούμε να το πετύχουμε πιο απλά. Για παράδειγμα, αντί για:

```
class OfferedCourse :  public Course
{
public:
// . . .
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

θα μπορούσαμε να έχουμε:

```
class OfferedCourse
{
public:
//   . . .
private:
   Course       ocC;
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

Στην περίπτωση αυτή λέμε ότι τα αντικείμενα των δύο κλάσεων συνδέονται με σχέση `has_a` (*OfferedCourse* `has_a` *Course*).

Ποιος τρόπος είναι προτιμότερος; Στο Project 6 ξαναγράφουμε με τον πρώτο τρόπο αυτά που είδαμε στο Project 4. Προσπάθησε να ξαναγράψεις έστω και μέρος του προγράμματος με τον δεύτερο τρόπο και θα καταλάβεις ότι ο πρώτος τρόπος είναι σαφώς προτιμότερος.

Εδώ θα σου δώσουμε ένα πιο απλό παράδειγμα. Έστω ότι έχουμε:

```
class B
```

```
{
public:
   B( int newMb=0 )
      :  mb( newMb ) { };
   virtual ~B() { };
   void f() {  cout << "this is base f" << endl;  };
   virtual void display( ostream& tout )
   {  tout << "base display: mb = " << mb;  };
   virtual void swap( B& other )
   {  std::swap( mb, other.mb );  }
private:
   int mb;
}; // B

class D1 :  public B
{
public:
   D1( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   virtual void swap( D1& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D1
```

Αν δώσουμε:

```
   D1  x( 3, 5 ), u( 19, 17 );
   x.f();
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
   x.B::display( cout );  cout << endl;
   x.swap( u );
   cout << "after swap" << endl;
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
```

θα πάρουμε:

```
this is base f
x: base display: mb = 3   derived display: md = 5
u: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
x: base display: mb = 19   derived display: md = 17
u: base display: mb = 3   derived display: md = 5
```

Αν θέλουμε να έχουμε την ίδια λειτουργικότητα χωρίς κληρονομιές θα πρέπει να έχουμε:

```
class D2
{
public:
   D2( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   ~D2() { };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  bo.f();  };
   void bdisplay( ostream& tout ) {  bo.display( tout );  };
   void swap( D2& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
private:
```

```
   B   bo;
   int md;
}; // D2
```

Όπως βλέπεις, πρέπει να εξοπλίσουμε τη *D2* με δύο ενδιάμεσες συναρτήσεις –*f()* και *bdisplay()*– που επιτρέπουν την πρόσβαση σε μεθόδους της *B*. Θα πάρουμε τα ίδια αποτελέσματα με την πρώτη περίπτωση δίνοντας:

```
   D2 y( 3, 5 ), t( 19, 17 );
   y.f();
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
   y.bdisplay( cout );  cout << endl;
   y.swap( t );
   cout << "after swap" << endl;
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
```

Παρατήρηση:▶

Αν, παραβιάζοντας τον κανόνα που έχουμε βάλει, μεταφέρουμε τη δήλωση "`B bo;`" στην περιοχή "`public`" τα πράγματα απλουστεύονται κάπως:

```
class D3
{
public:
   D3( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void swap( D3& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
   B   bo;
private:
   int md;
}; // D3
```

Δίνοντας:

```
   D3 z( 3, 5 ), v( 19, 17 );
   z.bo.f();
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   z.bo.display( cout );  cout << endl;
   z.swap( v );
   cout << "after swap" << endl;
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
```

παίρνουμε τα ίδια αποτελέσματα που πήραμε και πιο πάνω.◀

Ένα καίριο πρόβλημα που δεν φαίνεται στο παράδειγμά μας είναι το εξής: ενώ μπορούμε να δηλώσουμε "`B* p(new D1)`" δεν μπορούμε να δηλώσουμε "`B* p(new D2)`". Στην πρώτη περίπτωση το βέλος προς αντικείμενο της παράγωγης κλάσης "`new D1`" μετατρέπεται σε βέλος προς αντικείμενο της βασικής και στη συνέχεια μπορούμε να βρούμε τον τύπο του αντικειμένου (που δείχνει το βέλος) με δυναμική τυποθεώρηση ή τον τελεστή "`typeid`". Τι θα κάνεις αν προσπαθήσεις να μετατρέψεις την κλάση *CourseCollection* (αλλά και τη *StudentCollection*) του Project 6; Εκεί υπάρχει ένας δυναμικός πίνακας –ο ccArr– που κάθε στοιχείο του είναι τύπου "`Course*`" αλλά μπορεί να δείχνει και αντικείμενο κλάσης *OfferedCourse*. Πώς θα τον αλλάξεις; Μήπως θα μπορούσαμε να χρησιμοποιήσουμε βέλη "`void*`" και ερμηνευτική τυποθεώρηση; Ναι, βέβαια, αλλά δεν είναι και τόσο απλό. Σε ένα τέτοιο βέλος δεν μπορείς να κάνεις απόπαραπομπή αν δεν κάνεις προηγουμένως τυποθεώρηση. Σε ποιον τύπο όμως; Θα πρέπει κάπου να έχεις φυλαγμένη τη σχετική πληροφορία…

Η σχέση "`is_a`" μπορεί να μας λύσει προβλήματα με απλό τρόπο αλλά, από την άλλη μεριά, *οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και*

τη σωστή νοηματική σχέση. Ας πούμε ότι έχεις να γράψεις μια κλάση *Computer* που περιγράφει έναν υπολογιστή ενώ έχεις ήδη μια κλάση *HardDisk* που περιγράφει έναν σκληρό δίσκο. *Μην ξεκινήσεις* με τη λογική *Computer* `is_a` *HardDisk* –ώστε ένα αντικείμενο *Computer* να περιλαμβάνει ένα αντικείμενο *HardDisk*– επειδή συνειδητοποίησες ότι οι διεπαφές (περιοχές "`public`") μοιάζουν πολύ. Η σωστή σχέση είναι *Computer* `has_a` *HardDisk*.

Στη σχέση "`has_a`" θα επανέλθουμε.

Πάντως αν περιορίσεις τις επιλογές σου μεταξύ "`is_a`" και "`has_a`" μπορεί να οδηγηθείς σε σχεδιαστικά λάθη:

Παράδειγμα

Ας πούμε ότι προσθέτουμε στην *OfferedCourse* ένα μέλος επιπλέον, ας το πούμε *ocSemester*, για να κρατούμε και το εξάμηνο που προσφέρθηκε το μάθημα. Αν τώρα, σε ένα πρόγραμμα κάνουμε συγκριτική επεξεργασία της διεξαγωγής των μαθημάτων για τα τελευταία έξη εξάμηνα θα έχουμε εξάδες αντικειμενων *OfferedCourse* που θα έχουν το ίδιο υποαντικείμενο *Course*. Για να το απόφύγουμε μπορούμε

- να βάλουμε στην *OfferedCourse* απλώς ένα βέλος προς αντικείμενο κλάσης *Course* ή
- να βάλουμε στην *OfferedCourse* μόνον τον κωδικό μαθήματος.

Αυτή είναι μια περίπτωση **αντιστοίχισης** (association) **πολλά-προς-ένα** (many-to-one) ή *N* : 1· ούτε "`is_a`" ούτε "`has_a`".

Καλή λοιπόν η κληρονομιά, αλλά δεν είναι κατ' ανάγκη η καλύτερη επιλογή πάντοτε· χρειάζεται λίγη σκέψη πριν τη χρησιμοποιήσουμε.

23.15 "private", "protected" και "public"

Γενικότερα, όταν σχεδιάζεις μια κλάση θα πρέπει να σκεφτείς καλά πριν αποφασίσεις τι θα βάλεις **private**, τι θα βάλεις **protected** και τι θα βάλεις **public**.

- ***Αν ένα μέλος κλάσης είναι*** `private` ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης και β) φίλες κλάσεις ή συναρτήσεις της κλάσης.***
- ***Αν ένα μέλος είναι*** **protected** ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης β) φίλες συναρτήσεις της κλάσης γ) μεθόδους των παραγώγων κλάσεων της και δ) φίλες συναρτήσεις παραγώγων κλάσεων.***
- ***Αν ένα μέλος είναι*** **public** ***μπορεί να χρησιμοποιηθεί από οποιαδήποτε συνάρτηση.***

Όταν γράφεις μια παράγωγη κλάση θα πρέπει να αποφασίσεις για τον τρόπο που αυτή κληρονομεί τη βασική:

- ***Αν δηλώσεις "*** `class D: private B` ***" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***).***
- ***Αν δηλώσεις "*** `class D: protected B` ***" τότε τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από α) τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) τις μεθόδους και τις φίλες των παραγώγων κλάσεων της D. Δικαίωμα να μετατροπής ενός*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***) έχουν μόνον α) οι μεθοδοι της D και οι φίλες συναρτήσεις της D και β) οι μέθοδοι και τις φίλες των παραγώγων κλάσεων της D.***
- ***Αν δηλώσεις "*** `class D: public B` ***" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από οποιαδήποτε συνάρτηση β) οποιαδήποτε συνάρτηση μπορεί να μετατρέψει ένα βέλος*** `D*` ***σε*** `B*` ***και μια αναφορά*** `D&` ***σε*** `B&`***.***

Ας συμπληρώσουμε λοιπόν τον κανόνα που δώσαμε πιο πάνω:

♦ ***Αν έχουμε δηλώσει "`class D: public B`" τότε όπου μπορούμε να βάλουμε, στο πρόγραμμά μας, ένα αντικείμενο κλάσης B μπορούμε να βάλουμε και αντικείμενο κλάσης D.***

Αυτή είναι η ακριβέστερη διατύπωση της *δυνατότητας υποκατάστασης* που είδαμε στην §23.2.

Η κληρονομιά "**`private`**" είναι στην πραγματικότητα ένας τρόπος υλοποίησης της συσχέτισης "**`has_a`**" που είδαμε στην προηγούμενη παράγραφο. Στο πρόγραμμα που γράψαμε εκεί βάζουμε μια ακόμη κλάση:

```
class D4 : private B
{
public:
   D4( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D4() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  B::f();  };
   void bdisplay( ostream& tout ) {  B::display( tout );  };
   void swap( D4& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D4
```

Οι εντολές:

```
   D4 v( 3, 5 ), w( 19, 17 );
   v.f();
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
   v.bdisplay( cout );  cout << endl;
   v.swap( w );
   cout << "after swap" << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
```

δίνουν:

```
this is base f
v: base display: mb = 3   derived display: md = 5
w: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
v: base display: mb = 19   derived display: md = 17
w: base display: mb = 3   derived display: md = 5
```

Να παρατηρήσουμε τα εξής:

- Λέμε παραπάνω: «*τα μέλη της B που είναι "`public`" ή "`protected`" μπορεί να χρησιμοποιηθούν από τις μεθόδους της D.*» Με βάση αυτήν τη δυνατότητα γράφουμε τις μεθόδους *D4*`::`*f()*, *D4*`::`*bdisplay()* και *D4*`::`*swap()*. Δεν γίνεται να χρησιμοποιηθούν κάπως «πιο έξω», ας πούμε στη **`main()`**; Όχι! Αν γράψεις "**`v.f();`**" ο μεταγλωττιστής θα σου πεί: «**`'B::f()' is not accessible in function main()`**». Παρόμοια αντιμετώπιση θα βρεις αν απόπειραθείς να γράψεις "**`v.B::display(cout);`**". Έτσι, χρειάζονται οι ενδιάμεσες συναρτήσεις *D4*`::`*f()*και *D4*`::`*bdisplay()*, όπως ακριβώς και στη *D2*.
- Λέμε παραπάνω: «*μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα* **`D*`** *σε* **`B*`** *(και* **`D&`** *σε* **`B&`***).*» Με βάση αυτήν τη δυνατότητα γράφουμε τη *D4*`::`*swap()* που είναι ολόιδια με τη *D1*`::`*swap()*. Η κλήση "**`B::swap(other);`**" απαιτεί μετατροπή μιας αναφοράς *D4***`&`** (*other*) σε *B***`&`** που περιμένει η *B*`::`*swap()*. Ούτε αυτή η δυνατότητα μπορεί να χρησιμοποιηθεί «πιο έξω». Αν, ας πούμε, έχεις μια συνάρτηση

```
int f1( B& ap )
```

```
{
//...
```

τότε η κλήση:

```
cout << f1( v );
```

δεν είναι δεκτή· «`Cannot initialize 'B &' with 'D4' in function main()`» θα μας πει ο μεταγλωττιστής.

Ακόμη, δεν μπορείς μέσα στο πρόγραμμά σου να γράψεις:

```
B* bp( new D4 );
```

Όπως καταλαβαίνεις η περίπτωση "`class D: private B`" είναι χρήσιμη όταν θέλεις να χρησιμοποιήσεις τη *B* για την *υλοποίηση* των μεθόδων της *D*. Για τον λόγο αυτόν θα δεις να γράφεται "*D* `is_implemented_in_terms_of` *B*" εκτός από "*D* `has_a` *B*".

Τα ίδια ισχύουν σε μεγάλο ποσοστό και για την περίπτωση "`class D: protected B`". Στην περίπτωση αυτήν παράγωγες κλάσεις της *D* βλέπουν αυτά που κληρονόμησε η *D* από τη *B* ως "`protected`".

23.16 Αφηρημένες Κλάσεις

Αν παρατηρήσουμε τα παραδείγματα που δώσαμε μέχρι τώρα βλέπουμε το εξής: *Σε κάθε παραγωγή η βασική κλάση είναι πιο απλή από την παράγωγη*. Αυτός είναι ένας τρόπος να διαχειριζόμαστε την πολυπλοκότητα στα προγράμματα: ασχολούμαστε με τα προβλήματα σταδιακά, οι δυσκολίες έρχονται μια-μια και όχι όλες μαζί. Ας δούμε άλλο ένα παράδειγμα:

Μια εταιρεία παραγωγής/διανομής ηλεκτρικής ενέργειας έχει τρεις κατηγορίες πελατών:

1. *μεγάλες βιομηχανικές μονάδες που χρεώνονται με* 0.03 *ευρώ/ kWh,*
2. *μικρομεσαίες βιοτεχνικές μονάδες που χρεώνονται με* 0.06 *ευρώ/kWh συν μια πάγια χρέωση που είναι διαφορετική για κάθε καταναλωτή και*
3. *οικιακούς καταναλωτές που χρεώνονται με πάγια χρέωση –διαφορετική για κάθε καταναλωτή– συν* 0.06 *ευρώ/kWh για τις πρώτες A kWh και* 0.1 *ευρώ/kWh για την κατανάλωση πέραν της A. Η A καθορίζεται χωριστά για κάθε καταναλωτή.*

Κάθε καταναλωτής, άσχετα από την κατηγορία του, έχει έναν κωδικό αριθμό (θετικός ακέραιος).

Να ορισθεί ιεραρχία κλάσεων για να παρασταθούν τα στοιχεία των πελατών. Κάθε κλάση θα έχει οπωσδήποτε μέθοδο για τον υπολογισμό της χρέωσης του πελάτη.

Ας ξεκινήσουμε από την πρώτη κατηγορία, για την οποίαν γράφουμε μια κλάση, ας την πούμε *Industrial*. Τι χρειάζεται να κρατούμε για έναν πελάτη αυτής της κατηγορίας; Μόνον τον κωδικό αριθμό. Και τι μεθόδους θα χρειαστούμε; Μια μέθοδο για να παίρνουμε τον κωδικό, μια για να τον ορίζουμε και μια για τη χρέωση.

Πάμε τώρα στη δεύτερη κατηγορία (κλάση *SmallMediumEnt*). Εδώ χρειαζόμαστε όλα όσα έχει η πρώτη κλάση και ακόμη ένα μέλος για την πάγια χρέωση. Φυσικά θα χρειαστούμε και δύο μεθόδους για τον χειρισμός αυτού του μέλους.

Στην τρίτη κατηγορία (κλάση *HomeCons*) θα χρειαστούμε όσα έχει η δεύτερη και ακόμη ένα μέλος για την *A* και δύο μεθόδους για τον χειρισμό της.

Επομένως, βασική κλάση θα είναι η (απλούστερη από όλες) *Industrial*. Η επόμενη απλούστερη είναι η κλάση *SmallMediumEnt* που θα παράγεται από την πρώτη. Η κλάση *HomeCons* είναι η πιο πολύπλοκη και θα είναι παράγωγη *SmallMediumEnt*.

Μέθοδος για τη χρέωση θα υπάρχει και στις τρεις κλάσεις (πολυμορφική). Φυσικά δεν είναι ίδια και για τις τρεις κλάσεις.

Φτάσαμε λοιπόν σε μια ιεραρχία:

Industrial ← *SmallMediumEnt* ← *HomeCons*

Αλλά, μπορούμε να πούμε ότι μια "*μικρομεσαία βιοτεχνική μονάδα*" είναι μια "*μεγάλη βιομηχανική μονάδα*" ή ότι ένας "*οικιακός καταναλωτής*" είναι μια "*μικρομεσαία βιοτεχνική μονάδα*"; Όχι φυσικά! Εκείνο που μπορούμε να πούμε είναι ότι: Από προγραμματιστική άποψη

- ο λογαριασμός μιας "*μικρομεσαίας βιοτεχνικής μονάδας*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μεγάλης βιομηχανικής μονάδας*" και
- ο λογαριασμός ενός "*οικιακού καταναλωτή*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μικρομεσαίας βιοτεχνικής μονάδας*".

Δηλαδή, για να βρούμε τη λογική της ιεραρχίας των κλάσεων που βγάλαμε πρέπει *να προχωρήσουμε κατά ένα επίπεδο αφαίρεσης*.

Στο Σχ. 23-6 βλέπεις μια άλλη ιεραρχία κλάσεων για το παράδειγμά μας. Εισάγουμε μια νέα κλάση, την

```
class ElecConsumer
{
public:
   ElecConsumer( int consCodeP );
   unsigned long int getConsCode() const {  return ecConsCode;  }
   void setConsCode( int consCodeP );
   virtual double charge( double consumption ) const = 0;
protected:
   unsigned long int ecConsCode;
}; // ElecConsumer
```

Πρόσεξε όμως ένα ιδιαίτερο χαρακτηριστικό: Η εικονική μέθοδος *charge* δηλώνεται αλλά δεν ορίζεται· αυτό το δείχνουμε με εκείνο το "`= 0`". Λέμε ότι η *charge* είναι μια **γνήσια εικονική μέθοδος** (true virtual method).

Μια κλάση που έχει μια τουλάχιστον γνήσια εικονική μέθοδο λέγεται **αφηρημένη** (abstract) κλάση. Φυσικά, μια τέτοια κλάση, αφού είναι ελλιπής, δεν μπορεί να έχει αντικείμενα. Σε αντιδιαστολή, οι άλλες κλάσεις, που είναι πλήρεις, και έχουν δικά τους αντικείμενα ονομάζονται **συγκεκριμένες** (concrete) κλάσεις.

Χρησιμοποιούμε τις αφηρημένες κλάσεις για να δημιουργούμε "σωστές" ιεραρχίες κλάσεων, όπου φυσικά μια αφηρημένη κλάση (όπως η *ElecConsumer*) είναι βασική κλάση ενώ οι παράγωγές της είναι συγκεκριμένες. Για τον λόγο αυτόν θα τη δεις και ως **αφηρημενη βασική κλάση**. Σε μια ιεραρχία κλάσεων μπορεί να υπάρχουν αφηρημένες κλάσεις σε περισσότερα από ένα στάδια παραγωγής.

Από νοηματική άποψη η νέα ιεραρχία κλάσεων είναι πολύ καλύτερη από την αρχική. Πράγματι, στην αφηρημένη βασική κλάση *ElecConsumer* υπάρχουν τα γενικά χαρακτηριστικά ενός καταναλωτή ηλεκτρικής ενέργειας. Στη συνέχεια, αντί της

$$Industrial \leftarrow SmallMediumEnt \leftarrow HomeCons$$

έχουμε τρεις παραγωγές:

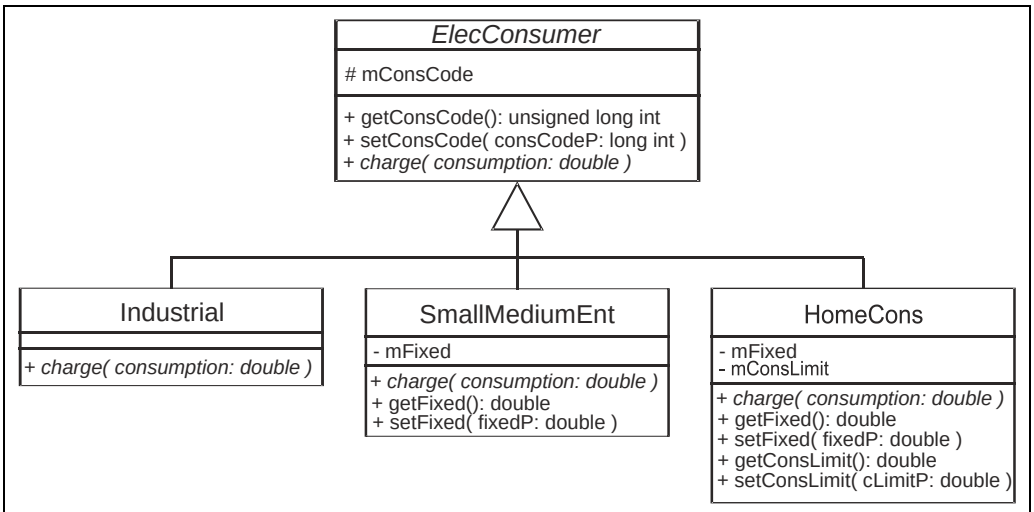
$$ElecConsumer \leftarrow Industrial$$
$$ElecConsumer \leftarrow SmallMediumEnt$$
$$ElecConsumer \leftarrow HomeCons$$

που έχουν νόημα: Ο *βιομηχανικός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*, η *μικρομεσαία επιχείρηση* `is_a` *καταναλωτής ηλεκτρικής ενέργειας* και ο *οικιακός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*.

23.17 Η Σειρά Δημιουργίας

Η σειρά δημιουργίας ενός αντικειμένου, που είδαμε εν μέρει στις §3.1.2 και §10.3 μπορεί τώρα να δοθεί ολοκληρωμένη:

- Δημιουργούνται τα υποαντικείμενα των εικονικών βασικών κλάσεων όπως εμφανίζονται στη λίστα κληρονομιάς.



Σχ. 23-6 Οι τρεις αληθινές κλάσεις, *Industrial*, *SmallMediumEnt*, *HomeCons*, κληρονομούν την αφηρημένη βασική κλάση *ElecConsumer*.

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται στην κλάση.
- Εκτελείται το σώμα του δημιουργού.

23.18 Πολλαπλή Κληρονομιά

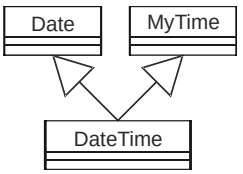
Έστω ότι έχουμε την κλάση:

```
class MyTime
{
friend ostream& operator<<( ostream& tout, const MyTime& rhs );
public:
   MyTime( int hp = 0, int minp = 0, int sp = 0 );
   unsigned int getHour() const {  return mtHour;  };
   unsigned int getMin() const {  return mtMin;  };
   unsigned int getSec() const {  return mtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
   MyTime& operator++();
protected:
   unsigned int mtHour;    // hour (0 .. 23)
   unsigned int mtMin;     // minutes (0 .. 59)
   unsigned int mtSec;     // seconds (0 .. 59)
}; // MyTime
```

Έχουμε επίσης τη γνωστή μας *Date*. Δες έναν άλλον τρόπο να πάρουμε μια κλάση *DateTime*:

```
class DateTime : public Date, public MyTime
{
friend ostream& operator<<( ostream& tout, const DateTime& rhs );
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 )
      : Date( yp, mp, dp ), MyTime( hp, minp, sp ) { };
}; // DateTime

ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return ( tout << rhs.dDay << '.' << rhs.dMonth << '.' << rhs.dYear << ' '
                 << rhs.mtHour << ':' << rhs.mtMin << ':' << rhs.mtSec );
} // operator<< DateTime
```

Δηλαδή η *DateTime* κληρονομεί δύο κλάσεις: τη *Date* και τη *MyTime*. Αυτό συμβολίζεται διαγραμματικώς όπως βλέπεις στο Σχ. 23-7.



Σχ. 23-7 Η *DateTime* κληρονομεί τη *Date* και τη *MyTime*.

Γενικώς μια παράγωγη κλάση μπορεί να κληρονομεί

- πολλές κλάσεις και
- όχι κατ' ανάγκη με τον ίδιο τρόπο, π.χ.:

```
class X : public A, private B, public C { /* ... */ };
```

Αν δηλώσεις:

```
DateTime dt( 2007, 11, 26, 19, 30, 31 );
```

τότε η *dt* έχει όλες τις ιδιότητες που κληρονόμησε και από τις δύο βασικές κλάσεις! Όλες; Και με τον "`++`" που ορίζεται και στη *Date* και στη *MyTime* τι γίνεται; Εδώ έχουμε μια περίπτωση **ασάφειας** ή **αμφιβολίας** (ambiguity) η οποία λύνεται έτσι:

```
dt.Date::operator++();
cout << dt << endl;
dt.MyTime::operator++();
cout << dt << endl;
```

Αποτέλεσμα:

```
27.11.2007 19:30:31
27.11.2007 19:30:32
```

Η πρώτη, `dt.Date::operator++()`, αυξάνει την ημέρα από 26 σε 27 ενώ η δεύτερη, `dt.MyTime::operator++()`, αυξάνει τα δευτερόλεπτα από 31 σε 32.

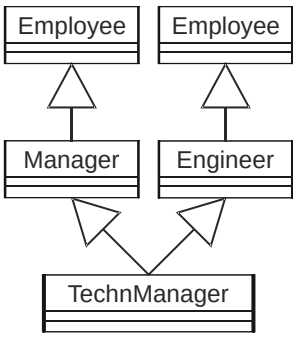
Οι περιπλοκές όμως μπορεί να είναι πιο εντυπωσιακές:

```
class B1 : public L { /* ... */ };
class B2 : public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```

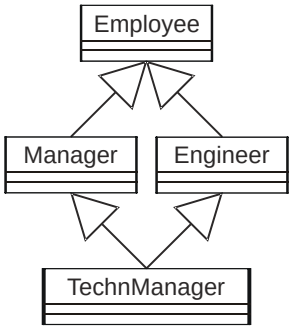
Κάθε αντικείμενο κλάσης *D* έχει δύο υποαντικείμενα κλάσης *L*, το ένα έρχεται από τη *B2* και το άλλο από τη *B3*. Απίθανη περίπτωση; Κάθε άλλο! Βάλε αντί για *L*, *Employee*, αντί για *B1*, *Manager*, αντί για *B2*, *Engineer*, αντί για *D*, *TechnManager* και έχεις το πιο «χειροπιαστό» παράδειγμα στο Σχ.23-8.

Η C++ σου δίνει δυνατότητα να ξεχωρίσεις τα δύο υποαντικείμενα: `B2::L` και `B3::L`. Αλλά το ερώτημα είναι: πόσο χρήσιμο είναι να έχουμε δύο υποαντικείμενα; Στο παράδειγμά μας, του Σχ. 23-8, είναι φανερό ότι για τον Τεχνικό Διευθυντή θέλουμε τα στοιχεία που κρατούμε για κάθε υπάλληλο μια φορά μόνον. Θέλουμε μια εικόνα κληρονομιάς σαν αυτήν του Σχ. 23-9. Αυτό επιτυγχάνεται με την **εικονική** (virtual) κληρονομιά:

```
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```



Σχ. 23-8 Η *TechnManager* κληρονομεί την *Employee* δύο φορές.



Σχ. 23-9 Με κληρονομιά `virtual` η *TechnManager* κληρονομεί την *Employee* μια φορά.

23.19 Διεπαφές

Όπως λέγαμε στην §19.1.3, «*Η οποιαδήποτε επαφή του κάθε προγράμματος με το αντικείμενο γίνεται μέσω αυτών που υπάρχουν στις περιοχές "*`public`*". Αυτά αποτελούν και το τμήμα διεπαφής (interface part) του αντικειμένου.*»

Η Java[16] δίνει την εξής δυνατότητα στον προγραμματιστή: να ορίσει ένα ανεξάρτητο τμήμα διεπαφής και στη συνέχεια να ορίσει τις κλάσεις που το χρησιμοποιούν. Για παράδειγμα, αντιγράφουμε από τα Java Tutorials[17]

```
interface Bicycle
{
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

Αυτή είναι η διεπαφή μιας κλάσης που παριστάνει ποδήλατα. Αν στη συνέχεια θέλουμε μια κλάση για ποδήλατα ACME γράφουμε:

```
class ACMEBicycle implements Bicycle
{
    int cadence = 0;
    int speed = 0;
    int gear = 1;
// . . .
    void speedUp(int increment)
    {   speed = speed + increment;   }
// . . .
}
```

Η C++ δεν μας δίνει αυτήν ακριβώς τη δυνατότητα αλλά μπορούμε να κάνουμε τα ίδια πράγματα χρησιμοποιώντας αφηρημένες κλάσεις. Για το παράδειγμά μας ορίζουμε:

```
class Bicycle
{
public:
   virtual void changeCadence( int newValue ) = 0;
   virtual void changeGear( int newValue ) = 0;
   virtual void speedUp( int increment ) = 0;
   virtual void applyBrakes( int decrement ) = 0;
};

class ACMEBicycle :  public Bicycle
{
public:
   ACMEBicycle()
   {  cadence = 0;  speed = 0;  gear = 1;  }

   virtual void changeCadence( int newValue )
   {  cadence = newValue;  }

   virtual void changeGear( int newValue )
   {  gear = newValue;  }

   virtual void speedUp( int increment )
   {  speed = speed + increment;  }

   virtual void applyBrakes( int decrement )
   {  speed = speed - decrement;  }

   void printStates()
   {  cout << "cadence:" << cadence << " speed:" << speed
```

[16] Παρόμοια δυνατότητα δίνει και η C#.

[17] `http://docs.oracle.com/javase/tutorial/java/concepts/interface.html`

```
          << " gear:" << gear << endl;  }
private:
   int cadence;
   int speed;
   int gear;
}; // ACMEBicycle
```

23.20 Ανακεφαλαίωση

Από μια *βασική κλάση B* μπορούμε να πάρουμε μια *παράγωγη κλάση D* με τη δήλωση:

```
class D :   public B
{ // . . .
```

Η κλάση *D κληρονομεί* τη *B* με την εξής έννοια:

- Όπου μπορούμε να βάλουμε αντικείμενο κλάσης *B* μπορούμε να βάλουμε αντικείμενο κλάσης *D* (*δυνατότητα υποκατάστασης*) αφού:
  - Κάθε αντικείμενο κλάσης *D* έχει όλα τα μέλη και όλες τις συναρτήσεις-μέλη που έχει ένα αντικείμενο κλάσης *B*.
  - Μπορεί να έχει επιπλέον μέλη ή/και συναρτήσεις-μέλη.
- Η κλάση *D κληρονομεί* στατικά μέλη και στατικές συναρτήσεις-μέλη της *B*.
- Δεν κληρονομούνται οι δημιουργοί, ο τελεστής εκχώρησης και ο καταστροφέας της βασικής κλάσης. Μπορείς όμως να τους χρησιμοποιήσεις για να ορίσεις τις αντίστοιχες συναρτήσεις της παράγωγης.

Στην παράγωγη κλάση μπορούμε να ξαναορίσουμε όποιες μεθόδους της βασικής θέλουμε οπότε αυτές *υπερισχύουν* των αντίστοιχων της βασικής. *Στη βασική κλάση οι μέθοδοι που θα ξαναορισθούν πρέπει να είναι δηλωμένες* "`virtual`".

Ορίζοντας τις συναρτήσεις-μέλη της παράγωγης κλάσης μπορεί να χρειαστεί να διαχειριστείς μέλη της βασικής. Ο καλύτερος τρόπος είναι να το κάνεις μέσω των δημιουργών και των μεθόδων "*get*" και "*set*". Αν θεωρείς απαραίτητη την κατ' ευθείαν πρόσβαση στα μέλη δήλωσέ τα (στη βασική) "`protected`" και όχι "`private`".

Λόγω της *δυνατότητας υποκατάστασης*, σε ορισμένες περιπτώσεις χρειάζεται να βρούμε τον τύπο κάποιου αντικειμένου. Η *δυναμική τυποθεώρηση* και ο τελεστής "`typeid`" δίνουν τη λύση σε αυτό το πρόβλημα.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

23-1 Τι θα γράψει το παρακάτω πρόγραμμα:

```
#include <iostream>
class A
{
public:
   A( char c = 'A' )  :  x( c )
   {   cout << "A=" << x; }
   ~A() {  cout << "~A";  }
   A& operator=( const A& a )
   {
      x = a.x;
      cout << "A=" << x;
   }
private:
   char x;
}; // class A
```

```
class C : public A
{
public:
   C( char c = 'C' )  :  z( c )
   {  cout << "C=" << z;  }
   ~C()
   {  cout << "~C";  }
   C& operator=( const C& c )
   {
      A::operator=(c);
      z = c.z;
      cout << "C=" << z;
   }
private:
```

```
class B : public A
{
public:
   B( char c = 'B' )  :  y( c )
   {  cout << "B=" << y;  }
   ~B()
   {  cout << "~B"; }
   B& operator=( const B& b )
   {
      A::operator=( b );
      y = b.y;
      cout << "B=" << y;
   }
private:
   char y;
}; // class B
```

```
   char z;
}; // class C

int main()
{
   cout << "Prints";
   cout << endl << "1:";
   A* b = new B;
   A* c = new C;
   cout << endl << "2:";
   A bb = *b;
   A cc = *c;
   cout << endl << "3:";
   bb = cc;
} // main
```

Β Ομάδα

23-2 Έχουμε τις εξής δύο κλάσεις:

```
class Order
{
public:
. . .
private:
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
}; // Order
```

```
class Auto
{
public:
. . .
private:
   std::string manuf;
   std::string model;
   char        country[3];
   Date        purchDate;
   std::string regNumber;
}; // Auto
```

όπου *Date* η γνωστή κλάση και:

```
struct Good
{
   unsigned int code;
   std::string  unit;
   double       quantity;
   double       price;
}; // Good
```

α) Η πρώτη (**Order**) περιγράφει μια παραγγελία: Ο δυναμικός πίνακας **goods**, με **gCount** στοιχεία, έχει τα αγαθά που παραγγέλνονται. Όπως φαίνεται και από την **struct Good**, για κάθε είδος, έχουμε τον κωδικό του (*code*), τη μονάδα μέτρησης (*unit*) και την ποσότητα (*quantity*) που παραγγέλνεται. Η αξία (*price*) δεν χρησιμοποποιείται. Στην **oDate** έχουμε την ημερομηνία της παραγγελίας ενώ έχουμε και έναν (μοναδικό) αριθμό παραγγελίας (**orderNum**).

β) Η δεύτερη (**Auto**) περιγράφει ένα αυτοκίνητο: τον κατασκευαστή (**manuf**), το μοντέλο (**model**), τη χώρα κατασκευής (**country**, κωδικός χώρας με δύο γράμματα), ημερομηνία αγοράς (**purchDate**), αριθμό (**regNumber**).[18]

Για κάθε μια από τις δύο κλάσεις, άσχετα από την προβλεπόμενη χρήση, παράθεσε (χωρίς υλοποίηση) τους δημιουργούς, τον καταστροφέα και τις μεθόδους / τελεστές που πρέπει να υλοποιήσουμε. Για κάθε ένα από αυτά θα δικαιολογείς γιατί χρειάζεται ή γιατί δεν χρειάζεται η υλοποίηση.

[18] Προφανώς και για τις δύο κλάσεις μπορούμε να σκεφτούμε και άλλα μέλη, απαραίτητα για διάφορες εφαρμογές. Εδώ, θα απαντήσεις τις ερωτήσεις που ακολουθούν με βάση τα μέλη που δίνονται.

Άσχετα από την απάντησή σου στην προηγούμενη ερώτηση γράψε τους δημιουργούς ερήμην και αντιγραφής και για τις δύο κλάσεις. Σχολίασε αυτά που έγραψες και με βάση τις απαντήσεις που έδωσες στην προηγούμενη ερώτηση.

Για την κλάση **Auto** υλοποίησε τον τελεστή εγγραφής ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**).

Για την κλάση **Order** υλοποίησε μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

23-3 Η κλάση:

```
class Invoice
{
public:
. . .
private:
   unsigned int invoiceNum;
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
   Date         iDate;
   double       total;
}; // Order
```

παριστάνει ένα τιμολόγιο. Έχει τον αριθμό της παραγγελίας στην οποίαν αναφέρεται αλλά και δικό του αριθμό τιμολογίου (**invoiceNum**). Έχει την ημερομηνία της παραγγελίας αλλά και την ημερομηνία έκδοσης του τιμολογίου (**iDate**). Τα αγαθά περιγράφονται με τον ίδιο τρόπο, όπως και στην παραγγελία, αλλά τώρα είναι συμπληρωμένη και η αξία (*price*). Τέλος, υπάρχει και η συνολική αξία (*total*).

Όπως έκανες για την **Order**, υλοποίησε και για την **Invoice** μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

Η **Invoice** μπορεί να θεωρηθεί ως παράγωγη της **Order**. Γράψε τη δήλωσή της ώστε να φαίνεται αυτή σχέση.

Στην **Order** και στις μεθόδους που έγραψες γι' αυτήν κάνε όλες τις αλλαγές που απαιτούνται ώστε η κληρονόμηση να γίνεται σωστά.

23-4 Έχουμε τις κλάσεις:

```
class A
{
public:
. . .
protected:
   double*      g;
   unsigned int nElmn;
   std::string  c;
}; // A
```

```
class B: public A
{
public:
. . .
private:
   double r;
}; // B
```

(η *nElmn* δείχνει το πλήθος στοιχείων του πίνακα g.)

Συμπλήρωσέ τις με ό,τι χρειάζεται ώστε

α) να μπορώ να γράψω τα εξής:

```
int main()
{
   double q[3] = { 3.0, 1.5, -1.1 };
   A a1;
// a1.nElmn==0 && a1.g==0 && c==""
   A a2( 3, q, "an object" );
// a2.nElmn==3 && a2.g[0]==3.0 && a2.g[1]==1.5 && a2.g[2]==-1.1 &&
// a2.c=="an object"
   A a3( a2 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==1.5 && a3.g[2]==-1.1 &&
```

```
// a3.c=="an object"

   a3.setG( 1, 5.5 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==5.5 && a3.g[2]==-1.1 &&
// a3.c=="an object" &&
// a2.g[1]==1.5 && q[1]==1.5

   B b1( a3, 7.35 );
// b1.nElmn==3 && b1.g[0]==3.0 && b1.g[1]==5.5 && b1.g[2]==-1.1 &&
// b1.c=="an object" && b1.r==7.35
   . . .
```

Στα σχόλια που ακολουθούν κάθε εντολή δίνεται η κατάσταση του θα έχουμε μετά την εκτέλεσή της.

β) Ακόμη, αν έχω δηλώσει:

```
   A*      z[2] = { &a2, &b1 };
   fstream tout( "abc.dta", ios_base::out|ios_base::binary );
```

οι εντολές:

```
   z[0]->write( tout );
   z[1]->write( tout );
```

θα πρέπει να έχουν ως αποτέλεσμα την εγγραφή στον αρχείο `abc.dta`, σε εσωτερική παράσταση (binary), των τιμών των *a*2, *b*1 με την εξής σειρά:

```
3, 3.0, 1.5, -1.1, an object, 3, 3.0, 5.5, -1.1, an object, 7.35
```

23-5 Θέλουμε να γράψουμε μια κλάση:

```
class LongBitMap
{
public:
   typedef unsigned long int  ULong;
   LongBitMap()  {  mBm = new ULong[1];  mBm[0] = 0;
                    mNoOfBits = 0;  mArrSize = 1;  }
   LongBitMap( LongBitMap& other );
   ~LongBitMap() {  delete [] ULong;  }
   long getNoOfBits() const {  return mNoOfBits;  }
   LongBitMap& operator=( LongBitMap& other );
   LongBitMap& operator&=( LongBitMap& other );
   LongBitMap& operator|=( LongBitMap& other );
   LongBitMap& operator^=( LongBitMap& other );
   void lSetBit( int pos );
   void lClearBit( int pos );
   ULong lCount1() const;
   int lBitValue( int pos ) const;
private:
   ULong* mBm;          // δυναμικός ψηφιοπίνακας
   ULong  mNoOfBits;    // πλήθος των bits που χρησιμοποιούμε
   ULong  mArrSize;     // πλήθος στοιχείων (τύπου ULong) του mBm
}; // class LongBitMap
```

για τη διαχείριση μεγάλων ψηφιοπινάκων. Όπως βλέπεις, ο ψηφιοπίνακας παριστάνεται με έναν δυναμικό πίνακα με στοιχεία τύπου `unsigned long int`. Στο mBm[0] υπάρχουν τα bits 0 μέχρι 31, στο mBm[1] τα bits 32 μέχρι 63, στο mBm[2] τα bits 64 μέχρι 95 κ.ο.κ. Το μέλος *mNoOfBits* μας δίνει το πλήθος bits που χρησιμοποιούνται. Το μέλος *mArrSize* μας δίνει το πλήθος των στοιχείων του δυναμικού πίνακα mBm.

Εσύ θα πρέπει να βοηθήσεις στην ολοκλήρωση της υλοποίησης. Να οι προδιαγραφές των μεθόδων που πρέπει να γράψεις:

Δημιουργός Αντιγραφής.

Τελεστής ψηφιακού "**and**" (`operator&=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Τελεστής ψηφιακού "**xor**" (`operator^=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Μέθοδος *lSetBit*. Βάζει 1 στο bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 τότε ο πίνακας επεκτείνεται (όπως μάθαμε στη *renew*) ώστε να περιέχει και θέση *pos*.

Μέθοδος *lBitValue*. Επιστρέφει την τιμή του bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 ρίχνει εξαίρεση.

Μπορείς να χρησιμοποιήσεις χωρίς να αντιγράψεις εδώ τα περιγράμματα που συναρτήσεων για ψηφιοχάρτες που υπάρχουν στις σημειώσεις.

project

6

Φοιτητές και Μαθήματα με Κληρονομιές

Περιεχόμενα:

Prj06.1 Το Πρόβλημα

Στην §Prj04.12 παρατηρούσαμε:

- *«Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:*

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
```

- *Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:*

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
```

```
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)»

Το πρόβλημα που έχουμε να λύσουμε είναι αυτό που λύσαμε στο Project 4 με την εξής διαφορά:

Παίρνοντας υπόψη μας τις παραπάνω παρατηρήσεις «σπάζουμε» την κάθε μια από τις παραπάνω κλάσεις σε δύο ως εξής:

```
class Student
{
public:
// . . .
private:
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
}; // Student

class EnrolledStudent :  public Student
{
public:
// . . .
private:
   char               esSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       esWH;           // ώρες ανά εβδομάδα
   unsigned int       esNoOfCourses;  // αριθμός μαθημάτων που
                                      // δήλωσε
   Course::CourseKey* esCourses;
}; // EnrolledStudent
```

και

```
class Course
{
public:
// . . .
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
}; // Course

class OfferedCourse :  public Course
{
public:
// . . .
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

Στον πίνακα μαθημάτων θα κάνουμε το εξής: αν σε κάποιο μάθημα δεν έγγραφεί ούτε ένας φοιτητής στον πίνακα θα εμφανίζεται με αντικείμενο Course ενώ αν έχει γραφεί έστω και ένας φοιτητής θα έχουμε αντικείμενο κλάσης OfferedCourse. Παρομοίως, στο μητρώο φοιτητών ένας φοιτητής που έχει εγγραφεί σε ένα τουλάχιστον μάθημα εμφανίζεται με αντικείμενο EnrolledStudent· αλλιώς εμφανίζεται με αντικείμενο κλάσης Student.

Παρατήρηση: ▶

«*"Σπάζουμε" την κάθε μια από τις παραπάνω κλάσεις σε δύο*»; Όχι δα! *Τα αντικείμενα κλάσης EnrolledStudent έχουν όλο το περιεχόμενο των αντικειμένων της «παλιάς» κλάσης Student.* Παρομοίως, *τα αντικείμενα κλάσης OfferedCourse έχουν όλο το περιεχόμενο των αντικειμένων της «παλιάς» κλάσης Course.* Προσπαθώντας –με τις νέες κλάσεις– να αντιμετωπίσουμε το πρόβλημα που επισημάναμε καταλήξαμε στα ίδια!

Οι *EnrolledStudent* και *OfferedCourse* θα έλυναν τα πρόβλημα αν *δεν κληρονομούσαν* τις *Student* και *Course* (αλλά περιείχαν τα κλειδιά –*sIdNum* και *cCode*– για αντιστοίχιση.) Ξαναδιάβασε την §23.14.

Το παράδειγμα αυτό έχει στόχο να δείξει άλλα πράγματα και όχι τη σωστή σχεδίαση.◀

Prj06.2 Οι Κλάσεις

Να δούμε πρώτα τις *Course* και *OfferedCourse*. Η κληρονομιά δεν είναι λάθος αφού πέρα από την προγραμματιστική τους σχέση και νοηματικώς λέμε:

Ένα *προσφερόμενο μάθημα* είναι ένα *μάθημα*

Να δούμε δύο ερωτήματα που μπορεί να προκύψουν για πολλούς:

- *Μήπως θα έπρεπε να γράψουμε:* `struct Course` *και να αφήσουμε όλα τα μέλη ανοικτά;* Η αναλλοίωτη της κλάσης δεν μας αφήνει περιθώρια για κάτι τέτοιο.
- *Και θα κρατήσουμε το "*`private`*"; Μήπως πρέπει να το κάνουμε "*`protected`*";* Αν και όταν χρειαστεί θα το κάνουμε· προς το παρόν το κρατούμε όπως είναι.

Παρόμοια σκεπτικά ισχύουν και γαι τις *Student* και *EnrolledStudent*.

Να δούμε τώρα τις συλλογές και πρώτα την *CourseCollection*. Πώς θα αποθηκεύσουμε τον «πίνακα μαθημάτων»; Ας εξετάσουμε δύο περιπτώσεις:

- Σε δύο (δυναμικούς) πίνακες: έναν με στοιχεία τύπου *Course* –στον οποίον αρχικώς φορτώνουμε όλα τα στοιχεία που παίρνουμε από το αρχείο– και έναν με στοιχεία τύπου *OfferedCourse*. Κάθε φορά που βρίσκουμε ένα μάθημα που το δήλωσε κάποιος φοιτητής το μεταφέρουμε από τον πρώτο στον δεύτερο.
- Σε έναν (δυναμικό) πίνακα, τον `ccArr`, αλλά με στοιχεία τύπου *Course*`*` και όχι *Course*. Αρχικώς το κάθε `ccArr[k]` δείχνει ένα (δυναμικό) αντικείμενο κλάσης *Course*, που είναι τα στοιχεία ενός μαθήματος όπως φορτώνονται από το αρχείο. Την πρώτη φορά που βρίσκουμε δήλωση ενός μαθήματος από κάποιον φοιτητή μετατρέπουμε το αντίστοιχο `*ccArr[k]` σε αντικείμενο κλάσης *OfferedCourse*.

Η πρώτη λύση είναι μάλλον πιο απλή. Θα προτιμήσουμε όμως τη δεύτερη για διδακτικούς λόγους.

Για τους ίδιους λόγους θα προτιμήσουμε παρόμοια λύση και για τη *StudentCollection*.

Prj06.3 Οι Κλάσεις *Course* και *OfferedCourse*

Στην κλάση *OfferedCourse* δηλώνεται το μέλος *ocNoOfStudents*. Έτσι, θα πρέπει να μεταφέρουμε εκεί τις μεθόδους *clearStudents()*, *add1Student()*, *delete1Student()* που χειρίζονται το μέλος αυτό. Οι μέθοδοι που χειρίζονται τα υπόλοιπα μέλη δηλώνονται στην *Course* (και κληρονομούνται από την *OfferedCourse*.)

Οι μέθοδοι *save()*, *load()* και *display()* έχουν να κάνουν με ολόκληρο το αντικείμενο είτε είναι τύπου *Course* είτε τύπου *OfferedCourse*. Αυτές θα δηλωθούν "`virtual`" στην *Course* και θα ξαναδηλωθουν στην *OfferedCourse*.

Τέλος, κάθε κλάση θα έχει τους δημιουργούς και τον καταστροφέα της. Και ξεκινούμε από αυτούς.

Ο ερήμην δημιουργός της *Course* χρειάζεται μια μόνον αλλαγή: τη διαγραφή της γραμμής

```
cNoOfStudents = 0;
```

ενώ ο καταστροφέας θα γίνει:

```
virtual ~Course() { };
```

Ας δούμε τώρα τα αντίστοιχα για την *OfferedCourse*. Θα γράψουμε τον δημιουργό όπως μάθαμε στην §23.3:

```
OfferedCourse::OfferedCourse( string aCode, string aTitle )
   : Course( aCode, aTitle ), ocNoOfStudents( 0 )  { }
```

Όπως βλέπεις, τα πάντα γίνονται με τη λίστα εκκίνησης και το σώμα της συνάρτησης είναι κενό.

Ο καταστροφέας είναι τετριμμένος:

```
virtual ~OfferedCourse() { };
```

Να έλθουμε τώρα στις τρεις μεθόδους. Αυτές δηλώνονται στην *Course* ως εξής:

```
virtual void save( ostream& bout ) const;
virtual void load( istream& bin );
virtual void display( ostream& tout ) const;
```

Στους ορισμούς τους θα γίνουν οι εξής αλλαγές:

- Στη *save(0* διαγράφουμε την εντολή:

```
bout.write( reinterpret_cast<const char*>(&cNoOfStudents),
            sizeof(cNoOfStudents) );
```

- Στη *load()* διαγράφουμε την εντολή:

```
bin.read( reinterpret_cast<char*>(&tmp.cNoOfStudents),
          sizeof(cNoOfStudents) ); // αριθ. φοιτ. στο μάθημα
```

- Στη *display()* διαγράφουμε το: "`<< '\t' << cNoOfStudents`"

Στην *OfferedCourse* κάνουμε την ίδια δήλωση, παρ' όλο που τα "`virtual`" δεν είναι απαραίτητα. Οι ορισμοί θα είναι:

```
void OfferedCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw CourseXptn( CourseKey(getCode()), "save",
                        CourseXptn::fileNotOpen );
   this->Course::save( bout );
   bout.write( reinterpret_cast<const char*>(&ocNoOfStudents),
               sizeof(ocNoOfStudents) ); // αριθ. φοιτ. στο μάθημα
   if ( bout.fail() )
      throw CourseXptn( CourseKey(getCode()), "save",
                        CourseXptn::cannotWrite );
} // OfferedCourse::save
```

Όπως βλέπεις, με τη "`this->Course::save(bout)`" φυλάγουμε το υποαντικείμενο κλάσης *Course* και στη συνέχεια φυλάγουμε και την τιμή του μέλους *ocNoOfStudents*.

```
void OfferedCourse::load( istream& bin )
{
   OfferedCourse tmp;
   tmp.Course::load( bin );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&tmp.ocNoOfStudents),
                sizeof(ocNoOfStudents) );// αριθ. φοιτ. στο μάθημα
```

```
      if ( bin.fail() )
         throw CourseXptn( CourseKey(getCode()), "load",
                           CourseXptn::cannotRead);
   } // if ( !bin.eof() ). . .
   *this = tmp;
} // OfferedCourse::load
```

Και εδώ, με την "`tmp.Course::load(bin)`" φορτώνουμε το υποαντικείμενο κλάσης *Course* του *tmp*. Στη συνέχεια φορτώνουμε την τιμή του μέλους *tmp*.*ocNoOfStudents*.

```
void OfferedCourse::display( ostream& tout ) const
{
   this->Course::display( tout );
   tout << ocNoOfStudents << endl;
   if ( tout.fail() )
      throw CourseXptn( CourseKey(getCode()), "display",
                        CourseXptn::cannotWrite );
} // OfferedCourse::display
```

Σε όλες τις ρίψεις εξαιρέσεων πρόσεξε τα εξής:

- Συνεχίζουμε να ρίχνουμε εξαιρέσεις *CourseXptn* αφού τα προβλήματα που βρίσκουμε τα έχουμε προβλέψει και για τη βασική κλάση και δεν παραπλανούμε αυτόν που θα συλλάβει την εξαίρεση. Στη συνέχεια πάντως θα ορίσουμε και μια *OfferedCourseXptn*.
- Αντικαταστήσαμε το "`cCode`" με το "`CourseKey(getCode())`". Αυτό είναι απαραίτητο διότι τα (`private`) μέλη της βασικής δεν είναι ορατά από την παράγωγη. Να το βάλουμε "`protected`"; Όχι αφού το χρησιμοποιούμε μόνο στη ρίψη εξαίρεσης.

Οι τρεις μέθοδοι διαχείρισης του μέλους *ocNoOfStudents* είναι: οι δύο **inline**

```
   void clearStudents() {  ocNoOfStudents = 0;  }
   void add1Student() {  ++ocNoOfStudents;  }
```

και η:

```
void OfferedCourse::delete1Student()
{
   if ( ocNoOfStudents <= 0 )
      throw OfferedCourseXptn( CourseKey(getCode()),"delete1Student",
                               OfferedCourseXptn::noStudent );
   --ocNoOfStudents;
} // OfferedCourse::delete1Student
```

Εδώ, όπως βλέπεις, χρησιμοποιούμε άλλη κλάση εξαιρέσεων διότι το πρόβλημα που εμφανίζεται δεν έχει οποιαδήποτε σχέση με τη βασική κλάση (όπως τα προβλήματα με τα ρεύματα.)

Όπως είπαμε (§23.7), «*η κλάση εξαιρέσεων της παράγωγης κλάσης είναι παράγωγη της κλάσης εξαιρέσεων της βασικής κλάσης.*» Έχουμε λοιπόν:

```
struct OfferedCourseXptn :  public CourseXptn
{
   enum { noStudent=20 };
   OfferedCourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
                      const char* sv="" )
      : CourseXptn( obk, mn, 0, sv ) {  errorCode = ec; }
}; // OfferedCourseXptn
```

Στην περίπτωσή μας, όπως βλέπεις, η παράγωγη κλάση διαφέρει από τη βασική μόνον στον ορισμό της σταθεράς *noStudent*. Γιατί βάλαμε εκείνο το "`=20`"; Διότι αλλιώς το *OfferedCourseXptn*`::`*noStudent* θα είχε τιμή "`0`" όπως ακριβώς και το *OfferedCourseXptn*`::`*keyLen* (κληρονομιά!)

Στο σώμα του δημιουργού το μόνο που κάνουμε είναι να δώσουμε τον σωστό κωδικό λάθους.

Prj06.3.1.1 Μετατροπες από *Course* σε *OfferedCourse*

Πρόσεξε τώρα ένα πρόβλημα που θα αντιμετωπίσουμε: Όπως λέγαμε «*Την πρώτη φορά που βρίσκουμε δήλωση ενός μαθήματος από κάποιον φοιτητή μετατρέπουμε το αντίστοιχο* `*ccArr[k]` *σε αντικείμενο κλάσης OfferedCourse*.» Πώς θα γίνεται αυτό; Ας πούμε ότι έχουμε δηλώσει:

```
   OfferedCourse* pOneCourse( new OfferedCourse );
```

Στο αντικείμενο που δείχνει το *pOneCourse* θα πρέπει να αντιγράψουμε το περιεχόμενο του `*ccArr[ndx]`, να μηδενίσουμε το *ocNoOfStudents* και μετά να βάλουμε:

```
      delete ccArr[ndx];
      ccArr[ndx] = pOneCourse;
```

Πώς μπορούμε να αντιγράψουμε το περιεχόμενο;

- Γράφοντας τον κατάλληλο δημιουργό μετατροπής» που από ένα αντικείμενο της κλάσης *Course* θα δημιουργεί αντικείμενο κλάσης *OfferedCourse* με τιμή "`0`" στο *ocNoOfStudents*. Έτσι θα μπορούμε να γράψουμε:

```
      OfferedCourse* pOneCourse( new OfferedCourse(*ccArr[ndx]) );
```

- Επιφορτώνοντας τον τελεστή εκχώρησης ώστε να κάνουμε νόμιμη την εντολή:

```
   *pOneCourse = *ccArr[ndx];
```

Προφανώς η πρώτη επιλογή είναι ταχύτερη αφού δημιουργεί κατ' ευθείαν το αντικείμενο που μας ενδιαφέρει. Δηλώνουμε λοιπόν:

```
   OfferedCourse( const Course& oneCourse );
```

και ορίζουμε, όπως κάναμε και στον ερήμην δημιουργό:

```
OfferedCourse::OfferedCourse( const Course& oneCourse )
   : Course( oneCourse ), ocNoOfStudents( 0 )  { }
```

Παρατηρήσεις: ▶

1. Δεν επιφορτώνουμε και τον τελεστή εκχώρησης για να δοκιμάσουμε και την άλλη επιλογή; Δεν χρειάζεται! Αφού έχουμε σωστό τελεστή εκχώρησης και τον δημιουργό "`OfferedCourse( const Course& oneCourse )`" έχουμε «νομιμοποιήσει» και την εκώρηση "`*pOneCourse = *ccArr[ndx]`".
2. Δεν θα χρειαστούμε και έναν δημιουργό

```
   Course( const OfferedCourse& oneOfrdCourse );
```

για να κάνουμε την αντίστροφη μετατροπή; Όχι· αυτό γίνεται –με τεμαχισμό φυσικά– από τον (συναγόμενο) δημιουργό αντιγραφής της *Course*.◀

Prj06.3.1.2 Οι Ορισμοί των Κλάσεων

Τώρα η *Course* θα είναι:[1]

```
class Course
{
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strncpy( s, aKey.c_str(), cCodeSz-1 );
         s[cCodeSz-1] = '\0';  }
   }; // CourseKey
   explicit Course( string aCode="", string aTitle="" );
   virtual ~Course() { };
// getters
```

[1] Αυτή δεν είναι άλλη από τη *SylCourse* που λέγαμε στο Project 2· φυσικά είναι πιο καλογραμμένη (πέρα από την προετοιμασία για την κληρονομιά.)

```
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   const char* getPrereq() const {  return cPrereq.s;  }
// setters
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) {  cCompuls = aCompuls;  };
   void setSector( char aSector );
   void setCateg( string aCateg );
   void setWH( int aWH );
   void setUnits( int aUnits );
   void setPrereq( const string& prCode );
// other
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout ) const;
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
}; // Course

typedef Course*  PCourse;
```

Οι επιφορτώσεις των "`!=`" και "`==`" για τους τύπους *CourseKey* και *Course* δεν αλλάζουν.

Η κλάση εξαιρέσεων είναι:

```
struct CourseXptn
{
   enum { keyLen, rangeError, noSuchSector, noSuchCateg, autoRef,
          fileNotOpen, cannotWrite, cannotRead };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
      : objKey( obk ),  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // CourseXptn
```

Η μόνη διαφορά από την προηγούμενη μορφή: έχει αφαιρεθεί ο ορισμός της σταθεράς *noStudent*.

Η παράγωγη κλάση:

```
class OfferedCourse :  public Course
{
public:
   explicit OfferedCourse( string aCode="", string aTitle="" );
```

```
   OfferedCourse( const Course& oneCourse );
   virtual ~OfferedCourse() { };
// getters
   unsigned int getNoOfStudents() const { return ocNoOfStudents; }
// setters
   void clearStudents() {  ocNoOfStudents = 0;  }
   void add1Student() {  ++ocNoOfStudents;  }
   void delete1Student();
// other
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout ) const;
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse

typedef OfferedCourse*  POfferedCourse;
```

Πρόσεξε ότι δεν μας χρειάζονται επιφορτώσεις για συγκρίσεις αντικειμένων κλάσης *OfferedCourse*: Μας αρκούν αυτές που έχουμε για την κλάση *Course*.

Τώρα βέβαια, το «*δεν μας χρειάζονται*» είναι «μεγάλη κουβέντα» διότι με τις επιφορτώσεις των τελεστών της βασικής κλάσης μπορούμε να συγκρίνουμε και αντικείμενα των δύο κλάσεων και να βρούμε ότι ένα αντικείμενο κλάσης *Course* είναι ίσο με ένα αντικείμενο κλάσης *OfferedCourse* επειδή έχουν τον ίδιο κωδικό! Είναι σωστό αυτό; Ας μην ανοίξουμε τώρα φιλοσοφική συζήτηση πάνω στο θέμα·[2] να πούμε μόνο το εξής: Αυτές οι συγκρίσεις μας χρειάζονται για την αναζήτηση μέσα στη συλλογή. Για τη δουλειά αυτή είναι σωστές!

Η κλάση εξαιρέσεων είναι αυτή που είδαμε και πιο πάνω:

```
struct OfferedCourseXptn :  public CourseXptn
{
   enum { noStudent=20 };
   OfferedCourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
                      const char* sv="" )
      : CourseXptn( obk, mn, 0, sv ) {  errorCode = ec; }
}; // OfferedCourseXptn
```

Prj06.4 Η Κλάση *CourseCollection*

Η βασική αλλαγή στην κλάση *CourseCollection* είναι στη δήλωση του δυναμικού πίνακα που από

```
   Course*   ccArr;
```

γίνεται πίνακας βελών:

```
   PCourse*  ccArr;
```

Επειδή, όπως πρέπει ήδη να κατάλαβες, η σπουδαιότερη μέθοδος στις κλάσεις-συλλογές είναι η *findNdx()*, θα πρέπει να ξεκινήσουμε από αυτήν. Η μεθοδος βασίζεται στη *linSearch()* από τη βιβλιοθήκη περιγραμμάτων `MyTmpltLib`. Η *linSearch()* δεν μπορεί να δουλέψει για τον νέο πίνακα διότι δεν μπορούμε να επιφορτώσουμε τελεστές για τύπους βελών. Το ίδιο πρόβλημα θα έχουμε και στη *findNdx()* της *StudentCollection*. Τι μπορούμε να κάνουμε για να λύσουμε αυτό το πρόβλημα;

- Να γράψουμε μέσα στις *findNdx()* τις εντολές αναζήτησης και να μη χρειαζόμαστε τη *linSearch()*.
- Να «κρύψουμε» σε μια κλάση το βέλος ώστε να μπορούμε να επιφορτώσουμε τους τελεστές σύγκρισης για την κλάση αυτή.
- Να γράψουμε μια νέα εκδοχή της *linSearch()* –ας την πούμε *linSearchP*– για πίνακες βελών.

[2] Πάντως δεν είναι κακό να το σκεφτείς λιγάκι και να το συζητήσεις με συναδέλφους σου...

Η τελευταία επιλογή φαίνεται πιο κομψή (και πιο απλή) από τις άλλες. Γράφουμε λοιπόν:

```
template< typename T >
int linSearchP( T** v[], int n,
                int from, int upto, const T& x )
{
   if ( v == 0 && n > 0 )
      throw MyTmpltLibXptn( "linSearchP",
                            MyTmpltLibXptn::noArray );
   int fv( -1 );
   if ( v != 0 && (0 <= from && from <= upto && upto < n) )
   {
      T  save( *v[upto+1] );  // φύλαξε το *v[upto+1]
      *v[upto+1] = x;         // φρουρός
      int k( from );
      while ( *v[k] != x )  ++k;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
      *v[upto+1] = save;      // όπως ήταν στην αρχή
   // (from <= fv <= upto && v[fv] == x) ||
   //                     (fv == -1 && (∀j:from..upto • v[j] != x))
   }
   return fv;
} // linSearchP
```

Όπως βλέπεις, αυτό που συγκρίνεται με τον "`!=`" είναι το "`*v[k]`".

Βλέπεις όμως και κάτι άλλο: πρέπει να υπάρχει το "`*v[upto+1]`". Αν ψάχνουμε σε ολόκληρον τον πίνακα αυτό σημαίνει ότι θα πρέπει να υπάρχει το "`*v[n]`" (το στοιχείο που κρατούμε για φρουρό στις αναζητήσεις).

Τώρα, μπορούμε να γράψουμε την

```
int CourseCollection::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearchP( ccArr, ccNOfCourses,
                        0, ccNOfCourses-1, Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

που, όπως βλέπεις, δεν διαφέρει από την αρχική, παρά μόνο στο όνομα της συνάρτησης αναζήτησης.

Θα συνεχίσουμε βλέποντας τις επιπτώσεις των αλλαγών που κάναμε στον ερήμην δημιουργό:

```
CourseCollection::CourseCollection()
{
   try
   {
      ccReserved = ccIncr;
      ccArr = new PCourse[ ccReserved ];
      ccNOfCourses = 0;
      ccPAllEnrollments = 0; // NULL
      ccArr[ccNOfCourses] = new Course;
      for ( int k(ccNOfCourses+1); k < ccReserved; ++k )
         ccArr[k] = 0;
   }
   catch( bad_alloc& )
   {
      throw CourseCollectionXptn( "CourseCollection",
                                  CourseCollectionXptn::allocFailed);
   }
} // CourseCollection::CourseCollection
```

Πρόσεξε την εντολή "`ccArr[ccNOfCourses] = new Course`" (το στοιχείο για τον φρουρό). Η **for** που ακολουθεί διασφαλίζει το ότι τα βέλη που δεν χρησιμοποιούνται έχουν τιμή "`0`".

Φυσικά αλλάζει και ο καταστροφέας:

```
CourseCollection::~CourseCollection()
{
   for ( int k(0); k <= ccNOfCourses; ++k )
      delete ccArr[k];
   delete[] ccArr;
} // CourseCollection::~CourseCollection
```

Παρόμοιες επιπτώσεις έχουμε και στις μεθόδους (**private**) «χαμηλού επιπέδου» *erase1Course()* και *insert1Course()*.

```
void CourseCollection::erase1Course( int ndx )
{
   delete ccArr[ndx];
   ccArr[ndx] = ccArr[ccNOfCourses-1];
   ccArr[ccNOfCourses-1] = ccArr[ccNOfCourses];
   ccArr[ccNOfCourses] = 0;
   --ccNOfCourses;
} // CourseCollection::erase1Course
```

Η "`ccArr[ndx] = ccArr[ccNOfCourses-1]`" δεν αρκεί πια:

- Πριν από αυτήν πρέπει να ανακυκλώσουμε το "`*ccArr[ndx]`".
- Μετά από αυτήν πρέπει να βάλουμε τον φρουρό στη σωστή θέση. Αυτό γίνεται με την "`ccArr[ccNOfCourses-1] = ccArr[ccNOfCourses]`".

Με την *insert1Course()* έχουμε πρόβλημα· δες την επικεφαλίδα της:

```
void CourseCollection::insert1Course( const Course& aCourse );
```

Αυτή μας επιτρέπει να εισαγάγουμε στη συλλογή ένα αντικείμενο *Course* αλλά και ένα αντικείμενο *OfferedCourse*. Στη πρώτη περίπτωση θα δώσουμε

```
ccArr[ccNOfCourses] = new Course( aCourse );
```

ενώ στη δεύτερη

```
ccArr[ccNOfCourses] = new OfferedCourse( aCourse );
```

Τι θα πρέπει να κάνουμε;

- Μια λύση είναι η *τυποθεώρηση της παραμέτρου αναφοράς* (§23.13.1). Αυτή έχει –στην περίπτωσή μας– το εξής πλεονέκτημα: Δεν θα χρειαστεί να αλλάξουμε την *add1Course()*.
- Μια άλλη λύση είναι η εξής: αλλάζουμε την παράμετρο στην επικεφαλίδα:

```
void CourseCollection::insert1Course( const Course* pCourse )
```

Με δυναμική τυποθεώρηση μπορούμε να βρούμε τον τύπο του αντικειμενου που δείχνει το *pCourse*. Μειονέκτημα: μπορεί να κληθεί με όρισμα τη διεύθυνση μη-δυναμικού αντικειμένου. Αυτό όμως μπορεί να ελεγχθεί αφού η *insert1Course()* είναι **private** και καλείται μόνον από τις (*CourseCollection*`::`) *add1Course()* και (*CourseCollection*`::`)*load()*. Αυτό που φαίνεται να είναι πλεονέκτημα είναι η απλότητά της· αλλά η αλήθεια είναι ότι η τυποθεώρηση της παραμέτρου αναφοράς μεταφέρεται στην *add1Course()*.

Προτιμούμε λοιπόν τη δεύτερη λύση.

```
void CourseCollection::insert1Course( Course* pCourse )
{
   if ( ccReserved <= ccNOfCourses+1 )
   {
      try
      {
         renew( ccArr, ccNOfCourses+1, ccReserved+ccIncr );
         ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses];  // αλλαγή φρουρού
         for ( int k(ccNOfCourses+2); k < ccReserved+ccIncr; ++k )
            ccArr[k] = 0;
```

```
         ccReserved += ccIncr;
      }
      catch( ... )
      {  throw CourseCollectionXptn( "insert1Course",
                                     CourseCollectionXptn::allocFailed );  }
   }
   ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses];
   ccArr[ccNOfCourses] = pCourse;
   ++ccNOfCourses;
} // CourseCollection::insert1Course
```

Όσο για τις άλλες αλλαγές:

- Κρατούμε την πρώτη από τις «νέες θέσεις» (*ccNOfCourses*+1) στον πίνακα για τον φρουρό ("`ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses]`"). Στις υπόλοιπες βάζουμε "`0`".
- Εισάγουμε τη νέα τιμή στον «παλιό φρουρό» ("`ccArr[ccNOfCourses] = pCourse`").
- Κάνουμε (με αρκετή αυθαιρεσία) την υπόθεση: αν κάτι δεν πάει καλά θα πει ότι έχουμε πρόβλημα δυναμικής μνήμης. Έτσι όποιουδήποτε τύπου ("`catch(...)`") εξαίρεση και να πιάσουμε εμείς ρίχνουμε *allocFailed*!

Στις αντίστοιχες μεθόδους (**public**) «υψηλού επιπέδου» η *delete1Course()* θα γίνει:

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // υπάρχει
   {
      *ccArr[ccNOfCourses] = Course();
      ccArr[ccNOfCourses]->setPrereq( code ); // φρουρός
      int k(0);
      while ( strcmp(ccArr[k]->getPrereq(), code.c_str()) != 0 )
         ++k;
      if ( k < ccNOfCourses )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::prereqRef,
                                     code.c_str() );
      OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
      if ( pOneCourse != 0 )
      {
         int enrStdnt( pOneCourse->getNoOfStudents() );
         if ( enrStdnt > 0 ) // υπάρχουν εγγεγραμμένοι φοιτητές
            throw CourseCollectionXptn( "delete1Course",
                                        CourseCollectionXptn::enrollRef,
                                        code.c_str(), enrStdnt );
      }
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Πέρα από τις αλλαγές από "`.`" σε "`->`" πρόσεξε αυτά που κάνουμε στο τέλος της: Με δυναμική τυποθεώρηση ελέγχουμε αν το `*ccArr[ndx]` είναι τύπου *OfferedCourse*.

- Αν δεν είναι (*pOneCourse* `==` 0) προχωρούμε κατ' ευθείαν στην "`erase1Course(ndx)`".
- Αν είναι (*pOneCourse* `!=` 0) κάνουμε επιπλέον τον έλεγχο

  pOneCourse`->`*getNoOfStudents()* `>` 0

  Προχωρούμε στην "`erase1Course(ndx)`" μόνον αν δεν ισχύει αυτή η συνθήκη.

Η επικεφαλίδα της *add1Course()* είναι:

```
void CourseCollection::add1Course( const Course& aCourse )
```

Γιατί να μην αλλάξουμε την παράμετρο σε "`const Course* pCourse`" και να κάνουμε τη ζωή μας πιο εύκολη; Διότι –αφού είναι **public**– μπορεί να υπάρχουν εφαρμογές που τη χρησιμοποιούν ήδη και θα πρέπει να πάμε να κάνουμε αλλαγές και στις εφαρμογές αυτές.

Όπως καταλαβαίνεις, το πρόβλημα που είδαμε –και παρακάμψαμε– παραπάνω ξαναεμφανίζεται και πρέπει να λυθεί τώρα.

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )    // υπάρχει προαπαιτούμενο
      {
         int ndx( findNdx(aCourse.getPrereq()) );
         if ( ndx < 0 )           // δεν βρέθηκε το κλειδί του προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::prereqRef,
                                        aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      try
      {
         try
         {
            const OfferedCourse&
                     oneCourse( dynamic_cast<const OfferedCourse&>(aCourse) );
            insert1Course( new OfferedCourse(oneCourse) );
         }
         catch( bad_cast )
         {  insert1Course( new Course(aCourse) );  }
      }
      catch( bad_alloc )
      {  throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::allocFailed );  }
   }
} // CourseCollection::add1Course
```

Πρόσεξε τι κάνουμε: αφού σιγουρευτούμε ότι πρέπει να κάνουμε την εισαγωγή δηλώνουμε

```
            const OfferedCourse&
                     oneCourse( dynamic_cast<const OfferedCourse&>(aCourse) );
```

Αν η πραγματική παράμετρος που έρχεται στην *aCourse* είναι κλάσης *OfferedCourse* τότε η τιμή της θα αντιγραφεί ως αρχική τιμή της *oneCourse*. Αλλιώς, αν είναι κλάσης *Course*, ρίχνεται εξαίρεση *bad_cast*. Αυτά γίνονται στην εσωτερική **try/catch** που, όπως βλέπεις, δεν χειρίζεται κάποια «εξαιρετική» κατάσταση αλλά λειτουργεί σαν **ifelse**. Αυτή είναι μια ανορθόδοξη χρήση του μηχανισμού εξαιρέσεων.

Σε κάθε περίπτωση, θα εκτελεσθεί μια **new**· η εξωτερική **try/catch** υπάρχει για τυχόν απότυχία της.

Φυσικά, η *clearStudents()* καλείται μόνον στην περίπτωση που το αντικείμενο που εισάγεται είναι κλάσης *OfferedCourse*.

Ενδιαφέρον έχουν οι μέθοδοι πρόσθεσης φοιτητή σε μάθημα και διαγραφής φοιτητή από μάθημα· *σε αυτές θα έχουμε και τις αλλαγές τύπου*.

```
void CourseCollection::add1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
   if ( pOneCourse == 0 )
   {
      try {  pOneCourse  = new OfferedCourse( *ccArr[ndx] );  }
```

```
      catch( bad_alloc& )
      {
         throw CourseCollectionXptn( "add1Student",
                                     CourseCollectionXptn::allocFailed,
                                     code.c_str() );
      }
      delete ccArr[ndx];
      ccArr[ndx] = pOneCourse;
   }
   pOneCourse->add1Student();
} // CourseCollection::add1Student
```

Κατ' αρχάς να τονίσουμε ότι δεν μπορείς να γράψεις "`ccArr[ndx]->add1Student()`": ο μεταγλωττιστής θα σου πει ότι η κλάση *Course* δεν έχει μέθοδο *add1Student()*. Έτσι, παίρνουμε ένα νέο βέλος, το *pOneCourse*, που δείχνει το ίδιο αντικείμενο με το `ccArr[ndx]`. Αν όμως το `*ccArr[ndx]` είναι τύπου *Course* η μετατροπή θα αποτύχει. Στην περίπτωση αυτήν κάνουμε τη μετατροπή όπως είπαμε στην §Prj06.3.1. Σε κάθε περίπτωση, όταν έρχεται η ώρα να εκτελεσθεί η "`pOneCourse->add1Student()`" το *pOneCourse* δείχνει το ίδιο αντικείμενο (τύπου *OfferedCourse*) με το `ccArr[ndx]`.

```
void CourseCollection::delete1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
   pOneCourse->delete1Student();
// τα παρακάτω μπορεί και να περισσεύουν . . .
   if ( pOneCourse->getNoOfStudents() == 0 )
   {
      Course* tmp( new Course(*pOneCourse) );
      delete pOneCourse;
      ccArr[ndx] = tmp;
   }
} // CourseCollection::delete1Student
```

Τι καινούριο υπάρχει εδώ; Αν μετά τη διαγραφή ενός φοιτητή μηδενίζεται το πλήθος των φοιτητών που θέλουν το μάθημα μετατρέπουμε το αντικείμενο που δείχνει το `ccArr[ndx]` από *OfferedCourse* σε *Course*.

Και είναι απαραίτητη αυτή η μετατροπή; Μετά από λίγο μπορεί να έλθει απαίτηση άλλου φοιτητή να εγγραφεί στο μάθημα. Σωστό! Το τι θα κάνουμε εξαρτάται από την εφαρμογή. Αν έχεις ένα πρόγραμμα που θέλει γρήγορες αποκρίσεις

- Δεν κάνεις αυτήν τη μετατροπή.
- Πριν από τη φύλαξη των στοιχείων της συλλογής σαρώνεις όλα τα μαθήματα και όπου βρίσκεις *OfferedCourse* με *ocNoOfStudents* `==` 0 το αλλάζεις σε *Course*.

Στην εφαρμογή του παραδείγματος «δεν μας κυνηγάει κανείς»…

Ας έλθουμε τώρα στις *save()* και *load()*. Το πρόβλημά μας είναι ότι στο ίδιο μη-μορφοποιημένο αρχείο θα έχουμε αντικείμενα δύο διαφορετικών κλάσεων: *Course* και *OfferedCourse*. Πώς το λύνουμε; Με ευρετήριο! Αλλά τώρα, εκτός από το κλειδί και τη θέση, θα έχουμε ένδειξη και για την κλάση του κάθε αντικειμένου. Για παράδειγμα:

```
struct CIndexEntry
{
   Course::CourseKey cCode;
   size_t            loc;
   short int         inhLvl;
}; // IndexEntry
```

που σημαίνει: το αντικείμενο με κλειδί (κωδικό) *cCode* αρχίζει από τη θέση *loc* του αρχείου και είναι κλάσης *Course* αν το *inhLvl* έχει τιμή "`0`" ή κλάσης *OfferedCourse* αν έχει τιμή "`1`".

Για τις αναζητήσεις θα χρειαστούμε έναν δημιουργό –που να δημιουργεί αντικείμενο από τον κωδικό– και επιφόρτωση του "!=":

```
struct CIndexEntry
{
   Course::CourseKey cCode;
   size_t            loc;
   short int         inhLvl;
   explicit CIndexEntry( Course::CourseKey aCode=Course::CourseKey(""),
                         int aLoc=0, short int aInhL=0 )
      :  cCode( aCode ),  loc( aLoc ),  inhLvl( aInhL )  {  }
}; // CIndexEntry

bool operator!=( const CIndexEntry& lhs, const CIndexEntry& rhs )
{  return ( lhs.cCode != rhs.cCode );  }

bool operator==( const CIndexEntry& lhs, const CIndexEntry& rhs )
{  return !(lhs != rhs);  }
```

Έτσι λοιπόν γράφουμε:

```
void CourseCollection::save( ofstream& bout, CIndexEntry* crsIndex ) const
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( int k(0); k < ccNOfCourses; ++k )
   {
      crsIndex[k].cCode = Course::CourseKey(ccArr[k]->getCode());
      crsIndex[k].loc = bout.tellp();
      OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[k]) );
      if ( pOneCourse == 0 )  // Course
      {  crsIndex[k].inhLvl = 0;
         ccArr[k]->save( bout );  }
      else                    // OfferedCourse
      {  crsIndex[k].inhLvl = 1;
         pOneCourse->save( bout );  }
   }
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::cannotWrite );
} // CourseCollection::save
```

Πρόσεξε ότι και εδώ η διαχείριση της δυναμικής μνήμης όπου υλοποιείται το ευρετήριο θα πρέπει να γίνεται από τη συνάρτηση που καλεί τη *save()*.

Δες τώρα πώς χρησιμοποιούμε το ευρετήριο κατά τη φόρτωση:

```
void CourseCollection::load( ifstream& bin, const CIndexEntry* crsIndex )
{
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(n) );
   if ( !bin.eof() )
   {
      CourseCollection tmp;
      try
      {
         for ( int k(0); k < n && !bin.fail(); ++k )
         {
            Course* pOneCourse;
            if ( crsIndex[k].inhLvl == 0 )  // Course
               pOneCourse = new Course;
            else                            // OfferedCourse
               pOneCourse = new OfferedCourse;
            pOneCourse->load( bin );
            tmp.insert1Course( pOneCourse );
         }
      }
```

```
      catch( bad_alloc& )
      {
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::allocFailed );
      }
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Prj06.5 Οι Κλάσεις *Student* και *EnrolledStudent*

Από την κλάση *Student* αφαιρούνται (και δηλώνονται στην *EnrolledStudent*) τα μέλη που έχουν σχέση με τον δυναμικό πίνακα κωδικών (*sNoOfCourses*, *sCourses*, *sReserved*) και το *sWH* (σύνολο εβδομαδιαίων ωρών διδασκαλίας που προκύπτει από τις δηλώσεις μαθημάτων). Φυσικά, ακολουθούνται από τις μεθόδους χειρισμού αυτών των μελών, δηλαδή: *getNoOfCourses()*, *getCourses()*, *clearCourses()*, *find1Course()*, *add1Course()*, *delete1Course()*, *findNdx()*, *insert1Course()*, *erase1Course()* και *getWH()*.

Τώρα, όλα απλουστεύονται. Ο ερήμην δημιουργός θα είναι:

```
Student::Student( int aIdNum )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
}; // Student()
```

ενώ δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής. Παρομοίως, δεν χρειάζεται να γράψουμε επιφόρτωση του τελεστή εκχώρησης. Θα γράψουμε όμως έναν (κενό) καταστροφέα:

```
   virtual ~Student() { };
```

Οι παρακάτω μέθοδοι θα υπάρχουν και στην παράγωγη κλάση και για τον λόγο αυτό θα πρέπει να δηλωθούν:

```
   virtual void swap( Student& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
```

και θα είναι αναλόγως απλουστευμένες:

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );
} // Student::swap

void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
```

```
   bout.write( sFirstname, sizeof(sFirstname) );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save

void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load

void Student::display( ostream& tout )
{
   tout << sIdNum << '\t' << sSurname << '\t' << sFirstname << endl
} // Student::display
```

Έτσι, η (βασική) κλάση *Student* θα είναι:

```
class Student
{
public:
// constructors, destructor
   explicit Student( int aIdNum=0 );
   virtual ~Student() { };
// getters
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
// setters
   void setIdNum( int aIdNum );
   void setSurname( string aSurname );
   void setFirstname( string aFirstname );
// other methods
   void readPartFromText( istream& tin );
   virtual void swap( Student& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
private:
   enum { sNameSz = 20 };
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];

   unsigned int countTabs( string aLine );
}; // Student

typedef Student* PStudent;
```

Η επιφόρτωση των "`!=`" και "`==`" δεν αλλάζει.

Η κλάση εξαιρέσεων απλουστεύεται κάπως:

```
struct StudentXptn
{
   enum { negIdNum, incomplete, fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, const char* sv="" )
```

```
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentXptn( int obk, const char* mn, int ec, int ev )
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Τα πιο πολύπλοκα κομμάτια των δημιουργών μεταφέρονται στην παράγωγη κλάση *EnrolledStudent*. Ας αρχίσουμε από τον ερήμην δημιουργό που δηλώνεται ως:

```
   explicit EnrolledStudent( int aIdNum=0 );
```

και ορίζεται:

```
EnrolledStudent::EnrolledStudent( int aIdNum )
   :  Student( aIdNum ),  esWH( 0 )
{
   try  {  esCourses = new Course::CourseKey[ esIncr ];  }
   catch( bad_alloc )
   {  throw StudentXptn( getIdNum(), "EnrolledStudent",
                         EnrolledStudentXptn::allocFailed );  }
   esReserved = esIncr;
   esNoOfCourses = 0;
} // EnrolledStudent::EnrolledStudent
```

Φυσικά, τώρα χρειαζόμαστε και δημιουργό αντιγραφής και επιφόρτωση του αντιγραφικού τελεστή εκχώρησης. Ο δημιουργός αντιγραφής δηλώνεται:

```
   EnrolledStudent( const EnrolledStudent& rhs );
```

και ορίζεται:

```
EnrolledStudent::EnrolledStudent( const EnrolledStudent& rhs )
   :  Student( rhs ),  esWH( rhs.esWH )
{
   try  {  esCourses = new Course::CourseKey[ rhs.esReserved ];  }
   catch( bad_alloc )
   {  throw EnrolledStudentXptn( getIdNum(), "Student",
                                 EnrolledStudentXptn::allocFailed );  }
   esReserved = rhs.esReserved;
   for ( int k(0); k < rhs.esNoOfCourses; ++k )
      esCourses[k] = rhs.esCourses[k];
   esNoOfCourses = rhs.esNoOfCourses;
}; // EnrolledStudent::EnrolledStudent
```

Για να επιφορτώσουμε τον τελεστή εκχώρησης όπως ξέρουμε θα χρειαστούμε τη *swap()* που τη δηλώνουμε:

```
   virtual void swap( EnrolledStudent& rhs );
```

και την ορίζουμε:

```
void EnrolledStudent::swap( EnrolledStudent& rhs )
{
   Student::swap( rhs );

   std::swap( esWH, rhs.esWH );
   std::swap( esNoOfCourses, rhs.esNoOfCourses );

   Course::CourseKey* svck( esCourses );
   esCourses = rhs.esCourses;  rhs.esCourses = svck;

   std::swap( esReserved, rhs.esReserved );
} // EnrolledStudent::swap
```

Όπως βλέπεις, αυτή στηρίζεται στη *swap()* της βασικής κλάσης. Πράγματι, η "`Student::swap(rhs)`" –που μπορείς να τη γράψεις και "`this->Student:: swap(rhs)`"– ανταλλάσσει τις τιμές των υποαντικειμένων τύπου *Student* του `*this` και του *rhs*.

Ο τελεστής εκχώρησης δηλώνεται ως:

```
   EnrolledStudent& operator=( const EnrolledStudent& rhs );
```

και ορίζεται με τη γνωστή συνταγή:

```
EnrolledStudent& EnrolledStudent::operator=( const EnrolledStudent& rhs )
{
   if ( &rhs != this )
   {
      try {  EnrolledStudent tmp( rhs );
             swap( tmp );  }
      catch( EnrolledStudentXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
}; // Student( const Student& rhs )
```

Κατά τα άλλα, έχουμε τις απλές μεθόδους:

```
// getters
   unsigned int getWH() const {  return esWH;  }
   unsigned int getNoOfCourses() const {  return esNoOfCourses;  }
   const Course::CourseKey* getCourses() const {  return esCourses;  }
// setters
   void clearCourses() {  esNoOfCourses = 0;  esWH = 0;  }
```

και τις μεθόδους ενός στοιχείου:

```
   bool find1Course( const string& code ) const
                                        {  return ( findNdx(code) >= 0 );  }
   void add1Course( const Course& oneCourse );
   void delete1Course( const Course& oneCourse );
```

Οι μέθοδοι (**public**) *add1Course()* και *delete1Course()* μένουν (σχεδόν) όπως ήταν στην προηγούμενη μορφή. Η μόνη διαφορά είναι η αντικατάσταση του "`sWH`" από το "`esWH`".

Παρόμοια ισχύουν και για τις μεθόδους (**private**) *findNdx()*, *insert1Course()* και *erase1-Course()*: Τα *sIncr*, *sCourses*, *sNoOfCourses*, *sReserved* γίνονται *esIncr*, *esCourses*, *esNoOfCourses*, *esReserved* αντιστοίχως. Δίνουμε ενδεικτικώς την *insert1Course()* όπου αλλάζει και ο τύπος των εξαιρέσεων που ρίχνονται: *EnrolledStudentXptn* αντί για *StudentXptn*.

```
void EnrolledStudent::insert1Course( const Course::CourseKey& aCode )
{
   if ( esReserved <= esNoOfCourses+1 )
   {
      try {  renew( esCourses, esNoOfCourses, esReserved+esIncr );
             esReserved += esIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw EnrolledStudentXptn( getIdNum(), "insert1Course",
                                    EnrolledStudentXptn::allocFailed );
      }
   }
   esCourses[esNoOfCourses] = aCode;
   ++esNoOfCourses;
} // Student::insert1Course
```

Οι πολυμορφικές μέθοδοι, εκτός της *swap()*, είναι:

```
void EnrolledStudent::save( ostream& bout ) const
{
   Student::save( bout );

   bout.write( reinterpret_cast<const char*>(&esWH), sizeof(esWH) );
   bout.write( reinterpret_cast<const char*>(&esNoOfCourses),
               sizeof(esNoOfCourses) );
   for ( int k(0); k < esNoOfCourses; ++k )
      bout.write( esCourses[k].s, Course::cCodeSz );

   if ( bout.fail() )
      throw StudentXptn( getIdNum(), "save",
                                        StudentXptn::cannotWrite );
} // EnrolledStudent::save
```

```
void EnrolledStudent::load( istream& bin )
{
   EnrolledStudent tmp;

   tmp.Student::load( bin );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&tmp.esNoOfCourses),
                sizeof(esNoOfCourses) );
      if ( tmp.esNoOfCourses >= tmp.esReserved )
      {
         delete[] tmp.esCourses;
         try
         {
            tmp.esCourses = new Course::CourseKey[
                           ((tmp.esNoOfCourses/esIncr)+1)*esIncr ];
            tmp.esReserved = ((tmp.esNoOfCourses/esIncr)+1)*esIncr;
         }
         catch( bad_alloc )
         {
            throw EnrolledStudentXptn( tmp.getIdNum(), "load",
                                       EnrolledStudentXptn::allocFailed );
         }
      }
      for ( int k(0); k < tmp.esNoOfCourses; ++k )
         bin.read( tmp.esCourses[k].s, Course::cCodeSz );
      if ( bin.fail() )
         throw StudentXptn( tmp.getIdNum(), "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // EnrolledStudent::load

void EnrolledStudent::display( ostream& tout )
{
   Student::display( tout );
   tout << esWH << '\n' << esNoOfCourses << endl;
   for ( int k(0); k < esNoOfCourses; ++k )
      tout << esCourses[k].s << endl;
} // EnrolledStudent::display
```

Και οι τρεις στηρίζονται (με τις "`Student::save(bout)`", "`tmp.Student:: load(bin)`", "`Student::display(tout)`") στις αντίστοιχες μεθόδους της βασικής κλάσης.

Όπως βλέπεις, δεν υπήρξε ανάγκη για πρόσβαση σε κάποιο μέλος της βασικής κλάσης ώστε να το δηλώσουμε "**protected**". Για την ακρίβεια μας χρειάστηκε το *sIdNum* στις ρίψεις εξαιρέσεων αλλά

- η *getIdNum()* είναι "**inline**",
- στη ρίψη εξαίρεσης δεν «βιαζόμαστε»!

Αλλά, δεν τελειώσαμε ακόμη· στις νέες προδιαγραφές του προγράμματος έχουμε: «*Παρομοίως, στο μητρώο φοιτητών ένας φοιτητής που έχει εγγραφεί σε ένα τουλάχιστον μάθημα εμφανίζεται με αντικείμενο EnrolledStudent· αλλιώς εμφανίζεται με αντικείμενο κλάσης Student.*» Ας σκεφτούμε λοιπόν το εξής σενάριο: Θεωρούμε ότι διαβάζουμε –από το `enrllmnt.txt`– την τιμή ενός αντικειμενου της βασικής κλάσης *Student*, που το δείχνει το βέλος `scArr[k]`· διαβάζουμε λοιπόν με τη *readPartFromText()* στο `*scArr[k]`. Μετά διαβάζουμε τον αριθμό μαθημάτων. Αν είναι θετικός θα πρέπει να «αλλάξουμε» τον τύπο του αντικειμένου ώστε να μπορεί να δεχτεί τους κωδικούς μαθημάτων. Όπως στον πίνακα μαθημάτων έτσι και εδώ θα κάνουμε το εξής:

```
   EnrolledStudent* pOneStudent;
// . . .
      try {  pOneStudent = new EnrolledStudent( *scArr[ndx] );  }
      catch( bad_alloc& )
      {   throw . . .     }
```

```
      delete scArr[ndx];
      scArr[ndx] = pOneStudent;
```

Για τη "`new EnrolledStudent(*scArr[k])`" θα χρειαστούμε έναν δημιουργό μετατροπής:

```
EnrolledStudent::EnrolledStudent( const Student& rhs )
   :  Student( rhs ),  esWH( 0 )
{
   try  {  esCourses = new Course::CourseKey[ esIncr ];  }
   catch( bad_alloc )
   {  throw EnrolledStudentXptn( getIdNum(), "EnrolledStudent",
                                 EnrolledStudentXptn::allocFailed );  }
   esReserved = esIncr;
   esNoOfCourses = 0;
} // EnrolledStudent::EnrolledStudent
```

Ο ορισμός αυτού του δημιουργού μαζί με τον τελεστή εκχώρησης της κλάσης «νομιμοποιεί» και εδώ, όπως στην περίπτωση της *OfferedCourse*, την εκχώρηση "αντικείμενο *EnrolledStudent* = αντικείμενο *Student*"· πρίν από την εκχώρηση καλείται αυτός ο δημιουργός για να μετατρέψει το αντικείμενο *Student* σε αντικείμενο *EnrolledStudent* και έτσι εκτελείται η εκχώρηση

"αντικείμενο *EnrolledStudent* = *EnrolledStudent(* αντικείμενο *Student)*"

Και να πώς έγινε η

```
class EnrolledStudent : public Student
{
public:
// constructors, destructor
   explicit EnrolledStudent( int aIdNum=0 );
   EnrolledStudent( const EnrolledStudent& rhs );
   EnrolledStudent( const Student& rhs );
   virtual ~EnrolledStudent() {  delete[] esCourses;  };
// copy assignement
   EnrolledStudent& operator=( const EnrolledStudent& rhs );
// getters
   unsigned int getWH() const {  return esWH;  }
   unsigned int getNoOfCourses() const {  return esNoOfCourses;  }
   const Course::CourseKey* getCourses() const
                                          {  return esCourses;  }
// setters
   void clearCourses() {  esNoOfCourses = 0;  esWH = 0;  }
// 1 Course methods
   bool find1Course( const string& code ) const
                                {  return ( findNdx(code) >= 0 );  }
   void add1Course( const Course& oneCourse );
   void delete1Course( const Course& oneCourse );
// other methods
   virtual void swap( EnrolledStudent& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
private:
   enum { esIncr = 3 };
   unsigned int       esWH;
   unsigned int       esNoOfCourses;
   Course::CourseKey* esCourses;
   unsigned int       esReserved;

   int findNdx( const string& code ) const;
   void insert1Course( const Course::CourseKey& aCode );
   void erase1Course( int ndx );
}; // EnrolledStudent

typedef EnrolledStudent* PEnrolledStudent;
```

Και εδώ δεν χρειάζεται να κάνουμε επιφόρτωση των "!=" και "==" αφού μας καλύπτουν οι επιφορτώσεις που κάναμε για τη βασική κλάση.

Η κλάση εξαιρέσεων θα είναι:

```
struct EnrolledStudentXptn :  public StudentXptn
{
   enum { allocFailed=20 };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   EnrolledStudentXptn( int obk, const char* mn, int ec, const char* sv="" )
      :  StudentXptn( obk, mn, 0, sv ) {  errorCode = ec;  }
   EnrolledStudentXptn( int obk, const char* mn, int ec, int ev )
      :  StudentXptn( obk, mn, 0, ev ) {  errorCode = ec;  }
}; // EnrolledStudentXptn
```

Prj06.6 Η Κλάση *StudentCollection*

Και στην κλάση *StudentCollection* η βασική αλλαγή είναι στη δήλωση του δυναμικού πίνακα που από

```
   Student*   scArr;
```

γίνεται πίνακας βελών:

```
   PStudent*  scArr;
```

Και εδώ θα ξεκινήσουμε από τη *findNdx()* και θα δώσουμε την ίδια λύση που δώσαμε στην *CourseCollection*:

```
int StudentCollection::findNdx( int aIdNum ) const
{
   int ndx;
   if ( aIdNum <= 0 )
      ndx = -1;
   else
      ndx = linSearchP( scArr, scNOfStudents, 0, scNOfStudents-1,
                        Student(aIdNum) );
   return ndx;
} // StudentCollection::findNdx
```

Συνεχίζουμε με τον ερήμην δημιουργό και τον καταστροφέα που –και αυτοί– μοιάζουν με τους αντίστοιχους της *CourseCollection*:

```
StudentCollection::StudentCollection()
{
   try
   {
      scReserved = scIncr;
      scArr = new PStudent[ scReserved ];
      scNOfStudents = 0;
      scPAllEnrollments = 0;
      scArr[scNOfStudents] = new Student;
      for ( int k(scNOfStudents+1); k < scReserved; ++k )
         scArr[k] = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentCollectionXptn( "StudentCollection",
                                   StudentCollectionXptn::allocFailed);
   }
} // StudentCollection::StudentCollection

StudentCollection::~StudentCollection()
{
   for ( int k(0); k <= scNOfStudents; ++k )
```

```
      delete scArr[k];
   delete[] scArr;
} // StudentCollection::~StudentCollection
```

«Αντιγράφοντας» τις *erase1Course()* και *insert1Course()* της *CourseCollection*, παίρνουμε τις δύο μεθόδους «χαμηλού επιπέδου», *erase1Student()* και *insert1Student()* αντιστοίχως:

```
void StudentCollection::erase1Student( int ndx )
{
   delete scArr[ndx];
   scArr[ndx] = scArr[scNOfStudents-1];
   scArr[scNOfStudents-1] = scArr[scNOfStudents];
   scArr[scNOfStudents] = 0;
   --scNOfStudents;
} // StudentCollection::erase1Student

void StudentCollection::insert1Student( Student* pStudent )
{
   if ( scReserved <= scNOfStudents+1 )
   {
      try
      {
         renew( scArr, scNOfStudents+1, scReserved+scIncr );
         scArr[scNOfStudents+1] = scArr[scNOfStudents];      // αλλαγή φρουρού
         for ( int k(scNOfStudents+2); k < scReserved+scIncr; ++k )
            scArr[k] = 0;
         scReserved += scIncr;
      }
      catch( MyTmpltLibXptn& )
      {  throw StudentCollectionXptn( "insert1Student",
                                      StudentCollectionXptn::allocFailed );  }
   }
   scArr[scNOfStudents+1] = scArr[scNOfStudents];
   scArr[scNOfStudents] = pStudent;
   ++scNOfStudents;
} // StudentCollection::insert1Student
```

Πρόσεξε ότι και εδώ –όπως στην *CourseCollection*::*insert1Course*– αλλάξαμε τον τύπο της παραμέτρου σε "`Student* pStudent`" και δεν μας απασχολεί ο τύπος του αντικειμένου που δείχνει το βέλος: με αυτό θα ασχοληθεί η *add1Student()* που καλεί την *insert1Student()*.

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      try
      {
         try
         {
            const EnrolledStudent&
                 oneStudent( dynamic_cast<const EnrolledStudent&>(aStudent) );
            insert1Student( new EnrolledStudent(oneStudent) );
            const Course::CourseKey* aStCourses( aStudent.getCourses() );
            for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
            {
               scPAllEnrollments->add1StudentInCourse(
                                        StudentInCourse(aStudent.getIdNum(),
                                                        aStCourses[k].s) );
            }
         }
         catch( bad_cast )
         {  insert1Student( new Student(aStudent) );  }
      }
      catch( bad_alloc )
```

```
      { throw StudentCollectionXptn( "add1Student",
                                     StudentCollectionXptn::allocFailed ); }
   }
} // StudentCollection::add1Student
```

Όσο για τη *StudentCollection*::*delete1Student()*, θα την αλλάξουμε όπως αλλάξαμε και την *CourseCollection*::*delete1Course()*: Θα διαγράφουμε έναν φοιτητή αν έχουμε για αυτόν

- αντικείμενο κλάσης *Student* ή
- αντικείμενο κλάσης *EnrolledStudent* αλλά με *getNoOfCourses()* == 0.

```
void StudentCollection::delete1Student( int aIdNum )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx >= 0 )  // υπάρχει
   {
      EnrolledStudent* pOneStudent(
                       dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
      if ( pOneStudent != 0 )  // EnrolledStudent
      {
         if ( pOneStudent.getNoOfCourses() > 0 )
            throw StudentCollectionXptn( "delete1Student",
                                  StudentCollectionXptn::enrollRef,
                                        aIdNum );
      }
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student
```

Πρόσεξε ότι –όταν το αντικείμενο είναι κλάσης *EnrolledStudent*– δεν μπορούμε να εξετάσουμε τον αριθμό των μαθημάτων από το `scArr[ndx]`· πρέπει να χρησιμοποιήσουμε το `pOneStudent`.

Τώρα θα δούμε πώς διορθώνουμε τις *StudentCollection*::*add1Course()* και *StudentCollection*::*delete1Course()* όπως διορθώσαμε τις *CourseCollection*::*add1Student()* και *CourseCollection*::*delete1Student()*.

Στην §Prj06.5 –στην ανάδειξη της ανάγκης για δημιουργό αντικειμένου *EnrolledStudent* από αντικείμενο κλάσης *Student*– είχαμε δει ένα σενάριο χρήσης αυτού του δημιουργού· θα το χρησιμοποιήσουμε στην *add1Course()*:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   EnrolledStudent* pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
   if ( pOneStudent == 0 )
   {
      try {  pOneStudent = new EnrolledStudent( *scArr[ndx] );  }
      catch( bad_alloc& )
      {
         throw StudentCollectionXptn( "add1Course",
                                      StudentCollectionXptn::allocFailed );
      }
      delete scArr[ndx];
      scArr[ndx] = pOneStudent;
   }
   pOneStudent->add1Course( aCourse );
} // StudentCollection::add1Course
```

Όπως βλέπεις, μετατρέψαμε καταλλήλως την *CourseCollection*::*add1Student()*· παραπέμπουμε στον σχολιασμό της για οτιδήποτε δεν καταλαβαίνεις. Θα επισημάνουμε όμως ότι η μέθοδος αυτή αν κληθεί να καταχωρίσει μάθημα σε αντικείμενο τύπου *Student* θα κάνει τη μετατροπή του σε αντικείμενο *EnrolledStudent* χωρίς να κάνει οτιδήποτε το πρόγραμμα που την καλεί.

```
void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   EnrolledStudent* pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
   if ( pOneStudent != 0 )
   {
      pOneStudent->delete1Course( aCourse );
   // τα παρακάτω μπορεί και να περισσεύουν . . .
      if ( pOneStudent->getNoOfCourses() == 0 )
      {
         Student* tmp( new Student(*pOneStudent) );
         delete pOneStudent;
         scArr[ndx] = tmp;
      }
   }
} // StudentCollection::delete1Course
```

Και αυτή προέρχεται από μεταροπή της *CourseCollection*::*delete1Student()* και σε παραπέμπουμε εκεί για να ξαναθυμηθείς γιατί «*τα παρακάτω μπορεί και να περισσεύουν . . .*»

Από μετατροπές των μεθόδων *CourseCollection*::*save()* και *CourseCollection*::*load()* θα πάρουμε τις *StudentCollection*::*save()* και *StudentCollection*::*load()* αντιστοίχως, αφού προηγουμενως ορίσουμε:

```
struct SIndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
   short int    inhLvl;
   explicit SIndexEntry( int aIdNum=0, int aLoc=0, short int aInhL=0 )
      : sIdNum( aIdNum ),  loc( aLoc ),  inhLvl( aInhL )  {  }
}; // SIndexEntry

typedef SIndexEntry* PSIndexEntry;

bool operator!=( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return ( lhs.sIdNum != rhs.sIdNum );  }

bool operator==( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return !(lhs != rhs);  }
```

Η *SIndexEntry* διαφέρει από τη *CIndexEntry* μόνο στον τύπο του κλειδιού.

```
void StudentCollection::save( ofstream& bout, SIndexEntry* stdntIndex ) const
{
   if ( bout.fail() )
      throw  StudentCollectionXptn( "save",
                                    StudentCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      stdntIndex[k].sIdNum = scArr[k]->getIdNum();
      stdntIndex[k].loc = bout.tellp();
      EnrolledStudent*
                     pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[k]) );
      if ( pOneStudent == 0 )  // Student
      {  stdntIndex[k].inhLvl = 0;
         scArr[k]->save( bout );  }
      else                      // EnrolledStudent
      {  stdntIndex[k].inhLvl = 1;
         pOneStudent->save( bout );  }
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
```

```
} // StudentCollection::save

void StudentCollection::load( ifstream& bin, const SIndexEntry* stdntIndex )
{
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(n) );
   if ( !bin.eof() )
   {
      StudentCollection tmp;
      try
      {
         for ( int k(0); k < n && !bin.fail(); ++k )
         {
            Student* pOneStudent;
            if ( stdntIndex[k].inhLvl == 0 ) // Student
               pOneStudent = new Student;
            else                             // EnrolledStudent
               pOneStudent = new EnrolledStudent;
            pOneStudent->load( bin );
            tmp.insert1Student( pOneStudent );
         }
      }
      catch( bad_alloc& )
      {
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::allocFailed );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Τέλος, στην *checkWH()* θα πρέπει να κάνουμε μια μικρή αλλαγή: Θα ελέγχουμε τον εβδομαδιαίο φόρτο εργασίας μόνο για τα `*scArr[k]` που είναι κλάσης *EnrolledStudent*.

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      EnrolledStudent*
                     pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[k]) );
      if ( pOneStudent != 0 )
      {
         if ( pOneStudent->getWH() > maxWH )
            log << "student with id num " << pOneStudent->getIdNum()
                << ": " << pOneStudent->getWH() << " hours/week"
                << endl;
      }
   } // for
} // StudentCollection::checkWH
```

Prj06.7 *StudentInCourse* και *StudentInCourseCollection*

Στην κλάση *StudentInCourse* δεν αλλάζει κάτι· αλλά τώρα είναι μια κλάση συσχέτισης των κλάσεων *OfferedCourse* και *EnrolledStudent*. Δες το Σχ. Prj06-1.

Ούτε η *StudentInCourseCollection* έχει αλλαγές!

Prj06.8 Το 1ο Πρόγραμμα (Δημιουργία Αρχείου)

Ξεκινούμε με αυτό που δεν αλλάζει από το πρόγραμμα του Project 4: η *readStudentData()* παραμένει όπως ήταν. Φυσικά, η δομή των *allStudents* και *allCourses* είναι διαφορετική.

Η *loadCourses()* παραμένει *περίπου* όπως ήταν: διαγράφεται μόνον η εντολή "`oneCourse.clearStudents();`" αφού τώρα για τα αντικείμενα κλάσης *Course* δεν υπάρχει μέθοδος *clearStudents()*.

Η διαφορά βρίσκεται στη φύλαξη των συλλογών. Κατ' αρχάς έχουμε δύο ευρετήρια· έτσι στη `main()` έχουμε:

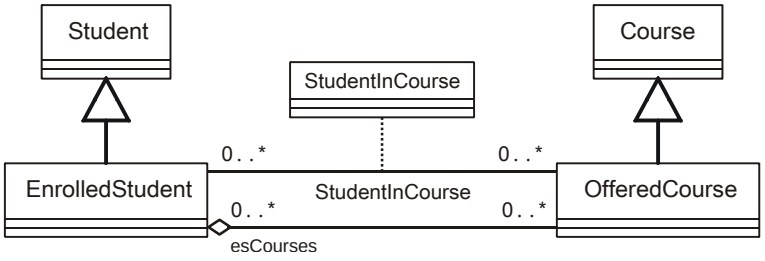
```
      CIndexEntry* cIndex;
      SIndexEntry* sIndex;
      try
      {  cIndex = new CIndexEntry[allCourses.getNOfCourses()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      try
      {  sIndex = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  delete[] cIndex;
         throw ProgXptn( "main", ProgXptn::allocFailed );  }
```

και η κλήση της *saveCollections()* γίνεται με τη

```
      saveCollections( allCourses, cIndex, allStudents, sIndex,
                       allEnrollments );
```

όπου:

```
void saveCollections( CourseCollection& allCourses,
                      CIndexEntry* cIndex,
                      StudentCollection& allStudents,
                      SIndexEntry* sIndex,
                      StudentInCourseCollection& allEnrollments )
{
   ofstream bout( "Courses.dta", ios_base::binary );
   allCourses.save( bout, cIndex );
   bout.close();
   bout.open( "courses.ndx", ios_base::binary );
//   bout.write( reinterpret_cast<const char*>(cIndex),
//               allCourses.getNOfCourses()*sizeof(CIndexEntry) );
   for ( int k(0); k < allCourses.getNOfCourses(); ++k )
      bout.write( reinterpret_cast<const char*>(&cIndex[k]),
                  sizeof(CIndexEntry) );
   bout.close();
   bout.open( "students.dta", ios_base::binary );
   allStudents.save( bout, sIndex );
   bout.close();
   bout.open( "students.ndx", ios_base::binary );
//   bout.write( reinterpret_cast<const char*>(sIndex),
//               allStudents.getNOfStudents()*sizeof(SIndexEntry) );
   for ( int k(0); k < allStudents.getNOfStudents(); ++k )
      bout.write( reinterpret_cast<const char*>(&sIndex[k]),
```



Σχ. Prj06-1 Η συσχέτιση *StudentInCourse* συσχετίζει τις κλάσεις *EnrolledStudent* (φοιτητές που έχουν εγγραφεί σε μαθήματα) και *OfferedCourse* (μαθήματα που έχουν δηλωθεί από φοιτητές). Αυτές κληρονομούν τις *Student* και *Course* αντιστοίχως.

```
                  sizeof(SIndexEntry) );
   bout.close();
   bout.open( "enrllmnt.dta", ios_base::binary );
   allEnrollments.save( bout );
   bout.close();
} // saveCollections
```

Θα έχουμε λοιπόν:

```
#include <string>
#include <fstream>
#include <new>
#include <iostream>

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "OfferedCourse.cpp"
#include "Student.cpp"
#include "EnrolledStudent.cpp"
#include "StudentInCourse.cpp"
#include "StudentInCourseCollection.h"
#include "CIndexEntry.h"
#include "CourseCollection.h"
#include "SIndexEntry.h"
#include "StudentCollection.h"
#include "CourseCollection.cpp"
#include "StudentCollection.cpp"
#include "StudentInCourseCollection.cpp"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadCourses( string flNm, CourseCollection& allCourses );
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log );
void saveCollections( CourseCollection& allCourses,
                      CIndexEntry* cIndex,
                      StudentCollection& allStudents,
                      SIndexEntry* sIndex,
                      StudentInCourseCollection& allEnrollments );

int main()
{
   OfferedCourse oc;
   try
   {
      CourseCollection allCourses;
      StudentCollection allStudents;
      StudentInCourseCollection allEnrollments;

      allEnrollments.setPAllStudents( &allStudents );
      allEnrollments.setPAllCourses( &allCourses );
```

```
      allCourses.setPAllEnrollments( &allEnrollments );
      allStudents.setPAllEnrollments( &allEnrollments );

      loadCourses( "gCourses.dta", allCourses );

      ofstream log( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt" );

      readStudentData( "enrllmnt.txt", allStudents, allEnrollments,
                       allCourses, log );

      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );

      log.close();

      CIndexEntry* cIndex;
      SIndexEntry* sIndex;
      try
      {  cIndex = new CIndexEntry[allCourses.getNOfCourses()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      try
      {  sIndex = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  delete[] cIndex;
         throw ProgXptn( "main", ProgXptn::allocFailed );  }

      saveCollections( allCourses, cIndex, allStudents, sIndex,
                       allEnrollments );

      delete[] sIndex;
      delete[] cIndex;
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( OfferedCourseXptn& x )  { /* . . . */ }
   catch( CourseXptn& x )  { /* . . . */ }
   catch( EnrolledStudentXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( StudentInCourseXptn& x )  { /* . . . */ }
   catch( CourseCollectionXptn& x )  { /* . . . */ }
   catch( StudentCollectionXptn& x )  { /* . . . */ }
   catch( StudentInCourseCollectionXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Prj06.9 Το 2ο Πρόγραμμα (Χρήση Αρχείου)

Στο πρόγραμμα χρήσης του αρχείου, στη `main()`, αλλάζουμε τη δήλωση

```
   IndexEntry* index;
```

σε

```
   SIndexEntry* sIndex;
```

Παρόμοια αλλαγή θα γίνει και στις δύο συναρτήσεις:

```
void loadIndex( string flNm,
                PSIndexEntry& sIndex, unsigned int& ndxSz );
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz );
```

Και στη μεν *loadIndex()* δεν κάνουμε άλλες αλλαγές· δεν ισχύει όμως το ίδιο και για τη *retrieve()*:

```
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "retrieve", ProgXptn::cannotOpen,
                      flNm.c_str() );
   string line;
   cout << "Student Id Number: ";  getline( cin, line, '\n' );
   while ( line != "ΤΕΛΟΣ" )
   {
      int idNum( atoi(line.c_str()) );
      int ndx( linSearch(sIndex, ndxSz, 0, ndxSz-1,
                         SIndexEntry(idNum)) );
      if ( ndx < 0 )
         cout << "unknown Student Id Number" << endl;
      else // ndx >= 0
      {
         Student* pOneStudent;
         try {
            if ( sIndex[ndx].inhLvl == 0 )
               pOneStudent = new Student;
            else // == 1
               pOneStudent = new EnrolledStudent;
         }
         catch( bad_alloc& )
         {
            bin.close();
            throw ProgXptn( "retrieve", ProgXptn::allocFailed );
         }
         bin.seekg( sIndex[ndx].loc );
         pOneStudent->load( bin );
         pOneStudent->display( cout );
         delete pOneStudent;  pOneStudent = 0;
      }
      cout << "Student Id Number: ";  getline( cin, line, '\n' );
   } // while
   bin.close();
} // retrieve
```

Πρόσεξε τώρα πώς χρησιμοποιούμε την πληροφορία για τον τύπο του αντικειμένου που έχουμε στο μέλος *inhLvl*:

- Αν –μετά την κλήση της *linSearch()* για αναζήτηση του *idNum* στον sIndex– η *ndx* πάρει μη αρνητική τιμή ξέρουμε ότι υπάρχει το αντικείμενο που ψάχνουμε. Το `sIndex[ndx].loc` μας λέει πού βρίσκεται.
- Το νέο ερώτημα που προκύπτει είναι: ποιος ο τύπος του αντικειμένου ή ποια *load()* να χρησιμοποιήσω; Αν το `sIndex[ndx].inhLvl` έχει τιμή "`0`" το αντικείμενο είναι τύπου *Student*· αλλιώς, αν έχει τιμή "`1`", είναι τύπου *EnrolledStudent*.
- Και πώς χρησιμοποιούμε αυτό που μάθαμε; Όπως βλέπεις, έχουμε δηλώσει ένα βέλος "`Student* pOneStudent`" που μπορεί να δείχνει αντικείμενα κλάσης *Student* και των παραγώγων της. Στην περίπτωσή μας έχουμε μια μόνον παράγωγη, την *EnrolledStudent*:

```
            if ( sIndex[ndx].inhLvl == 0 )
               pOneStudent = new Student;
            else // == 1
               pOneStudent = new EnrolledStudent;
```

- Μετά από αυτό οι εντολές:

```
         pOneStudent->load( bin );
         pOneStudent->display( cout );
```

θα χρησιμοποιήσουν τις *σωστές* μεθόδους αφού στη *Student* (αλλά και στην *EnrolledStudent*) έχουμε δηλώσει:

```
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
```

Ακόμη, στη "`delete pOneStudent`", θα κληθεί ο σωστός καταστροφέας αφού έχουμε δηλώσει

```
virtual ~Student() { };
```

(και "`virtual ~EnrolledStudent() { delete[] esCourses; }`".)

Η `main()` είναι:

```
#include <fstream>
#include <iostream>
#include <new>

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "EnrolledStudent.cpp"
#include "SIndexEntry.h"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadIndex( string flNm,
                PSIndexEntry& sIndex, unsigned int& ndxSz );
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz );

int main()
{
   SIndexEntry* sIndex;
   unsigned int ndxSz;
   try
   {
      loadIndex( "students.ndx", sIndex, ndxSz );
      retrieve( "students.dta", sIndex, ndxSz );
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( EnrolledStudentXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Prj06.10Τι Είδαμε σε Αυτό το Παράδειγμα

Κατ' αρχάς θα επαναλάβουμε ότι εδώ *δεν έχουμε* καλή σχεδίαση κλάσεων και ότι «*οι EnrolledStudent και OfferedCourse θα έλυναν τα πρόβλημα αν δεν κληρονομούσαν τις Student και Course (αλλά περιείχαν τα κλειδιά –sIdNum και cCode– για αντιστοίχιση.) Ξαναδιάβασε την §23.14.*»

Τώρα, ας επισημάνουμε αυτά που είδαμε σε αυτό παράδειγμα:

- Είδαμε δύο παραδείγματα κληρονομιάς: *Course* ← *OfferedCourse* (§Prj06.3) και *Student* ← *EnrolledStudent* (§Prj06.5).

- Είδαμε πώς ορίζουμε τους δημιουργούς των παράγωγων κλάσεων από αυτούς των βασικών. Στην περίπτωση της *OfferedCourse* βλέπεις ότι τα πάντα ρυθμίζονται στη λίστα εκκίνησης και τα σώματα των συναρτήσεων είναι κενά (§Prj06.3).
- Είδαμε πώς μπορούμε να ορίσουμε στις παράγωγες κλάσεις τους κατάλληλους *δημιουργούς μετατροπής*

```
OfferedCourse( const Course& oneCourse ); // (§Prj06.3)
EnrolledStudent( const Student& rhs ); // (§Prj06.5)
```

ώστε, μεταξύ άλλων, να «νομιμοποιήσουμε» και τις εκχωρήσεις

αντικείμενο *OfferedCourse* = αντικείμενο *Course*
αντικείμενο *EnrolledStudent* = αντικείμενο *Student*

χωρίς να επιφορτώσουμε τον σχετικό τελεστή εκχώρησης.

- Είδαμε πώς μπορούμε να βάλουμε στον ίδιο πίνακα αντικείμενα της βασικής και της παράγωγης κλάσης. Πώς; Κάνοντας πίνακα με βέλη προς αντικείμενα της βασικής:

```
PCourse*  ccArr; // (§Prj06.4)
PStudent* scArr; // (§Prj06.6)
```

- Φυσική συνέπεια της παραπάνω λύσης είναι η ανάγκη να μαθαίνουμε τον τύπο του αντικειμένου που δείχνει κάποιο βέλος `ccArr[k]` ή `scArr[k]`. Είδαμε τη λύση αυτού του προβλήματος με *δυναμική τυποθεώρηση*.[3]
- Είδαμε ακόμη και τη χρήση της *δυναμικής τυποθεώρησης* –μαζί με «ανορθόδοξη» διαχείριση εξαίρεσης– για να μάθουμε τον τύπο της πραγματικής παραμετρου που αντιστοιχεί σε τυπική *παράμετρο αναφοράς*.

[3] Δεν είναι κακή ιδέα να δοκιμάσεις και άλλη λύση.

κεφάλαιο

24

«Πέφτεις σε Λάθη...» – Εξαιρέσεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Θα ξαναδούμε –συμπληρωμένα– τα εργαλεία που σου προσφέρει η C++ (και η C) για να διαχειριστείς προβλήματα που παρουσιάζονται όταν εκτελείται το πρόγραμμά σου. Θα δεις τις εξαιρέσεις από την καλή και την ανάποδη:

- Θα κατάλάβεις την υπεροχή του μηχανισμού εξαιρέσεων της C++.
- Θα καταλάβεις όμως ότι πρέπει να ασφαλίζεις το πρόγραμμά σου και από τις εξαιρέσεις που το ίδιο ρίχνει.

Προσδοκώμενα αποτελέσματα:

Εξαιρέσεις χρησιμοποιούμε σε μεγάλη έκταση στα μισά από τα προηγούμενα κεφάλαια. Τι καινούριο θα μάθεις εδώ από πρακτική άποψη;

- Κάποια εργαλεία που θα κάνουν τη διαχείριση (των δικών μας) εξαιρέσεων πιο λειτουργική (τουλάχιστον στον καλύτερο εντοπισμό των προβλημλατων).
- Κανόνες για να σχεδιάσεις δικές σου κλάσεις εξαιρέσεων (αν δεν σου αρέσουν αυτές της C++ και οι δικές μας).
- Περισσότερα πράγματα για την ασφάλεια προς τις εξαιρέσεις.

Έννοιες κλειδιά:

- *συνάρτηση assert()*
- *τερματισμός εκτέλεσης*
- *συνάρτηση exit()*
- *προδιαγραφές εξαιρέσεων*
- *συναρτησιακή ομάδα* `try`
- *κλάσεις εξαιρέσεων*
- *ασφάλεια προς τις εξαιρέσεις*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Σε πολλά από τα προγράμματα-παραδείγματα που γράψαμε μέχρι τώρα δίναμε ιδιαίτερη προσοχή στη διασφάλιση κάποιων συνθηκών σε ορισμένα σημεία του προγράμματος. Χαρακτηριστικά παραδείγματα:

- εγκυρότητα των στοιχείων εισόδου που είναι μέρος της εξασφάλισης της προϋπόθεσης του προγράμματος,
- εξασφάλιση της αναλλοίωτης πριν από μια επαναληπτική εντολή,
- εξασφάλιση ότι τα ορίσματα μιας συνάρτησης ανήκουν στο πεδίο ορισμού της (εξασφάλιση προϋπόθεσης),
- εξασφάλιση της αναλλοίωτης μιας κλάσης.

Στην αρχή, μάθαμε να αντιδρούμε «βιαίως» αν βρίσκαμε να μην ισχύει κάποια επιθυμητή συνθήκη. Στην §4.3, στο πρόγραμμα της ελεύθερης πτώσης, είχαμε:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
```

Αν η συνθήκη-όρισμα της *assert()* πάρει τιμή `false`, η συνάρτηση βγάζει ένα μήνυμα και διακόπτει την εκτέλεση του προγράμματος.

Στο παράδ. 1 της §5.5 γράφαμε κάπως αλλιώς το ίδιο πρόγραμμα:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   // κάνουμε τους υπολογισμούς
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
```

Αν η τιμή που δίνουμε στην h δεν είναι έγκυρη (αν δηλαδή $h < 0$) τότε βγάζουμε ένα μήνυμα και δεν κάνουμε οτιδήποτε άλλο.

Παρομοίως, στην §7.7 γράφαμε τη

```
   // factorial -- Υπολογίζει το a!
   unsigned long int factorial( int a )
   {
      if ( a < 0 )
      {
         cout << " η factorial κλήθηκε με όρισμα " << a << endl;
         exit( EXIT_FAILURE );
      }
      unsigned long int fv( 1 );
      for ( int k(1);  k <= a;  ++k )  fv *= k;
      return fv;
   } // factorial
```

(ή κάπως έτσι.)

Εδώ, στην περίπτωση που η *factorial()* κληθεί με όρισμα εκτός πεδίου ορισμού (αρνητικό), η συνάρτηση διακόπτει την εκτέλεση του προγράμματος.

Αργότερα μάθαμε να είμαστε πιο ελαστικοί: να δίνουμε τη δυνατότητα στον χρήστη να ξαναδώσει τα (πιθανόν λαθεμένα) στοιχεία εισόδου ή *να ρίχνουμε μια εξαίρεση* αντί να καλέσουμε την *exit()*.

Από τα μέχρι τώρα παραδείγματα θα πρέπει να έχεις αντιληφθεί ότι η αντιμετώπιση των σφαλμάτων στο πρόγραμμα έχει δύο σκέλη:

- Να βρούμε το σφάλμα *εγκαίρως*, πριν μας «κάνει τη ζημιά». Για παράδειγμα, στο πρόγραμμα της ελεύθερης πτώσης (§2.7 και §3.8) αφού η προϋπόθεση λέει ότι θα πρέπει να έχουμε $h \geq 0$ δεν χρειάζεται να φθάσουμε στον υπολογισμό της $\sqrt{2h/g}$.
- Να αντιμετωπίσουμε την κατάσταση. Και πώς γίνεται αυτό;
  - Ένας τρόπος είναι να σταματήσουμε την εκτέλεση του προγράμματος αφού προηγουμένως φυλάξουμε όσα μπορούμε από αυτά που ήδη έκανε. Αυτή η λύση είναι η λιγότερο επιθυμητή.
  - Ένας καλύτερος τρόπος είναι να «καθαρίσουμε» ό,τι έχει «μολυνθεί» από τα λανθασμένα στοιχεία, να φέρουμε το πρόγραμμά μας σε μια αποδεκτή κατάσταση και να συνεχίσουμε από εκεί.

Στο κεφάλαιο αυτό θα συζητήσουμε πιο εκτεταμένα τα εργαλεία που μας δίνει η C++ για να αντιμετωπίζουμε τις «δύσκολες» καταστάσεις. Κατά βάση θα δούμε τις εξαιρέσεις, αλλά πριν από αυτό θα δούμε μερικά εργαλεία που μας δίνει η C, κυρίως αυτά που χρησιμοποιήσαμε ήδη αμέσως ή εμμέσως.

Σημείωση:▶

Πριν προχωρήσουμε θα πρέπει να αναφέρουμε ότι και η C έχει σύστημα εξαιρέσεων με το οποίο όμως δεν θα ασχοληθούμε.◀

24.1 Τι Έχουμε από τη C

Όπως είδαμε η C μας δίνει τη συνάρτηση *assert()* για να ελέγχουμε αν ισχύει κάποια συνθήκη επαλήθευσης σε συγκεκριμένο σημείο του προγράμματός μας.[1] Είδαμε ακόμη τη συνάρτηση *exit()* με την κλήση της οποίας τερματίζεται η εκτέλεση του προγράμματός μας.

24.1.1 Η Συνάρτηση *assert()*

Μπορείς να σκέφτεσαι τη (μακρο)συνάρτηση *assert()* ως:

```
void assert( bool test )
```

Όταν κληθεί, αν το όρισμά της υπολογισθεί σε `true` δεν «αντιδρά». Αν υπολογισθεί `false` τότε δίνει μήνυμα και διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση *abort()*.

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cassert>`".

[1] Για το νέο πρότυπο της C προτάθηκε (με την τεχνική αναφορά ISO/IEC TR 24731-1-2007) μια οικογένεια εργαλείων για τη διαχείριση παραβίασης περιορισμών (constraints) κατά τη διάρκεια της εκτέλεσης. Σε σχέση με τα εργαλεία αυτά η *assert()* είναι μάλλον απλοϊκή.

24.1.2 Συναρτήσεις Τερματισμού Εκτέλεσης

Η συνάρτηση *exit()*: Την είδαμε για πρώτη φορά στην §7.7 (Παράδ. 1)· προκαλεί κανονικό τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void exit( int status )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση:

- Κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την *atexit()*, που θα δούμε στην συνέχεια. Οι συναρτήσεις εκτελούνται με σειρά αντίστροφη της σειράς καταχώρισης.
- Στη συνέχεια, οι ενταμιευτές των ρευμάτων που έχουν ανοιχθεί για γράψιμο γράφονται στα αντίστοιχα αρχεία και όλα τα ρεύματα κλείνουν.
- Τέλος, τερματίζεται η εκτέλεση του προγράμματος και ο έλεγχος περνάει στο ΛΣ. Αν η τιμή της *status* είναι "`0`" ή *EXIT_SUCCESS* στο ΛΣ περνάει η πληροφορία *επιτυχούς τερματισμού* (successful termination). Αν η τιμή της *status* είναι "`1`" ή *EXIT_FAILURE* περνάει η πληροφορία *ανεπιτυχούς τερματισμού* (unsuccessful termination).[2]

Η συνάρτηση *abort()*: Είπαμε παραπάνω ότι η *assert()* μπορεί να «*διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση abort()*.» Η *abort()* προκαλεί *μη-κανονικό* τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void abort( )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση τερματίζεται η εκτέλεση του προγράμματος και στο ΛΣ περνάει η πληροφορία *ανεπιτυχούς τερματισμού*. Το τι θα γίνει με τα ανοιχτά ρεύματα εξαρτάται από τη συγκεκριμένη υλοποίηση.

Η συνάρτηση *_Exit()*: Η δήλωσή της (στο `cstdlib`) είναι (περίπου):

```
void _Exit( int status );
```

Προκαλεί κανονικό τερματισμό του προγράμματος και περνάει στο ΛΣ πληροφορία όπως η *exit()*. Σε σχέση με τα ρεύματα ισχύουν αυτά που είπαμε για την *abort()*.

Η συνάρτηση *atexit()*: Παραπάνω είπαμε ότι όταν κληθεί η *exit()* «*κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την atexit()*».

Η δήλωσή της *atexit()* (στο `cstdlib`) είναι (περίπου):

```
int atexit( void (*func)() );
```

Δηλαδή, η *atexit()* περιμένει ένα όρισμα που είναι βέλος προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την "`void f()`").

Αν η καταχώριση γίνει επιτυχώς η συνάρτηση επιστρέφει "`0`" ενώ σε περίπτωση αποτυχίας επιστρέφεται μη-μηδενική τιμή. Κάθε υλοποίηση της C++ θα πρέπει να επιτρέπει τουλάχιστον 32 καταχωρίσεις.

Δες ένα παράδειγμα:

```
#include <iostream>
#include <cstdlib>
using namespace std;

void bye()
{  cout << "Bye bye!" << endl;  }

void thatsAll()
{  cout << "That is all!!" << endl;  }
```

---

[2] Στο `cstdlib` υπάρχουν οι ορισμοί:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

```
int main()
{
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   if ( atexit(&thatsAll) != 0 )
      cout << "thatsAll registration failed" << endl;

   exit(0);
}
```

Αποτέλεσμα:

```
That is all!!
Bye bye!
```

Πρόσεξε τα εξής:

1. Πρώτα έγινε η καταχώριση της *bye()* και μετά της *thatsAll()*· αλλά κατά τον τερματισμό εκτελείται πρώτα η *thatsAll()* και μετά η *bye()*.
2. Όπως μάθαμε στην §14.3, μπορούμε να μην βάλουμε το "**&**" και να γράψουμε πιο απλά:

```
   if ( atexit(bye) != 0 ) . . .
   if ( atexit(thatsAll) != 0 ) . . .
```

Προσοχή! Αν στο πρόγραμμά σου δεν βάλεις "**using namespace std**" θα πρέπει να χρήσιμοποιείς το πρόθεμα "**std::**" για όλες τις συναρτήσεις που είδαμε παραπάνω.

24.1.2.1 Σχέση std::exit() και return

Τι διαφορά υπάρχει μεταξύ τερματισμού με *exit()* και τερματισμού με "**return**" στη **main**;

- Αν δώσεις "**return** *n*" θα καταστραφούν –με κλήση των αντίστοιχων καταστροφέων– όλα τα αντικείμενα της **main** που έχουν δημιουργηθεί στη στοίβα και μετά θα εκτελεσθεί η κλήση "**exit(n)**".
- Αν δώσεις "**exit(n)**" δεν θα κληθούν οι καταστροφείς· έτσι, αν περιμένεις κάτι από αυτούς θα πρέπει να φροντίσεις να γίνει με άλλον τρόπο.

Να ένα παράδειγμα:

```
#include <iostream>
#include <cstdlib>
using namespace std;

struct B
{
   B( int k ) :  bM( k ) { };
   ~B()
   {  cout << "destruction of a B object: " << bM << endl;  }
private:
   int bM;
}; // B

B b1( 3 );

int main()
{
   B b2( 7 );

   return 0;
}
```

Αυτό το πρόγραμμα θα σου δώσει:

```
destruction of a B object: 7
destruction of a B object: 3
```

Αν αλλάξεις το "**return 0**" σε "**exit(0)**" θα πάρεις:

```
destruction of a B object: 3
```

Δηλαδή καλείται ο καταστροφέας να καταστρέψει το *b1* που υλοποιείται σε *στατική* μνήμη αλλά όχι το *b2* που υλοποιείται σε μνήμη στοίβας.[3]

Ωραία όλα αυτά, αλλά αφού και στις δύο περιπτώσεις θα σταματήσει η εκτέλεση του προγράμματος θα «καταστραφούν» τα πάντα! Τι θα μπορούσε να περιμένει από έναν καταστροφέα ένα *καλό* πρόγραμμα; Μια απάντηση μπορεί να είναι η εξής: Να «καθαρίζει» τα αντικείμενα από ευαίσθητες πληροφορίες (ξαναδές την §16.14) πριν επιστραφεί η μνήμη τους στη στοίβα ή στον σωρό!

24.1.3 Τι Λάθος Έκανα; *errno*

Αν έχουμε κάποιο πρόβλημα όταν καλούμε μια συνάρτηση από τις βιβλιοθήκες της C (π.χ.: με λάθος τιμές παραμέτρων) μπορεί να πάρουμε μήνυμα λάθους μέσω της καθολικής μεταβλητής *errno*. Ας ξαναγράψουμε το πρόγραμμα της ελεύθερης πτώσης:

```
#include <iostream>
#include <cmath>
#include <cerrno>
#include <string>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
// προσπάθησε να υπολογίσεις το tP
   errno = 0;
   tP = sqrt( (2/g)*h );
   if ( errno != EDOM )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX) && (tP ≈ √(2h/g))
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   // Τύπωσε τα tP, vP
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   }
   else // errno == EDOM
      cout << strerror( errno )
           << ": το ύψος δεν μπορεί να είναι αρνητικό" << endl;
}
```

Να δύο παραδείγματα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: 80
 Αρχικό ύψος = 80 m
 Χρόνος Πτώσης = 4.03855 sec
 Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
 Δώσε μου το αρχικό ύψος σε m: -10
Domain error: το ύψος δεν μπορεί να είναι αρνητικό
```

Τι είναι η *errno*; Μπορείς να τη σκέφτεσαι ως μια καθολική μεταβλητή που έχει δηλωθεί «πριν από σένα για σένα» ως[4]

```
int errno;
```

---

[3] Φυσικά, αν δηλώσεις "`static B b2( 7 )`" θα κληθεί ο καταστροφέας και για αυτό.

[4] Το πρότυπο της C λέει ότι μπορεί να είναι κάποιος μακροορισμός που καταλήγει σε μια *τροποποιήσιμη τιμή-l*.

Όταν αρχίζει η εκτέλεση του προγράμματος η τιμή της *errno* είναι "`0`". Ορισμένες συναρτήσεις βάζουν στην *errno* θετική τιμή *αλλά δεν υπάρχει συνάρτηση που να τη μηδενίζει.*

- Ορισμένες συναρτήσεις, μεταξύ αυτών και οι συναρτήσεις της math, αν τροφοδοτηθούν με ορίσματα εκτός πεδίου ορισμού βάζουν στην *errno* τιμή *EDOM* (ορίζεται στο `cerrno`).
- Ορισμένες συναρτήσεις όταν υπολογίσουν εξαιρετικώς μεγάλη τιμή (που δεν παριστάνεται στον τύπο του αποτελέσματος) βάζουν στην *errno* τιμή *ERANGE*.

Τι κάναμε λοιπόν εδώ; Βάλαμε στην *errno* τιμή "`0`" και –χωρίς να ελέγξουμε την τιμή της *h*– δώσαμε "`tP=sqrt((2/g)*h)`". Μετά από αυτό ελέγχουμε την τιμή της *errno* και προχωρούμε μόνο στην περίπτωση που `errno != EDOM`. Ενώ στο αρχικό πρόγραμμα προσέχαμε «*να βρούμε το σφάλμα εγκαίρως, πριν μας "κάνει τη ζημιά"*» εδώ προχωρήσαμε στο επικίνδυνο βήμα και μετά είπαμε «*δεν έπρεπε να το κάνω!*»

Η συνάρτηση *strerror()* παίρνει την τιμή της *errno* και μας δίνει το αντίστοιχο μήνυμα. Στην περίπτωσή μας απεικόνισε το *EDOM* στο "`Domain error`".

Ως δεύτερο παράδειγμα δες το παρακάτω, αφού πρώτα ξαναδείς αυτά που λέγαμε στην §13.8.

```
double d;
char* p;

errno = 0;
d = strtod( "7.1e+318", &p );
if ( (d == HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
errno = 0;
d = strtod( "-7.1e+318", &p );
if ( (d == -HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
```

Αποτέλεσμα:

```
TRUE
TRUE
```

Πράγματι, οι τιμές $\pm 7.1\times10^{+318}$ δεν μπορούν να παρασταθούν στον τύπο `double` και έτσι παίρνουμε ως αποτέλεσμα `±HUGE_VAL` αντιστοίχως ενώ η *errno* παίρνει τιμή *ERANGE*. Το C99 λέει: «*Αν η σωστή τιμή είναι εκτός της περιοχής τιμών που μπορεί να παρασταθούν, επιστρέφεται συν ή πλην HUGE_VAL και η τιμή του ERANGE αποθηκεύεται στην errno.*» Για τον λόγο αυτόν λέγαμε στην §13.8 ότι πρέπει να ελέγχεις τη συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

Γενικώς, οι συναρτήσεις των βιβλιοθηκών της C σε περίπτωση λάθους

- είτε επιστρέφουν τιμή εκτός πεδίου τιμών (όπως η `HUGE_VAL`),
- είτε επιστρέφουν *ακραία* τιμή εντός του πεδίου τιμών (όπως η `INT_MAX`).

ενώ παραλλήλως μπορεί να δίνουν και κάποια θετική τιμή στην *errno*.

Έτσι, η οδηγία που δίνεται για τη χρήση της *errno* είναι η εξής:

- Πριν από την κλήση της συνάρτησης βάλε την εντολή "`errno = 0`".
- Μετά την κλήση της έλεγξε την τιμή που επίστρεψε η συνάρτηση και την τιμή της *errno*.

Πάντως:

- Το πρότυπο δεν δεσμεύει τις διάφορες υλοποιήσεις να επιστρέφουν σε κάθε περίπτωση αποτυχίας συγκεκριμένες τιμές. Για παράδειγμα: δεν λέει τι τιμή θα επιστρέφει η *sqrt()* σε περίπτωση που θα τροφοδοτηθεί με αρνητικό όρισμα.[5]
- Δεν δεσμεύει τις υλοποιήσεις στο να δίνουν τιμή στην *errno*. Για παράδειγμα, για τη *strtod()* το C99 λέει: «*Αν στο αποτέλεσμα έχουμε υποχείλιση (underflow), η συνάρτηση επιστρέφει τιμή που το μέγεθός της δεν υπερβαίνει τον ελάχιστο κανονικοποιημένο*

[5] Οι περισσότερες υλοποιήσεις θα επιστρέψουν `NaN`.

αριθμό του τύπου επιστροφής· το αν η errno θα πάρει τιμή ERANGE καθορίζεται από την υλοποίηση.»[6]

Όπως καταλαβαίνεις, με αυτόν τον τρόπο δεν είναι εύκολο να γράψεις προγράμματα που θα είναι δυνατόν να μεταφερθούν από τη μια υλοποίηση στην άλλη χωρίς αλλαγές.

Τι θα κάνεις λοιπόν;

- ***Προσπάθησε να αποφεύγεις τη χρήση της errno.***
- ***Αν πρέπει να το χρησιμοποιήσεις υποχρεωτικώς συμβουλέψου το C99 και το εγχειρίδιο χρήσης της C που χρησιμοποιείς.***

24.2 Μήνυμα με Τιμές Συνάρτησης και Παραμέτρων

Τα εργαλεία που μας δίνει η C είναι στην πραγματικότητα αυτά που χρειάζονται οι προγραμματιστές που χρησιμοποιούν γλώσσες χωρίς εργαλεία για τη διαχείριση εξαιρέσεων. Οι συναρτήσεις που υπήρχαν στις παλιές βιβλιοθήκες έκοβαν πολύ εύκολα την εκτέλεση του προγράμματος όταν εβρισκαν κάποιο σοβαρό λάθος. Έτσι και οι προγραμματιστές έγραφαν τα προγράμματά τους με τον ίδιο τρόπο· επομένως, η *exit()* και η *abort()* ήταν εργαλεία πολύ χρήσιμα. Αργότερα τα πράγματα άλλαξαν· ο προγραμματιστής δεν ήθελε να του κόβει το πρόγραμμα η *sqrt()* ή κάποια συνάρτηση που έγραφε ο ίδιος ο προγραμματιστής και έτσι η *exit()* και η *abort()* έπαψαν να είναι τόσο χρήσιμες. Αλλά αυτό είχε ως αντίτιμο το να γεμίσει το πρόγραμμα με **if**· και αυτό μπορεί να μην ήταν ενοχλητικό από την άποψη της ταχύτητας εκτέλεσης –οι ταχύτητες των επεξεργαστών είχαν βελτιωθεί– ήταν όμως ενοχλητικό από την άποψη της πολυπλοκότητας του κώδικα καθώς έτσι είχαμε «ανακάτεμα» του αλγόριθμου με τις άμυνες από πιθανά λάθη.

Οι συναρτήσεις έστελναν μηνύματα για τα διάφορα προβλήματα που έβρισκαν με την τιμή της συνάρτησης, με παραμέτρους **out** ή με καθολικές μεταβλητές σαν την *errno*.

Σε πολλές περιπτώσεις, για να μην αυξάνεται το πλήθος των παραμέτρων, η ίδια παράμετρος χρησιμοποιείται για να μεταφέρει όχι μόνον κωδικό λάθους αλλά και άλλες πληροφορίες. Ήδη, εδώ είδαμε:

- Στη συνάρτηση *bisection()*, στην §14.3, επιστρέφουμε μήνυμα λάθους με μια παράμετρο αναφοράς, την *errCode*. Αυτό το είδος λάθους που έχουμε εκεί πιθανότατα απαιτεί ρίψη εξαίρεσης. Ας πούμε όμως ότι κάνεις την αλλαγή που λέγαμε στις παρατηρήσεις και ζητάς να πετύχεις την $|f(x)| < \varepsilon'$ σε *nMax* επαναλήψεις το πολύ (διότι η $|b - a|/2 < \varepsilon$ μπορεί να μη είναι ικανοποιητική). Στην περίπτωση αυτή βάζουμε και άλλη μια παράμετρο, *nMax*, με την οποία βάζουμε έναν μέγιστο αριθμό επαναλήψεων που επιθυμούμε να γίνουν. Πώς θα γνωστοποιήσει η *bisection()* στη συνάρτηση που την καλεί το εξής γεγονός: "*έκανα nMax επαναλήψεις αλλά η* $|f(x)| < \varepsilon'$ *δεν ισχύει*"; Με εξαίρεση; Όχι βέβαια.
- Στην άσκηση 13-10 σε καλούμε να γράψεις συνάρτηση, την *trinomial()*, που μέσω μιας παρόμοιας παραμέτρου επιστρέφει *το πλήθος των πραγματικών ριζών ή κωδικό λάθους*. Αλλά, ας ξανασκεφτούμε αυτήν την περίπτωση: Το να καλέσουμε τη συνάρτηση για να μας λύσει την εξίσωση "7 = 0" (κωδικός: –1) ή "0 = 0" (κωδικός: *INT_MAX*) σημαίνει πιθανότατα ότι το πρόγραμμά μας έχει κάποιο σοβαρό λάθος. Το να βρούμε ότι το τριώνυμο έχει 0 ή 1 ή 2 πραγματικές ρίζες είναι κάτι το αναμενόμενο. Έτσι, το να εξετάζουμε την τιμή μιας παραμετρου για να καταλάβουμε από την τιμή της αν είναι επιστρεφόμενη τιμή ή κωδικός λάθους μας οδηγεί στο να γράψουμε ένα μπερδεμένο πρόγραμμα. Αν έχεις λύσει την άσκηση, θα έχεις δει ήδη το σχετικό μπέρδεμα και *μέσα* στη συνάρτηση.

[6] Ο μεταγλωττιστής gcc (Dev-C++) βάζει τιμή *ERANGE* στην *errno*.

Θα ήταν λοιπόν καλύτερο να βάλουμε *δύο* παραμέτρους με επιστρεφόμενη τιμή: μια με τον κωδικό σφάλματος και μιαν άλλη για το πλήθος των πραγματικών ριζών. Θα μπορούσαμε ακόμη να μιμηθούμε τη C: να επιστρέφουμε με παράμετρο το πλήθος και να βάζουμε κωδικό λάθους στην *errno* (ή σε κάποια δική μας *myErrNo*.)

Ναι, θα μπορούσαμε να κάνουμε κάτι από αυτά αλλά το μπέρδεμα της διαχείρισης σφαλμάτων με συνήθεις υπολογισμούς παραμένει.

Το πόσο μπερδεμένο μπορεί να είναι ένα πρόγραμμα γραμμένο με αυτόν τον τρόπο φαίνεται όταν έχεις συναρτήσεις που επιστρέφουν μια τιμή. Θα ξαναγράψουμε το πρόγραμμα-παράδειγμα για τους συνδυασμούς που είδαμε στην §14.9.1 με τις εξής απαιτήσεις:

- Κάθε μια από τις συναρτήσεις *comb()*, *factorial()* και *natProduct()* θα πρέπει να αποφεύγει υπολογισμούς με ορίσματα εκτός πεδίου ορισμού, ενώ σε τέτοια περίπτωση θα πρέπει να επιστρέφει κάποιο μήνυμα λάθους. Επειδή οι συναρτήσεις αυτές είναι χρήσιμες γενικώς, θα πρέπει να έχουν μορφή που να τις κάνει χρήσιμες και πέρα από το συγκεκριμένο πρόγραμμα.[7]

  Ας πάρουμε τη *natProduct()*. Για αυτήν η προϋπόθεση είναι:

$$0 < m \le n$$

Θα μας δίνει πάντοτε ως αποτέλεσμα έναν θετικό ακέραιο. Για να επιστρέψουμε μήνυμα λάθους

- μπορούμε να βάλουμε μία ακόμη παράμετρο, οπότε θα την κάνουμε (σύμφωνα με τις συνήθειές μας) `void`,

```
// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
void natProduct( int m, int n,
                 unsigned long int& fv, int& err )
{
   if ( m <= 0 || n < m )
      err = 1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
      err = 0;
   }
} // natProduct
```

- μπορούμε όμως να επιστρέφουμε και κάποια αρνητική τιμή:

```
long int natProduct( int m, int n )
{
   long int fv;

   if ( m <= 0 || n < m )
      fv = -1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
   }
   return fv;
} // natProduct
```

Πώς γίνεται τώρα η *factorial()*; Για την πρώτη περίπτωση θα έχουμε:

```
void factorial( int a,
                unsigned long int& fv, int& err )
{
   if ( a == 0 ) {  fv = 1;  err = 0;  }
```

[7] Η τελευταία απαίτηση μας λέει απλούστατα ότι δεν μπορούμε να στηριχθούμε σε έναν έλεγχο στη **main** και να «καθαρίσουμε» για τα πάντα.

```
            else natProduct( 1, a, fv, err );
} // factorial
```

ενώ για τη δεύτερη περίπτωση παραμένει σχεδόν όπως ήταν:

```
long int factorial( int a )
{
   return ( (a == 0) ? 1 : natProduct(1, a) );
} // factorial
```

Το μόνο που άλλαξε είναι ο τύπος του αποτελέσματος –που έγινε **long int**– για να επιτρέπει την επιστροφή της τιμής "**-1**" σε περίπτωση λάθους.

Η εντυπωσιακή αλλαγή γίνεται στην *comb()*:

```
// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
void comb( int m, int n,
           unsigned long int& fv, int& err )
{
   unsigned long int v1, v2;

   if ( n < m-n )
   {
      factorial( n, v1, err );
      if ( err == 0 )
      {
         natProduct( m-n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
   else
   {
      factorial( m-n, v1, err );
      if ( err == 0 )
      {
         natProduct( n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
} // comb
```

Φρίκη! Υπολογισμοί και διαχείριση σφαλμάτων έγιναν «μαλλιά κουβάρια»! Σύγκρινε αυτήν την εκδοχή με την αρχική… Ίδια είναι η κατάσταση και στην περίπτωση που η συνάρτηση επιστρέφει τιμή "**-1**" (άσκηση για σένα!)

Στη **main** θα έχουμε:

```
int main()
{
   int m, n;
   unsigned long int s;
   int  err;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   comb( m, n, s, err );
   if ( err == 0 )
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << s << endl;
   else
      cout << " Λάθος δεδομένα" << endl;
} // main
```

Και στις δύο περιπτώσεις, έχουμε χάσει την ευκολία και την κομψότητα της κλήσης μιας συνάρτησης με τύπο.

Να συγκρίνουμε αυτό το πρόγραμμα και την εκτέλεσή του με αυτό που γράψαμε στην §14.9.1:

1. Το πρόγραμμα με την εξαίρεση είναι πολύ πιο απλό και πιο κατανοητό. Η διαφορά έγκειται στη δυνατότητα που έχουμε να καλούμε τις συναρτήσεις (με τύπο) μέσα σε παραστάσεις.
2. Στο πρώτο πρόγραμμα το ξετύλιγμα της στοίβας γίνεται αυτομάτως και πολύ απλά. Εδώ, γίνεται βήμα-προς-βήμα αφού κάθε φορά ελέγξουμε την τιμή της *err*.
3. Το δεύτερο πρόγραμμα έχει γεμίσει με εντολές για τη διαχείριση κωδικών λάθους. Πιο χαρακτηριστική είναι η περίπτωση της *comb()*.

Βλέπουμε λοιπόν δύο σημαντικά πλεονεκτήματα του μηχανισμού διαχείρισης εξαιρέσεων που προσφέρει η C++ (αλλά και άλλες γλώσσες προγραμματισμού):

- Το αυτόματο ξετύλιγμα της στοίβας και η δυνατότητα χρήσης συναρτησεων με τύπο.
- Ο συστηματικός *διαχωρισμός των εντολών διαχείρισης εξαιρετικών συμβάντων* από το υπόλοιπο πρόγραμμα.

Στη συνέχεια θα δούμε και άλλα.

24.3 Συμπληρώματα στην «Ιστορία με Εξαιρέσεις»

Θα δώσουμε τώρα μερικά συμπληρώματα στην «Ιστορία με Εξαιρέσεις» που είδαμε στην §14.9.1. Θα ξεκινήσουμε όμως με έναν ορισμό που αντιγράφουμε από το Java Tutorial:[8] «*Ο όρος εξαίρεση είναι συντομογραφία της φράσης "εξαιρετικό συμβάν."*

- ***Μια εξαίρεση (exception) είναι ένα συμβάν –κατά τη διάρκεια της εκτέλεσης ενός προγράμματος– που διακόπτει την ομαλή ροή των εντολών του προγράμματος.»***

Όταν σε κάποιο σημείο μιας συνάρτησης βρούμε να μην ισχύει κάποια συνθήκη που είναι αναγκαία, **ρίχνουμε** (throw) ή **εγείρουμε** (raise) μιαν εξαίρεση:

"`throw`", *αντικείμενο εξαίρεσης* ;

π.χ.:

```
throw -1;      throw n;      throw x;
```

Συνήθως χρησιμοποιούμε τον όρο *εξαίρεση* αντί για *αντικείμενο εξαίρεσης*.

Τα χαρακτηριστικά μιας εξαίρεσης είναι:

- ο τύπος και
- *η τιμή*

του αντικειμένου της.

Στη συνάρτηση *v()* που είδαμε στην §14.9 έχουμε την εντολή "`throw x;`". Τι θα γίνει αν εκτελεσθεί αυτή η εντολή; Σύμφωνα με αυτά που είδαμε στην §14.9.1, «*Τα σχετικά με την κλήση της» v() «φεύγουν από τη στοίβα και επιστρέφουμε»* στη συνάρτηση που κάλεσε τη *v()*. Και η *x*; Αυτή μας χρειάζεται, αφού είναι το αντικείμενο της εξαίρεσης (και θα πρέπει να γίνει τιμή της παραμετρου της **catch** που θα συλλάβει την εξαίρεση)!

Αυτό που γίνεται είναι το εξής: όταν εκτελεσθεί η **throw** το αντικείμενο της εξαίρεσης αντιγράφεται σε «σίγουρη» θέση όπου ζει μέχρι να τελειώσει η τελευταία **catch** που θα τη συλλάβει (και δεν θα την ξαναρίξει με "`throw;`"). Μετά την εκτέλεση αυτής της **catch** θα κληθεί ο καταστροφέας να καταστρέψει την εξαίρεση.

Αν το αντικείμενο της εξαίρεσης είναι μια σταθερά, όπως το "`-1`" που ρίχνει η *natProduct()*, ή ένα αντικείμενο χωρίς όνομα, όπως "`StudentXptn("Student", StudentXptn::noMemory)`", ο μεταγλωττιστής *μπορεί να κανονίσει* να μη χρειαστεί αντιγραφή (φυλάγοντας καταλλήλως το αντικείμενο της εξαίρεσης εξ αρχής).

Σε κάθε περίπτωση, τα παραπάνω βάζουν την εξής υποχρέωση στον προγραμματιστή που γράφει δικές του κλάσεις εξαιρέσεων:

---

[8] `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`

- ***Μια κλάση εξαιρέσεων θα πρέπει να έχει σωστό δημιουργό αντιγραφής και σωστό κατάστροφέα.***

Όσο η εξαίρεση «ταξιδεύει», κατά το «*ξετύλιγμα της στοίβας*», καλούνται καταστροφείς για να καταστρέψουν τα τοπικά αντικείμενα στις συναρτήσεις που κλείνουν. Όταν εκτελείται κάποιος από αυτούς τους καταστροφείς ή –κάποια συνάρτηση που καλείται από κάποιον καταστροφέα– μπορούμε να ελέγξουμε αν υπάρχει εξαίρεση που δεν έχει συλληφθεί ελέγχοντας την τιμή που επιστρέφει η συνάρτηση (*std*::)*uncaught_exception()*.

Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται αυτομάτως η (*std*::)*terminate()* (που καλεί την (*std*::)*abort()*) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_terminate()*– να αντικαταστήσεις την *terminate()* με δική σου.

Ας πούμε τώρα ότι είχες *προδιαγραφή εξαιρέσεων*:

```
unsigned long int natProduct( int m, int n ) throw( char* )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// . . .
} // natProduct
```

και εκτελείται η "`throw -1`". Στην περίπτωση αυτήν –και κάθε φορά που ρίχνεται εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση (*std*::)*unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Όπως στην περίπτωση της *terminate()*, η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_unexpected()*– να αντικαταστήσεις την (*std*::)*unexpected()* με δική σου.

24.4 Οι Συναρτήσεις Διαχείρισης Εξαιρέσεων

Θα δούμε τώρα τις συναρτήσεις που δίνει η C++[9] για τη διαχείριση εξαιρέσεων. Αλλά να τονίσουμε από την αρχή το εξής: *Αν σχεδιάσεις καλά το προγράμμά σου, αυτές οι συναρτησεις δεν θα σου χρειαστούν*. Αργότερα όμως –όταν οι τροποποιήσεις και οι προσαρμογές στις απαιτήσεις που αλλάζουν– δεν θα καλύπτονται από τις προβλέψεις του αρχικού σχεδίου, αυτές οι συναρτήσεις μπορεί να σε βοηθήσουν να κάνεις διάφορα «μπαλώματα».

24.4.1 Η Συνάρτηση `std::set_terminate()`

Λέγαμε στην προηγούμενη παράγραφο: «*… καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (std::)set_terminate()– να αντικαταστήσεις την terminate() με δική σου.*»

Φαντάσου τώρα ότι η *terminate()*, για την οποίαν τα λέμε στη συνέχεια, είναι κάτι σαν:

```
void terminate()
{
   terminate_handler pDth( &dth );
// . . .
   (*pDth)();
}
```

όπου

```
void dth()
{
// . . .
```

[9] Αναφερόμαστε στη C++03. Η C++11 δίνει και άλλες…

```
   abort();
}
```

και

```
typedef void (*terminate_handler)();
```

Δηλαδή, ο *terminate_handler* είναι τύπος βέλους προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την παράμετρο της *atexit()*).

Όπως καταλαβαίνεις, η *terminate()* καλεί την *abort()* εμμέσως, μέσω της *dth()*.

Η *set_terminate()* σου επιτρέπει να αλλάξεις τη συνάρτηση που καλεί η *terminate()* και – στο παράδειγμά μας– δείχνει το *pDth*. Για να τη χρησιμοποιήσεις θα θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <exception>`". Στο `exception` μπορείς να δεις την επικεφαλίδα της:

```
terminate_handler set_terminate(terminate_handler f) throw();
```

Ας πούμε ότι γράφουμε τη:

```
void myTerminateHandler()
{
   logFl << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
         << "Φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler
```

και μέσα στο πρόγραμμά μας δίνουμε:

```
   terminate_handler oldTH;
   oldTH = set_terminate( &myTerminateHandler );
```

Τι θα γίνει;

- Ο *pDth* της *terminate()* θα πάρει τιμή "`&myTerminateHandler`", τη διεύθυνση του νέου χειριστή τερματισμού.
- Η *oldTH* θα πάρει ως τιμή αυτήν που είχε πριν η *pDth*, θα δείχνει τον παλιό χειριστή τερματισμού.

Έτσι δουλεύει η *set_terminate()*.[10] Η τιμή που επιστρέφει η συνάρτηση μας είναι συνήθως άχρηστη (εκτός από την περίπτωση που θα θελήσεις να ξαναεγκαταστήσεις αργότερα τον αρχικό χειριστή) και το πιθανότερο είναι να δίνεις απλώς:

```
   set_terminate( &myTerminateHandler );
```

Δες ένα πρόγραμμα που δοκιμάζει τα παραπάνω:

```
#include <fstream>
#include <exception>
#include <cstdlib>

using namespace std;

ofstream logFl( "log.txt" );

void myTerminateHandler()
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   set_terminate( &myTerminateHandler );
   throw -1;
}
```

Γιατί βάλαμε τη συνάρτησή μας να γράφει σε αρχείο και όχι στην οθόνη; Για να κάνουμε φανερό ότι αν θέλουμε κάτι τέτοιο θα πρέπει να το κάνουμε με ένα καθολικό αντικείμενο, όπως είναι το ρεύμα *logFl*, αφού η συνάρτησή μας δεν (μπορεί να) έχει παραμέτρους.

---

[10] Τώρα καταλαβαίνεις γιατί λέγαμε «φαντάσου»: εδώ φαίνεται ότι έρχεται η *set_terminate()* να αλλάξει την τιμή μιας τοπικής μεταβλητής (*pDth*) της *terminate()*…

Αν θελήσεις να γράψεις δική σου συνάρτηση για τον χειρισμό του τερματισμού πρόσεξε το εξής:

- ***Η συνάρτηση χειρισμού τερματισμού δεν θα πρέπει να επιστρέφει ως συνήθως με "`return`" αλλά θα πρέπει να τερματίζει την εκτέλεση του προγράμματος, κατά προτίμηση με κλήση της abort().***

  Την *abort()* καλεί και η συνάρτηση που βάζει αρχικώς ο μεταγλωττιστής.

24.4.2 Η Συνάρτηση `std::terminate()`

Όπως λέγαμε: «*Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί, καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος.*» Αυτό το βλέπεις και στο παράδειγμα που δώσαμε για τη *set_terminate()*: Στη `main` υπάρχει η "`throw -1`" που θα εκτελεσθεί σίγουρα αλλά δεν υπάρχει **catch** για να τη συλλάβει.

Στη συνέχεια βλέπεις πολλές περιπτώσεις που η *terminate()* καλείται αυτομάτως. Πάντως μπορείς να βάλεις και εντολή κλήσης της στο πρόγραμμά σου αρκεί να βάλεις μια "`#include <exception>`":

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

int main()
{
   set_terminate( &myTerminateHandler );
   terminate();
}
```

Αποτέλεσμα:[11]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

Στο `exception` μπορείς να δεις δήλωση που μοιάζει με:

```
void terminate();
```

Στη συνέχεια παραθέτουμε όλες τις περιπτώσεις που καλείται αυτομάτως η *terminate()* για να διακόψει την εκτέλεση του προγράμματός σου. Πάντοτε η κλήση της γίνεται πριν από το «ξετύλιγμα της στοίβας» εκτός φυσικά από την δεύτερη περίπτωση (δεν βρέθηκε **catch** να συλλάβει την εξαίρεση). Η *std::terminate()* καλείται:[12]

- Όταν ο μηχανισμός διαχείρισης εξαιρέσεων, αφού δημιουργήσει το αντικείμενο της εξαίρεσης και πριν αυτό συλληφθεί από κάποια **catch**, καλέσει κάποια συνάρτηση που ρίχνει εξαίρεση. Ας πούμε ότι έχουμε:

```
struct Xptn
{
```

[11] Borland C++ v.5.5.

[12] Αντιγράφουμε από το πρότυπο…

```
   int c;
   Xptn( int ac=0 )  {  c = ac;  }
   Xptn( const Xptn& rhs ) {  throw -1;  }
}; // Xptn
```

και

```
int main()
{
   Xptn x;
   set_terminate( &myTerminateHandler );
   try {
      throw x;
   }
   catch( Xptn& )
   {  cout << "in catch( Xptn& )" << endl;  }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Αποτέλεσμα:[13]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Το αντικείμενο που ρίχνει η "`throw x`" είναι έτοιμο. Όταν γίνει προσπάθεια αντιγραφής σε «σίγουρη» θέση –όπως μάθαμε στην §24.3– καλείται ο δημιουργός αντιγραφής. Αυτός ρίχνει άλλη εξαίρεση (`throw -1`) και έτσι καλείται η *terminate()*.

- Όταν έχει ριχτεί μια εξαίρεση, το ξετύλιγμα της στοίβας έχει φτάσει στη **main** και η εξαίρεση δεν έχει συλληφθεί από κάποια **catch**. Ας πούμε ότι έχουμε την *Xptn* και τη *myTerminateHandler()* που είχαμε παραπάνω και ακόμη:

```
void f1()
{  throw Xptn(-1);  }

void f2()
{  f1();  }

int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f2();
   }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Το πρόγραμμα θα μας δώσει το ίδιο αποτέλεσμα που πήραμε και από το προηγούμενο παράδειγμα. Πώς εξηγείται; Η **main** καλεί την *f2()* και αυτή την *f1()* που ρίχνει εξαίρεση "`Xptn(-1)`". Με το «ξετύλιγμα της στοίβας» επιστρέφουμε στην *f2()* και στη συνέχεια στη **main**. Η εξαίρεση δεν συλλαμβάνεται από την "`catch(int&)`" και αφού δεν υπάρχει "`catch(Xptn&)`" ούτε "`catch(...)`" καλείται η *terminate()*.

- Όταν κατά τη διάρκεια του ξετυλίγματος κληθεί κάποιος καταστροφέας ο οποίος ρίχνει μιαν εξαίρεση. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  }
   ~C()  {  throw 0;  }
};
```

[13] gcc.

```
void f()
{  C lc( 7 );
   throw -1;  }
```

και

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f();
   }
   catch( int& )
   {  cout << "int exception" << endl;  }
}
```

Μετά την εκτέλεση της "`throw -1`" καλείται ο καταστροφέας της C για να καταστρέψει το *lc*. Αλλά αυτός ρίχνει εξαίρεση και έτσι καλείται η *terminate()*· το αποτέλεσμα είναι αυτό που είδαμε στα άλλα παραδείγματα.

- Όταν ριχτεί εξαίρεση κατά τη δημιουργία ή την καταστροφή ενός στατικού αντικειμένου. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  throw -1;  }
};

C gc;
```

Το *gc* είναι ένα καθολικό στατικό αντικείμενο. Με τη εκτέλεση της "`throw -1`" κατά τη δημιουργία του καλείται η *terminate()*.[14] Στην περίπτωση που έχουμε:

```
void f()
{
   static C c( 7 );
}

int main()
{
   set_terminate( &myTerminateHandler );
   f();
}
```

και πάλι, κατά τη δημιουργία του *c*, θα έχουμε κλήση της *terminate()* αλλά τώρα θα δούμε και τους στίχους από το γνωστό άσμα.

- Όταν κατά την εκτέλεση συνάρτησης που έχουμε καταχωρίσει με την *atexit()* ριχτεί εξαίρεση. Αλλάζουμε λίγο τη *bye()* που είδαμε στην §24.1.2:

```
void bye()
{  cout << "Bye bye!" << endl;
   throw -1;  }
```

και δοκιμάζουμε τη

```
int main()
{
   set_terminate( &myTerminateHandler );
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   exit( 0 );
}
```

Αποτέλεσμα:

```
Bye bye!
```

[14] Πριν αρχίσει η εκτέλεση της **main** και επομένως χωρίς να εκτελεσθεί η "`set_terminate( &myTerminateHandler )`".

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

- Όταν γίνει απόπειρα εκτέλεσης μιας "`throw;`" (που ξαναρίχνει την εξαίρεση που χειριζόμαστε) ενώ δεν γίνεται διαχείριση κάποιας εξαίρεσης, π.χ.:

```
int main()
{
   set_terminate( &myTerminateHandler );
   throw;
}
```

- Από την *unexpected()*, όπως θα δούμε παρακάτω.

Από τα παραπάνω βγαίνουν και δύο κανόνες προγραμματιστικής πρακτικής· τον πρώτο τον έχουμε ήδη διατυπώσει:

♦ ***Οι καταστροφείς δεν επιτρέπεται να ρίχνουν εξαιρέσεις.***

Θα πεις «δεν μπορώ να φανταστώ περίπτωση που θα έβαζα μια εντολή `throw` σε έναν καταστροφέα.» Σωστό βέβαια, αλλά πρόσεχε αν ο καταστροφέας καλεί άλλες συναρτήσεις. Στην περίπτωση αυτή, ο καταστροφέας θα πρέπει να συλλαμβάνει όλες τις εξαιρέσεις και να σταματάει την μετάδοσή τους.

Τον δεύτερο τον τηρούμε παρ' όλο που δεν τον διατυπώσαμε:

♦ ***Αν γράφεις δικές σου κλάσεις εξαιρέσεων καλό είναι να μη ρίχνονται εξαιρέσεις όχι μόνο από τους καταστροφείς τους αλλά και από τους δημιουργούς τους.***

24.4.3 * Προδιαγραφές Εξαιρέσεων και Σχετικές Συναρτήσεις

Η (*std*::)*unexpected()* καλείται αυτομάτως όταν μια συνάρτηση –που έχει προδιαγραφή εξαιρέσεων– ρίξει εξαίρεση που ο τύπος της δεν υπάρχει στη λίστα τύπων της προδιαγραφής. Η (*std*::)*set_unexpected()* σου δίνει τη δυνατότητα να αλλάζεις την *unexpected()*.

Οι προδιαγραφές εξαιρέσεων έχουν ταλαιπωρήσει αρκετά τους προγραμματιστές που δοκίμασαν να τις χρησιμοποιήσουν. Οι γνώμες ενάντια στο εργαλείο αυτό ήταν πολλές και στο C++11 αποθαρρύνεται η χρήση τους εκτός από την περίπτωση "`throw()`" (η συνάρτηση δεν ρίχνει εξαίρεση). Έτσι λοιπόν και εμείς δίνουμε τη συμβουλή:

♦ ***Μη χρησιμοποιείς προδιαγραφές εξαιρέσεων εκτός από τη "`throw()`" και κατά συνέπεια τις συναρτησεις* unexpected() *και* set_unexpected().**

Πάντως στη συνέχεια θα δούμε εν συντομία τις δύο συναρτήσεις και τη χρήση της κλάσης (*std*::)*bad_exception*.

Ας πούμε ότι έχουμε το πρόγραμμα:

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

unsigned long int natProduct( int m, int n ) throw( char* )
{
   throw -1;
} // natProduct
```

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
}
```

Αποτέλεσμα:

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Τι έγινε; Η *natProduct()* έριξε εξαίρεση με αντικείμενο "`-1`" (`int`) εκτός προδιαγραφών εξαιρέσεων. Κλήθηκε η *unexpected()* και αυτή με τη σειρά της κάλεσε την *terminate()*.

Τώρα κάνουμε τις εξής αλλαγές:

- Γράφουμε την:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   throw;
}
```

- Αλλάζουμε την προδιαγραφή εξαιρέσεων της *natProduct()*:

```
unsigned long int natProduct( int m, int n ) throw( char*, bad_exception )
{
   throw -1;
} // natProduct
```

- Τέλος, αλλάζουμε και τη **main**:

```
int main()
{
   set_terminate( &myTerminateHandler );
   set_unexpected( &myUnexpected );      // <<<<<<<<<<<<<<
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
   catch( bad_exception& )               // <<<<<<<<<<<<<<
   {  cout << "catching bad_exception" << endl;  }
}
```

Αποτέλεσμα:

```
   unexpected; ασ' το σε μένα...
   catching bad_exception
```

Βάζοντας στη λίστα τύπων εξαιρέσεων τον (*std*::)*bad_exception* και γράφοντας τη δική μας *myUnexpected()*, γλυτώσαμε από την *terminate()*!

24.4.4 * Η Συνάρτηση `std::uncaught_exception()`

Η επικεφαλίδα της συνάρτησης, όπως μπορείς να τη δεις στο `exception`, είναι:

```
bool uncaught_exception() throw();
```

Η συνάρτηση αυτή επιστρέφει τιμή "`true`" αν, όταν κληθεί, υπάρχει εξαίρεση που δεν έχει συλληφθεί.

Ας δούμε ένα παράδειγμα χρήσης. Η *myUnexpected()* που γράψαμε παραπάνω έχει το εξής πρόβλημα: Ενώ μπορεί να κληθεί κατ' ευθείαν από το πρόγραμμά μας, σε τέτοια περίπτωση, αν δεν υπάρχει κάποια εξαίρεση «στον αέρα», η "`throw;`" θα προκαλέσει την (αυτοματη) κλήση της *terminate()*. Η σωστή *myUnexpected()* είναι:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   if ( uncaught_exception() )
      throw;
}
```

Πού αλλού θα μπορούσε να χρησιμοποιηθεί αυτή η συνάρτηση; Για να την απαντησουμε θα πρέπει να σκεφτούμε ποιες συναρτήσεις ενεργοποιούνται όταν επιστρέφει "`true`" αυτή η συνάρτηση, δηλαδή από τη στιγμή που ρίχτηκε μια εξαίρεση μέχρι να συλληφθεί. Τι γίνεται στο διάστημα αυτό; «Κλείνουν» διάφορες συναρτήσεις και καλούνται καταστροφείς για να καταστρέψουν τα αντικείμενα που ζούσαν μέσα σε αυτές. Επομένως;... Αυτό που κατάλαβες! Μπορείς να τη χρησιμοποιήσεις μόνο μέσα σε καταστροφείς (ή σε συναρτήσεις που αυτοί καλούν). Και πώς θα τη χρησιμοποιήσεις; Θα γράφεις κάτι σαν:

```
   if ( uncaught_exception() )
   {  E1  }
   else
   {  E2  }
```

που σημαίνει: αν έχουμε εξαίρεση «στον αέρα» θα καταστρέψεις το αντικείμενο με τις *E1* αλλιώς θα το καταστρέψεις με τις *E2* (που μπορεί να ρίχνουν και καμιά εξαίρεση).

Ένας προγραμματιστής που σέβεται τον εαυτό του δεν θα γράψει κάτι τέτοιο ακόμη και στην πιο απελπισμένη προσπάθεια να «μπαλώσει» ένα προβληματικό πρόγραμμα.

24.5 Συναρτησιακή Ομάδα try

Στην §23.3 γράψαμε τον δημιουργό της *DateTime* ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
// . . .
}; // DateTime::DateTime
```

Αν ο δημιουργός της *Date* ρίξει εξαίρεση εμείς θα πάρουμε το μήνυμα ότι κάποιο πρόβλημα βρήκε ο δημιουργός της *Date*· αυτό όμως είναι ψέμα ή –τουλάχιστον– μισή αλήθεια: Το πρόβλημα βρήκε ο δημιουργός της *Date όταν κλήθηκε από τον δημιουργό της DateTime*.

Τι μπορούμε να κάνουμε για να το διορθώσουμε; Να πιάσουμε την εξαίρεση *DateXptn* από τον δημιουργό της *Date* και να ρίξουμε νέα εξαίρεση *DateTimeXptn*. Αυτό μπορεί να γίνει ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}
catch( DateTimeXptn& x )
```

```
{  throw;  }
catch( DateXptn& x )
{
   throw DateTimeXptn( "DateTime", x.errorCode,
                       x.errVal1, x.errVal2, x.errVal3 );
} // DateTime::DateTime
```

Τι έχουμε εδώ;

- Η ομάδα της **try** ταυτίζεται με το σώμα της συνάρτησης· από εδώ και το όνομα **συναρτησιακή ομάδα try** (functional try-block). Οι **catch** ανήκουν στη συνάρτηση παρ' όλο που φαίνεταο να είναι έξω από αυτήν.
- Η λίστα εκκίνησης ελέγχεται από την **try**, παρ' όλο που βρίσκεται πριν από το "`{`".
- Η πρώτη **catch** πιάνει τις εξαιρέσεις που ρίχνονται από τον ίδιο τον δημιουργό και τις ξαναρίχνει χωρίς άλλη ενέργεια ("`throw;`").
- Η δεύτερη **catch** πιάνει οποιαδήποτε εξαίρεση ριχτεί από τη λίστα εκκίνησης –στην περίπτωσή μας από την κλήση του δημιουργού της βασικής– και την ξαναρίχνει αφού τη μετατρέψει σε *DateTimeXptn*.

Αμφότερες οι ομάδες **catch** τελειώνουν με `throw`. Σύμπτωση; Όχι!

♦ ***Αν έχεις συναρτησιακή ομάδα try σε δημιουργό όλες οι ομάδες catch που αντιστοιχούν σε αυτήν πρέπει να τελειώνουν με `throw`. Αν δεν υπάρχει τότε ο μεταγλωττιστής θα βάλει ερήμην σου μια "`throw;`".***

Φυσικά, αυτό δεν είναι περίεργο: το ότι ήλθε κάποια εξαίρεση σημαίνει ότι δεν είναι δυνατή η δημιουργία κάποιου μέλους του αντικειμένου (ή του υποαντικειμένου της βασικής κλάσης). Επομένως δεν είναι δυνατόν να δημιουργηθεί το αντικείμενο!

Αυτά που ισχύουν για συναρτησιακή ομάδα **try** *σε δημιουργό ισχύουν και για τον καταστροφέα.* Αφού όμως έχουμε εξηγήσει ότι από τον καταστροφέα δεν επιτρέπεται να ξεφεύγουν εξαιρέσεις, η δυνατότητα να βάλουμε συναρτησιακή ομάδα **try** σε καταστροφέα μας είναι τελείως άχρηστη!

Μπορείς να βάλεις συναρτησιακή ομάδα **try** σε οποιαδήποτε συνάρτηση εκτός από τη **main**. Στην περίπτωση αυτή έχουμε μια διαφορά από αυτά που ισχύουν για δημιουργούς και καταστροφείς: *δεν υπάρχει υποχρέωση για* `throw` *στις αντίστοιχες ομάδες* **catch**. Αλλά και αυτή η δυνατότητα μας είναι άχρηστη αφού

Η

```
void f( . . . )
try {
// . . .
}
catch( . . . )
{
// . . .
}
```

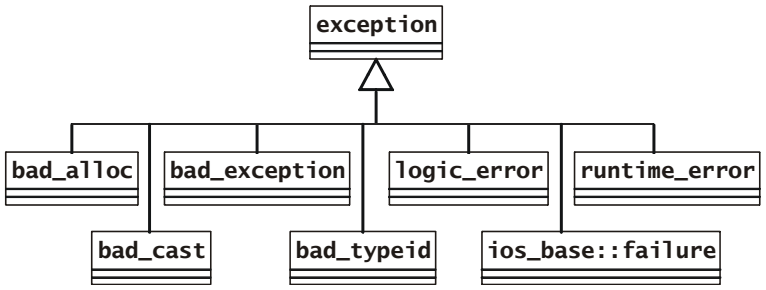
είναι ισοδύναμη με

```
void f( . . . )
{
   try
   {
   // . . .
   }
   catch( . . . )
   {
   // . . .
   }
}
```

24.6 Οι Τύποι Εξαιρέσεων της C++

Η C++ έχει μερικές κλάσεις εξαιρέσεων για χρήση στις βιβλιοθήκες της· οι κλάσεις αυτές είναι διαθέσιμες και στους προγραμματιστές. Σύμφωνα με το πρότυπο της γλώσσας, η κλάση αυτή, που δηλώνεται στο `exception`, πρέπει να είναι:

```
class exception
{
public:
   exception() throw();
   exception( const exception& ) throw();
   exception& operator=( const exception& ) throw();
```



Σχ. 24-1 Η *exception* και οι επτά κλάσεις εξαιρέσεων που την κληρονομούν. Οι *logic_error* και *runtime_error* έχουν δικές τους κλάσεις-κληρονόμους.

```
   virtual ~exception() throw();
   virtual const char* what() const throw();
};
```

Όπως βλέπεις, θα πρέπει να έχει:

- ερήμην δημιουργό,
- δημιουργό αντιγραφής,
- τελεστή εκχώρησης,
- καταστροφέα (`virtual`) και
- μια μέθοδο (`virtual`), τη *what()*, που επιστρέφει ένα κείμενο.

Τα πάντα έχουν το χαρακτηριστικό "`throw()`": *δεν ρίχνουν εξαιρέσεις!* Στην §16.6.1 λέγαμε ότι, στις δικές μας κλάσεις εξαιρέσεων, έπρεπε να βάλουμε "`char funcName[100]`" και όχι "`string funcName`" για να έχουμε τη σιγουριά ότι δεν θα ριχτεί εξαίρεση όταν καλείται ο δημιουργός για να δημιουργήσει ή να αντιγράψει ένα αντικείμενο εξαίρεσης. Το πρότυπο λέει: «*Για παράδειγμα, αν το αντικείμενο [εξαίρεσης] είναι κλάσης με δημιουργό αντιγραφής θα κληθεί η* `std::terminate()` *αν η εκτέλεση του δημιουργού αντιγραφής διακοπεί με εξαίρεση όταν εκτελείται μια* `throw`.»

Τι είναι η *what()*; Μια μέθοδος που επιστρέφει ένα κείμενο-μήνυμα. Για παράδειγμα στη Borland C++ v.5.5 το μήνυμα λέει:

```
no named exception thrown
```

Αυτό που είδαμε παραπάνω δεν είναι δήλωση κλάσης· μπορείς να το θεωρήσεις ως διεπαφή.[15] Η κλάση *exception* δεν χρησιμοποιείται στα προγράμματα. Χρησιμοποιούνται οι παράγωγές της που τις βλέπεις στο Σχ. 24-1.

Η πρώτη παράγωγη που ήδη χρησιμοποιούμε είναι η (*std*::)*bad_alloc*. Το πρότυπο λέει:

```
class bad_alloc : public exception
{
public:
   bad_alloc() throw();
   bad_alloc( const bad_alloc& ) throw();
   bad_alloc& operator=( const bad_alloc& ) throw();
   virtual ~bad_alloc() throw();
   virtual const char* what() const throw();
};
```

Όπως βλέπεις, είναι ολόιδια με τη βασική. Βέβαια, εδώ η *what()* βγάζει άλλο μήνυμα (BC++ v.5.5):

```
bad alloc exception thrown
```

Στη C++ που χρησιμοποιείς θα βρεις τη δήλωσή της στο `new`.

Παρόμοια ισχύουν και για τις άλλες παράγωγες κλάσεις, εκτός από την `ios_base::failure`.

[15] Αν ψάξεις στο `exception` μπορεί να βρεις πώς το ολοκληρώνει ως κλάση η C++ που χρησιμοποιείς.

Γνωρίσαμε τη (*std*::)*bad_cast* στην §23.13.1 και στην §Prj06.4. Η δήλωσή της στο `typeinfo`.

Στην §24.4.3 είδαμε την (*std*::) *bad_exception*. Η δήλωσή της στο `exception`.

Η (*std*::)*bad_typeid* ρίχνεται από την *typeid* αν της δώσεις όρισμα (αποπαραπομπή σε) βέλος "`0`". Για να τη χρησιμοποιήσεις πρέπει να βάλεις "`#include <typeinfo>`". Για παράδειγμα, οι:

```
   Student* p( 0 );
   try {
      cout << typeid(*p).name() << endl;
// . . .
   }
   catch ( bad_typeid& x )
   {
      cout << "bad_typeid caught" << endl;
      cout << x.what() << endl;
   }
```

θα δώσουν (g++):

```
bad_typeid caught
St10bad_typeid
```

Εξαιρέσεις κλάσης (*std*::)*ios_base*::*failure* ρίχνονται όταν αποτύχουν μερικές πράξεις διαχείρισης ρευμάτων.[16] Ο ορισμός της κλάσης έχει την εξής διαφορά από τις άλλες κλάσεις που είδαμε: έχει έναν ακόμη δημιουργό

```
   explicit failure( const string& msg );
```

με τον οποίον εισάγουμε (τιμή του *msg*) το κείμενο που μας δίνει η *what()*.

24.6.1 Η Κλάση *logic_error* και οι Παράγωγές της

Η *logic_error* ορίζεται στο `stdexcept`:

```
class logic_error : public exception
{
public:
   explicit logic_error( const string& what_arg );
};
```

Δηλαδή, εκτός από αυτά που έχει η βασική υπάρχει ένας επιπλέον δημιουργός όπως αυτός που είδαμε στην *ios_base*::*failure*.

Όπως λέει το πρότυπο, «*η κλάση logic_error ορίζει τον τύπο των αντικειμένων που ρίχνονται ως εξαιρέσεις για να αναφέρουν σφάλματα υποθετικώς ανιχνεύσιμα πριν από την εκτέλεση του προγράμματος, όπως παραβιάσεις λογικών προϋποθέσεων ή αναλλοίωτων κλάσεων.*»

24.6.1.1 Η Κλάση *domain_error*

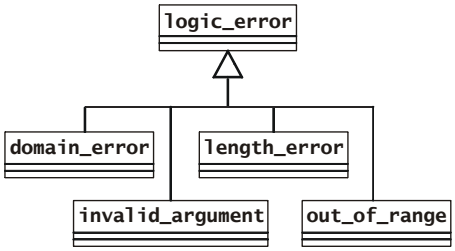
Η πρώτη παράγωγη κλάση είναι η:

```
class domain_error : public logic_error
{
public:
   explicit domain_error( const string& what_arg );
};
```

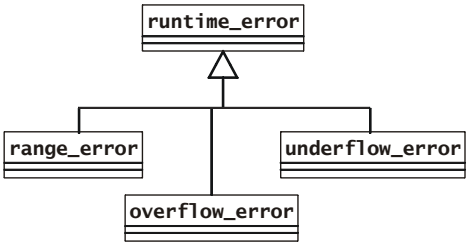
Τη ρίχνουμε όταν μια συνάρτηση καλείται με ορίσματα έξω από το **πεδίο ορισμού** (domain of definition) της. Θα μπορούσαμε να γράψουμε τη *factorial()* που είδαμε στις *Εισαγωγικές Παρατηρήσεις* ως εξής:

---

[16] Στο C++11 η κλάση αυτή είναι παράγωγη της *system_error* που, με τη σειρά της, είναι παράγωγη της *runtime_error*.



Σχ. 24-2 Η *logic_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.



Σχ. 24-3 Η *runtime_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.

```
unsigned long int factorial( int a )
{
   if ( a < 0 )
   {
      ostringstream ssout;
      ssout << "η factorial κλήθηκε με όρισμα " << a;
      throw domain_error( ssout.str() );
   }
   unsigned long int fv( 1 );
   for ( int k(1);  k <= a;  ++k )  fv *= k;
   return fv;
} // factorial
```

Δημιουργούμε το αντικείμενο της εξαίρεσης με τον δημιουργό της κλάσης συναρμολογώντας το μήνυμα με ένα ρεύμα κλάσης *ostringstream* (§10.12).

Παράδειγμα χρήσης:

```
#include <iostream>
#include <stdexcept>  // για τη domain_error
#include <sstream>    // για το ostringstream

using namespace std;

unsigned long int factorial( int a )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main ()
{
   try
   {
      cout << factorial( 7 ) << endl;
      cout << factorial( -7 ) << endl;
   }
   catch ( domain_error& x )
   {
      cout << "domain error: " << x.what() << endl;
   }
}
```

Αποτέλεσμα:

```
5040
domain error: η factorial κλήθηκε με όρισμα -7
```

24.6.1.2 Η Κλάση *invalid_argument*

Ορίζεται στο `stdexcept` ως:

```
class invalid_argument : public logic_error
{
public:
   explicit invalid_argument( const string& what_arg );
};
```

Ένα «προβληματικό» όρισμα συνάρτησης συνήθως έχει τιμή εκτός πεδίου ορισμού και το αντιμετωπίζουμε με "`domain_error`". Αν έχει άλλο πρόβλημα χρησιμοποιούμε την "`invalid_argument`".

24.6.1.3 Η Κλάση *length_error*

Ορίζεται στο `stdexcept` ως:

```
class length_error : public logic_error
{
public:
   explicit length_error( const string& what_arg );
};
```

Θα μπορούσε να ονομάζεται και "`size_error`". Τη ρίχνουμε όταν το «μέγεθος» (ό,τι και αν είναι αυτό) κάποιου αντικειμένου υπερβαίνει κάποιο όριο που βάζει η υλοποίηση (ή το πρόγραμμα εφαρμογής). Για παράδειγμα, αν έχουμε δηλώσει:

```
   string s1;
```

οι εντολές:

```
   try {
      s1.reserve( s1.max_size()+1 );
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):[17]

```
std::length_error
invalid string size parameter in function: basic_string::reserve(size_t)
 size: -14 is greater than maximum size: -15
```

24.6.1.4 Η Κλάση *out_of_range*

Ορίζεται στο `stdexcept` ως:

```
class out_of_range : public logic_error
{
public:
   explicit out_of_range( const string& what_arg );
};
```

Θα τη δούμε με ένα παράδειγμα· αν έχουμε:

```
   s1( "test text" );
```

οι εντολές:

```
   try {
      cout << s1.at(17) << endl;
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):

```
std::out_of_range
position beyond end of string in function: basic_string::at(size_t)
 index: 17 is greater than max_index: 9
```

Στην πραγματικότητα, τουλάχιστον στην περίπτωση αυτή, πρόκειται για ειδική περίπτωση του "`domain_error`".

---

[17] Τι είναι τα "`-14`" και "`-15`"; Δεν ξέρεις; Κεφ. 17!

24.6.2 Η Κλάση *runtime_error* και οι Παράγωγές της

Η *runtime_error* ορίζεται στο `stdexcept`:

```
class runtime_error : public exception
{
public:
   explicit runtime_error( const string& what_arg );
};
```

Όπως βλέπεις, κληρονομεί την *exception* με τον ίδιο τρόπο που την κληρονομεί και η *logic_error*.

Σε τι διαφέρει από τη *logic_error*; Η *logic_error* (ή κάποια παράγωγή της) ρίχνεται όταν ξεκινάμε έναν υπολογισμό με τιμή(-ές) κάποιας(-ων) παραμέτρου (-ων) που δεν είναι αποδεκτή(-ές). Η *runtime_error* ρίχνεται κατά τη διάρκεια του υπολογισμού, όταν κάποιο αποτέλεσμα –τελικό ή ενδιάμεσο– έχει κάποιο πρόβλημα.

24.6.2.1 Η Κλάση *range_error*

Η *range_error* είναι παράγωγη της *runtime_error*:

```
class range_error : public runtime_error
{
public:
   explicit range_error( const string& what_arg );
};
```

Ρίχνουμε εξαίρεση αυτού του τύπου όταν έχουμε τιμή συνάρτησης εκτός πεδίου τιμών. Αυτό έχει συχνά να κάνει με τη διαφορά που υπάρχει μεταξύ του πεδίου τιμών της συνάρτησης και του τύπου που παριστάνει αυτό το σύνολο στον υπολογιστή. Για παράδειγμα, αν το πεδίο τιμών μιας συνάρτησης *f* είναι το $\mathbb{R}$,

$f: D \to \mathbb{R}$

όταν την υλοποιήσουμε με την f_C θα το παραστήσουμε με τον `float` ή τον `double` ή τον `long double`:

$f_C: D_C \to$ `float` (ή `double` ή `long double`) (C)

Έτσι όμως κάποιες τιμές της συνάρτησης που είναι στο $\mathbb{R}$ μπορεί να μην είναι δυνατόν να παρασταθούν στον τύπο που επιλέξαμε, δηλαδή έχουμε *υπερχείλιση*. Με βάση τη (C) έχουμε πρόβλημα πεδίου τιμών.

Χαρακτηριστικό παράδειγμα είναι η συνάρτηση που *addInt()* που γράψαμε στην §17.3 για να αντιμετωπίσουμε την «ύπουλη» υπερχείλιση στις προσθέσεις ακεραίων. Το σύνολο των ακεραίων $\mathbb{Z}$ είναι κλειστό ως προς την πρόσθεση: το άθροισμα ακεραίων είναι πάντοτε ακέραιος. Αλλά το άθροισμα δύο τιμών τύπου `int` δεν είναι σίγουρο ότι μπορεί να παρασταθεί στον `int`. Εκεί θα μπορούσαμε να ρίχνουμε εξαίρεση τύπου *range_error* αντί για *IntOvrflXptn*.

24.6.2.2 Οι Κλάσεις *overflow_error* και *underflow_error*

Στην *addInt()* θα μπορούσαμε να ρίχνουμε και εξαίρεση κλάσης *overflow_error* που είναι και αυτή παράγωγη της *runtime_error*:[18]

```
class overflow_error : public runtime_error
{
public:
   explicit overflow_error( const string& what_arg );
};
```

Για τις πράξεις στους τύπους κινητής υποδιαστολής υπάρχει και το πρόβλημα της *υποχείλισης* για την οποίαν λέγαμε στην §17.9 «*Συνήθως, ο υπολογιστής σε μια τέτοια περί-*

[18] Πάντως, για ακέραιους τύπους προτιμούμε τον όρο "range error"· ο όρος *υπερχείλιση* αναφέρεται συνήθως σε τύπους κινητής υποδιαστολής.

πτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!»[19] Η κλάση

```
class underflow_error : public runtime_error
{
public:
   explicit underflow_error( const string& what_arg );
};
```

μας δίνεται για τέτοιες περιπτώσεις.

Για να δούμε ένα παράδειγμα χρήσης αυτών των κλάσεων γράφουμε μια *multDbl()* που –ακολουθώντας την *addInt()*– πολλαπλασιάζει δύο τιμές τύπου **double** αλλά έχει πρόβλεψη για υπερχείλιση και υποχείλιση:

```
double multDbl( double x, double y )
{
   double fv;

   if ( fabs(x) >= 1 )
   {
      if ( fabs(y) < 1 )  fv = x * y;
      else if ( fabs(y) <= DBL_MAX / fabs(x) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw overflow_error( ssout.str() );
      }
   }
   else    // |x| < 1
   {
      if ( fabs(y) >= 1 )  fv = x * y;
      else if ( fabs(x) >= DBL_MIN / fabs(y) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw underflow_error( ssout.str() );
      }
   }
   return fv;
} // multDbl
```

Αν δώσεις:

```
  try
  {
     cout << multDbl( DBL_MAX, DBL_MAX ) << endl;
  }
  catch( exception& x )
  {
     cout << typeid(x).name() << endl << x.what() << endl;
  }
```

θα πάρεις (g++, Dev-C++):

```
St14overflow_error
η multDbl κλήθηκε με ορίσματα 1.79769e+308, 1.79769e+308
```

Αν δώσεις:[20]

```
      cout << multDbl( DBL_MIN, DBL_MIN ) << endl;
```

---

[19] Αλλά στην §17.15 δίνουμε δύο παραδείγματα για το πώς να *αποφύγουμε* την υποχείλιση με περισσότερη προσοχή στους υπολογισμούς!

[20] Αν δώσεις απλώς "`cout << DBL_MIN*DBL_MIN << endl`" θα πάρεις "`0`".

θα πάρεις (g++, Dev-C++):

```
St15underflow_error
η multDbl κλήθηκε με ορίσματα 2.22507e-308, 2.22507e-308
```

24.6.3 Να Χρησιμοποιούμε Αυτές τις Κλάσεις;

Όπως είδες, οι κλάσεις εξαιρέσεων της C++ έχουν διαφορετική κατηγοριοποίηση από αυτές που χρησιμοποιούμε εμείς:

- Υπάρχει μια βασική κλάση, η *exception*.
- Από αυτήν παράγονται άλλες κλάσεις για τα διάφορα είδη προβλημάτων.
- Η πληροφορία για το είδος του προβλήματος υπάρχει στο όνομα της κλάσης και οποιαδήποτε άλλα σχετικά στοιχεία διαβιβάζονται μέσω της μεθόδου *what()*.

Αν σε βολεύουν μπορείς να τις χρησιμοποιείς και ακόμη μπορείς να ορίσεις δικές σου παράγωγες κλάσεις για πιο εξειδικευμένα προβλήματα.

24.7 Πώς να Σχεδιάζεις Δικές σου Κλάσεις Εξαιρέσεων

Στις σελίδες των βιβλιοθηκών Boost μπορείς να βρεις οδηγίες για τη διαχείριση λαθών και εξαιρέσεων (Abrahams 2010). Μεταξύ αυτών υπάρχουν και συμβουλές για το πώς να γράφεις δικές σου κλάσεις εξαιρέσεων. Να μια περίληψη:

- **Κάνε τις κλάσεις εξαιρέσεων παράγωγες της `std::exception`**.

  Πρόσεξε τα παραδείγματα που δώσαμε στις προηγούμενες παραγράφους: βάζουμε "`catch(exception& x)`" και με την "`typeid(x).name()`" βρίσκουμε τον τύπο της εξαίρεσης. Έτσι, αν ξέρεις ότι όλες οι κλάσεις εξαιρέσεων του προγράμματός σου είναι παράγωγες της *std*::*exception* μπορείς να χρησιμοποιείς την "`catch(exception& x)`" αντί για την "`catch(...)`".
- **Χρησιμοποίησε εικονική κληρονομιά** ώστε οι κλάσεις εξαιρέσεων να μπορούν να κληρονομούν δύο ή περισσότερες κλάσεις χωρίς πρόβλημα.

  Ε, τώρα, μη πάρεις και πολύ στα σοβαρά αυτή τη συμβουλή… «*Κλάση εξαιρέσεων που να κληρονομεί δύο ή περισσότερες κλάσεις εξαιρέσεων*»;! Αν τύχει να σου χρειαστεί κάτι τέτοιο μην ξεχάσεις να βάλεις το "`virtual`"!
- **Μη βάζεις στην κλάση εξαιρέσεων μέλη κλάσης `std::string` ή άλλης κλάσης με δημιουργό που μπορεί να ρίξει εξαίρεση**.

  Παρόμοια συμβουλή δώσαμε στην §24.4.2.
- **Μορφοποίησε το μήνυμα της `what()` (μόνο) αν ζητηθεί.** Γενικώς η μορφοποίηση χρειάζεται μνήμη και χρόνο και… μπορεί να φύγει και κάποια εξαίρεση.
- **Μην ασχολείσαι πολύ με το μήνυμα της `what()`**. Είναι καλό βέβαια να δώσεις στον προγραμματιστή τη δυνατότητα να καταλάβει τι συμβαίνει, αλλά είναι μάλλον απίθανο να μπορέσεις να συνθέσεις ένα μήνυμα με τέτοια «προσόντα» όταν ρίχνεις την εξαίρεση.
- **Δώσε πληροφορίες στη διεπαφή (public) της κλάσης εξαιρέσεων για την αιτία του σφάλματος**. Η προσήλωση στο μήνυμα της `what()` πιθανότατα σημαίνει ότι αμελείς την παρουσίαση (χρήσιμων) πληροφοριών.
- **Δώσε στην κλάση εξαιρέσεων «ανοσία» σε διπλή καταστροφή**. Δυστυχώς μερικοί μεταγλωττιστές, σε ορισμένες περιπτώσεις, θα βάλουν εντολές που καταστρέφουν δύο

φορές το αντικείμενο της εξαίρεσης. Αν η κλάση εξαιρέσεων έχει βέλη μην ξεχνάς, στον καταστροφέα, μετά τη "`delete`" να βάλεις και τιμή "`0`" στο κάθε βέλος.[21]

24.8 Οι Δικές μας Κλάσεις Εξαιρέσεων

Στις κλάσεις εξαιρέσεων που χρησιμοποιούμε στα μαθήματά μας έχουμε διαφορετική κατηγοριοποίηση από αυτήν που βλέπουμε στις κλάσεις της C++:

- Μια κλάση εξαιρέσεων για κάθε κλάση. Και ακόμη: αν η κλάση *D* είναι παράγωγη της κλάσης *B* τότε και η κλάση εξαιρέσεων της *D* είναι παράγωγη της κλάσης εξαιρέσεων της *B*.
- Μια κλάση εξαιρέσεων για κάθε βιβλιοθήκη.
- Μια κλάση εξαιρέσεων για κάθε πρόγραμμα.

  Ακολουθούμε τις συμβουλές της προηγούμενης παραγράφου;
- Μέχρι τώρα οι κλάσεις μας δεν είναι παράγωγες της *std*::*exception*. Από εδώ και πέρα θα είναι! Για παράδειγμα, οι κλάσεις εξαιρέσεων του Project 6 θα είναι:

```
struct CourseXptn :  public exception { /* . . . */ };
struct CourseCollectionXptn :  public exception { /* . . . */ };
struct StudentXptn :  public exception { /* . . . */ };
struct StudentCollectionXptn :  public exception { /* . . . */ };
struct StudentInCourseXptn :  public exception { /* . . . */ };
struct StudentInCourseCollectionXptn :  public exception { /* . . . */ };
struct MyTmpltLibXptn :  public exception { /* . . . */ };
struct ProgXptn :  public exception { /* . . . */ };
```

  Σε όλες αυτές τις κλάσεις θα πρέπει να βάλεις και τον υπερισχύοντα κατάστροφέα, π.χ. για την *CourseXptn*:

```
virtual ~CourseXptn() throw() { };
```

  Αυτό θα πρέπει να επαναληφθεί και για την *OfferedCourseXptn* που κληρονομεί την *CourseXptn*:

```
virtual ~OfferedCourseXptn() throw() { };
```

  Ακόμη, στη **main()**, αντί για την "`catch(...)`" βάζουμε:

```
catch( exception& x )
{
   cout << "unexpected exception " << typeid(x).name() << endl;
}
```

- Έχουμε επιλέξει να μην βάζουμε στις κλάσεις εξαιρέσεων
  - μέλη-βέλη προς δυναμικές μεταβλητές και
  - μέλη κλάσεων που να χρειάζονται δημιουργούς αντιγραφής.
- Το μορφοποιημένο μήνυμα προς τον χρήστη συντίθεται μετά την σύλληψη του αντικειμένου της εξαίρεσης. Με τη *what()* δεν ασχολούμαστε.
- Όταν ρίχνουμε μια εξαίρεση φροντίζουμε στο αντικείμενό της να υπάρχει
  - το όνομα της συνάρτησης από όπου ρίχτηκε η εξαίρεση και
  - οι «ένοχες» τιμές που την προκάλεσαν.
  - Ακόμη, όταν η εξαίρεση ρίχνεται από μέθοδο ενός αντικειμένου έχουμε και το κλειδί του αντικειμένου.

  Έτσι, δίνουμε αρκετές πληροφορίες για τον εντοπισμό και τη διόρθωση λαθών του προγράμματος. Δίνουμε επίσης τα δεδομένα για διορθωτικές πράξεις κατά τη διάρκεια της εκτέλεσης του προγράμματος, αν κάτι τέτοιο είναι δυνατό.

---

[21] Αυτή η συμβουλή αμφισβητείται διότι δεν φαίνεται να υπάρχουν –τώρα πια– μεταγλωττιστές που κάνουν «διπλή καταστροφή» του αντικειμένου της εξαίρεσης.

Αυτά σε σχέση με τις συμβουλές της προηγούμενης παραγράφου. Θα πρέπει όμως να βελτιώσουμε περισσότερο τις κλάσεις εξαιρέσεων. Ήδη στην §19.4 είχαμε επισημάνει:

- *«Η εξαίρεση θα μας φέρει τη μέθοδο που εμφανίστηκε το πρόβλημα, το είδος του προβλήματος και τις τιμές που το προκάλεσαν … αλλά δεν θα μας πει σε ποιο αντικείμενο έγιναν όλα αυτά. ... Μια πρώτη σκέψη είναι να αντιγράφουμε στην εξαίρεση και το κλειδί του αντικειμένου που έχει το πρόβλημα. Αλλά πολύ συχνά αυτό δεν είναι αρκετό. … Αν εξοπλίσουμε κάθε αντικείμενο με ένα επιπλέον μέλος, μια ταυτότητα αντικειμένου που θα την αντιγράφουμε και στο αντικείμενο της εξαίρεσης λύνουμε το πρόβλημά μας.»*
- *«Το δεύτερο πρόβλημα μπορεί να το έχεις αντιμετωπίσει ήδη αν έγραψες κάπως μεγάλα προγράμματα: "Ναι, η εξαίρεση ρίχτηκε από τη συνάρτηση myFunc, αλλά η myFunc καλείται στο πρόγραμμά μου σε 37 διαφορετικές «διαδρομές» εκτέλεσης!"»*

Ήδη, στο Project 6, είδαμε αντικείμενα εξαιρέσεων με το κλειδί του αντικειμενου· πάντως, όπως υποσχεθήκαμε και στην §19.4 *δεν θα δώσουμε παράδειγμα με ταυτότητα αντικειμένου.*

24.8.1 Δύο Μέθοδοι για τις Κλάσεις Εξαιρέσεων

Θα ασχοληθούμε όμως με το δεύτερο πρόβλημα ξεκινώντας από ένα παράδειγμα της §21.6.1. Εκεί, γράφοντας την επιφόρτωση του "`operator=`"της *BString* με βάση τον δημιουργό αντιγραφής, είχαμε γράψει:

```
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
```

Δηλαδή: πιάναμε την όποια εξαίρεση *BStringXptn*, διορθώναμε το *funcName* και την ξαναρίχναμε. Έτσι, φαίνεται ότι την εξαίρεση τη ρίχνει ο "`operator=`".

Το σωστό όμως θα ήταν να πούμε ότι *την εξαίρεση τη ρίχνει ο δημιουργός όταν καλείται από τον* "`operator=`". Θα μπορούσαμε να γράψουμε:

```
      catch( BStringXptn& x )
      {  strcat( x.funcName, "|operator=" );
         throw;  }
```

Έτσι, η *x.funcName* έχει τιμή `"BString|operator="`. Και αν εξοπλίσουμε τη *BStringXptn* με μια:

```
   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                              else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
```

από τις

```
// . . .
         case BStringXptn::noMemory:
            cout << "out of memory "; x.displayFuncName( cout );
            break;
// . . .
```

θα μπορούσαμε να πάρουμε:

```
out of memory thrown in BString
 called by operator=
```

Καλό θα είναι να εξοπλίσουμε τη *BStringXptn* και με μια

```
   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
```

```
      strcat( funcName, fn );  } // appendFuncName
```

και έτσι θα γράφουμε:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  x.appendFuncName( "operator=" );
         throw;  }
   }
   return *this;
} // BString::operator=
```

Η *BStringXptn* τώρα έχει γίνει:

```
struct BStringXptn :  public exception
{
   static const int fnLength = 100;
   enum { noMemory, outOfRange };
   char funcName[ fnLength ];
   int  errorCode;
   int  errorValue;
   BStringXptn( const char* fn, int ec, int ev = 0 )
   {  strncpy( funcName, fn, fnLength-1 );
                                    funcName[fnLength-1] = '\0';
      errorCode = ec;  errorValue = ev;  }
   virtual ~BStringXptn() { };

   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
      strcat( funcName, fn );  } // appendFuncName

   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                              else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
}; // BStringXptn
```

Εκτός από τις δύο μεθόδους πρόσεξε και κάτι άλλο: Αντικαταστήσαμε τη «μαγική σταθερά» "`100`" με "`fnLength`" που ορίζεται ως:

```
   static const int fnLength = 100;
```

Αν σκεφτείς ότι μπορεί να χρειαστεί να αποθηκεύεις στη *funcName* όλα τα βήματα-κλήσεις συναρτήσεων και στα ονόματα συναρτήσεων-μελών να βάζεις και το όνομα της κλάσης –π.χ. όχι "`appendFuncName("load")`" αλλά πιο συγκεριμένα "`appendFuncName( "Course::load")`"– το "`100`" μπορεί να μην είναι αρκετό.

Ένα δεύτερο παράδειγμα χρήσης αυτών των εργαλείων μπορούμε να δούμε στον δημιουργό της *DateTime* όπως τον ξαναγράψαμε στην §24.5.

Αλλάζουμε την κλάση εξαιρέσεων:

```
struct DateXptn :  public exception
{
   static const int fnLength = 100;
   enum { yearErr, monthErr, dayErr, outOfLimits,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[ fnLength ];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
```

```
   Date errDateVal;
   DateXptn( const char* fn, int ec,
             int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : errorCode(ec), errVal1(ev1), errVal2(ev2), errVal3(ev3)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   DateXptn( const char* fn, int ec, int ev1, const Date& ed )
      : errorCode(ec), errVal1(ev1), errDateVal(ed)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   virtual ~DateXptn() throw() {  };

   void appendFuncName( const char* fn )
   // ΟΠΩΣ ΣΤΗ BStringXptn

   void displayFuncName( ostream& tout )
   // ΟΠΩΣ ΣΤΗ BStringXptn
}; // DateXptn
```

και τον δημιουργό της *DateTime*:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}
catch( DateTimeXptn& x )
{  throw;  }
catch( DateXptn& x )
{
   x.appendFuncName( "DateTime" );
   throw;
} // DateTime::DateTime
```

Τώρα μπορεί να έχουμε ένα μήνυμα σαν

```
day error ( 44 ) thrown in Date
 called by DateTime
```

που είναι ακριβέστερο από το να πούμε απλώς ότι «*βρήκαμε λάθος στον δημιουργό της DateTime.*»

Ας δούμε τώρα πώς θα αξιοποιήσουμε αυτά τα εργαλεία. Μήπως θα πρέπει να ξαναγράψουμε όλες τις συναρτήσεις και όλες τις μεθόδους ώστε να πιάνουμε ολόκληρη την ακολουθία κλήσεων; Ούτε να το σκέφτεσαι!

- ***Η διαδικασία «πιάσε – διόρθωσε – ξαναρίξε» είναι χρονοβόρα.***

Δεν μπορείς επομένως να την έχεις παντού σε ένα πρόγραμμα που είναι σε εκμετάλλευση. Και αν μεν η ρίψη της εξαίρεσης θα έχει αποτέλεσμα την διακοπή της εκτέλεσης του προγράμματος οι καθυστερήσεις δεν μας πειράζουν. Αν όμως υπάρχει δυνατότητα να κάνεις διόρθωση και να συνεχίσεις την εκτέλεση το πράγμα αλλάζει.

Όταν κάνεις την ανάπτυξη του προγράμματος και ψάχνεις για να εντοπίσεις και να διορθώσεις μερικά –ενδεχομένως δύσκολα– λάθη αυτά τα εργαλεία μπορεί να σε βοηθήσουν.

Μπορεί να τα χρησιμοποιήσεις και στο πρόγραμμα που είναι σε εκμετάλλευση αλλά σε σημεία που θα επιλέξεις με προσοχή.

Σε κάθε περίπτωση πάρε υπόψη σου και το εξής: *Υπάρχουν ακολουθίες κλήσεων που ταυτοποιούνται με μοναδικό τρόπο από την αρχή και το τέλος.*

24.9 Ασφάλεια ως προς τις Εξαιρέσεις

Στις §21.6 και §21.6.1 είδαμε δύο μορφές της `BString::operator=()` που είπαμε ότι έχουν **ασφάλεια προς τις εξαιρέσεις** (exception safety) επιπέδου **ισχυρής εγγύησης** (strong guarantee). Όπως είπαμε, με τον όρο αυτόν εννοούμε ότι αυτή η μέθοδος (επιφόρτωσης) έχει τα εξής χαρακτηριστικά: σε περίπτωση που θα εγερθεί εξαίρεση όταν εκτελείται η "`a = b`"

- Δεν θα έχουμε διαρροή μνήμης.
- Δεν θα αλλάξει η τιμή της *a*.

Η ασφάλεια ισχυρής εγγύησης της δεύτερης μορφής στηρίζεται στη σιγουριά μας ότι η *BString*`::`*swap()* δεν θα ρίξει εξαίρεση. Λέμε ότι η *BString*`::`*swap()* έχει **ασφάλεια προς τις εξαιρέσεις** σε επίπεδο **εγγύησης μη-ρίψης** (no-throw guarantee).

Στην §20.7.2.3 είδαμε την

```
void Route::erase1RouteStop( int ndx )
{
   for ( int k(ndx); k <= rNoOfStops; ++k )
      rAllStops[k] = rAllStops[k+1];
   --rNoOfStops;
} // Route::erase1RouteStop
```

που «σβήνει» το στοιχείο που βρίσκεται στη θέση *ndx* σε έναν ταξινομημένο πίνακα. Λέγαμε όμως ότι κάποια από τις "`rAllStops[k] = rAllStops[k+1]`" της *erase1RouteStop()* μπορεί να ρίξει *bad_alloc* διότι τα `rAllStops[k]`, που είναι κλάσης *RouteStop*, έχουν μέλος "`string sName`".

Ας πούμε λοιπόν ότι ρίχνεται η εξαίρεση όταν ο δείκτης *k* έχει τιμή k_0. Ο τελεστής εκχώρησης της *string* έχει ασφάλεια ισχυρής εγγύησης που σημαίνει:

- δεν έχουμε διαρροή μνήμης και
- το `rAllStops[`k_0`]` κρατάει την αρχική του τιμή. Επομένως:
  - Αν k_0 `==` *ndx* σημαίνει ότι ο πίνακας έχει μείνει αθικτος και η αναλλοίωτη της κλάσης ισχύει.
  - Αν k_0 > *ndx*, στο προηγούμενο βήμα της **for** το `rAllStops[`k_0-1`]` έχει πάρει την τιμή του `rAllStops[`k_0`]`. Έτσι, ενώ έχουμε σβήσει το `rAllStops[`*ndx*`]`, το πλήθος των στοιχείων μεταξύ των δύο φρουρών παραμένει αυτό που ήταν αφού τα στοιχεία στις θέσεις k_0-1 και k_0 έχουν την ίδια τιμή. Αυτή η κατάσταση παραβιάζει την αναλλοίωτη της κλάσης και συγκεκριμένα το κομμάτι:

$$(\forall\ k\colon 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{+}1].sDist > \text{rAllStops}[k].sDist)$$

Πάντως: και στις δύο περιπτώσεις το βέλος `rAllStops` δείχνει τον δυναμικό πίνακα, το *rReserved* μας λέει πόση δυναμική μνήμη έχουμε πάρει και το *rNoOfStops* πόση από αυτήν είναι σε χρήση. Δηλαδή το αντικείμενο είναι διαχειρίσιμο είτε με τον καταστροφέα είτε με την *Route*`::`*clearRouteStops()*.

Λέμε ότι η *Route*`::`*erase1RouteStop()* έχει **ασφάλεια** επιπέδου **βασικής εγγύησης** (basic guarantee).[22]

Να τα τρία επίπεδα εγγύησης ασφάλειας μιας μεθόδου (ή μιας συνάρτησης που διαχειρίζεται ένα αντικείμενο) ως προς τις εξαιρέσεις (Abrahams 2001):

- Η *βασική* εγγύηση: *Δεν υπάρχει διαρροή πόρων και η αναλλοίωτη του αντικειμένου διατηρείται*. Το ότι ισχύει η αναλλοίωτη σημαίνει ότι μπορεί να κληθεί ο καταστροφέας (ας πούμε με το ξετύλιγμα της στοίβας) που έχει την αναλλοίωτη ως προϋπόθεση. Κατά τα άλλα, αυτό δεν λέει και πολλά πράγματα διότι καταστάσεις που να ισχύει η αναλλοίωτη μπορεί να υπάρχουν πολλές. Αν θέλεις να συνεχίσεις να δουλεύεις με το αντικείμε-

[22] Μερικοί ονομάζουν αυτό το επίπεδο **ελάχιστης** (minimal) εγγύησης διότι δεν ισχύει ολόκληρη η αναλλοίωτη.

νο θα πρέπει να το φέρεις σε συγκεκριμένη κατάσταση π.χ. να κάνεις κάποιο είδος επανεκκίνησης.

- Η *ισχυρή* εγγύηση: Η εκτέλεση της μεθόδου
  - είτε *ολοκληρώνεται επιτυχώς*
  - είτε *διακόπτεται με έγερση εξαίρεσης αλλά το πρόγραμμα* (και το αντικείμενο) *βρίσκεται στην κατάσταση που ήταν πριν αρχίσει η εκτέλεση της μεθόδου. Δεν υπάρχει διαρροή πόρων.*
- Η εγγύηση *μη-ρίψης*: *Η εκτέλεση της μεθόδου ολοκληρώνεται επιτυχώς χωρίς να ριχτεί (ή να περάσει μέσα από αυτήν) εξαίρεση.*

Προφανώς για κάθε μέθοδο θα θέλαμε να έχουμε το μέγιστο δυνατό επίπεδο ασφάλειας. Ποιο ακριβώς είναι το επίπεδο εξαρτάται από τη συγκεκριμένη μέθοδο. Και πώς πετυχαίνουμε το επίπεδο που θέλουμε; Ας ξαναγυρίσουμε, για παράδειγμα, στη *Route*::*erase1-RouteStop* και ας πούμε ότι προσπαθούμε να την κάνουμε να έχει ισχυρή εγγύηση ασφάλειας. Μπορούμε να ξαναδώσουμε την αρχική τιμή στα στοιχεία που αλλάξαμε; Ούτε λόγος! Θα πρέπει να κάνουμε αντιγραφές που –λογικώς– θα ρίξουν εξαίρεση. Ο μόνος τρόπος είναι ο εξής: Να μην πειράξουμε τον αρχικό πίνακα αλλά να τον αντιγράψουμε σε έναν άλλον

```
RouteStop* tmp( new RouteStop[rReserved] );
```

παραλείποντας το `rAllStops[`*ndx*`]`:

- Αν η αντιγραφή ολοκληρωθεί επιτυχώς ανταλλάσσουμε τις τιμές των βελών *rAllStops* και *tmp*.
- Αλλιώς, αν ριχτεί εξαίρεση, ο πίνακας παραμένει όπως ήταν.

Δηλαδή: για να διαγράψουμε μια τιμή αντιγράφουμε ολόκληρον τον πίνακα. Είναι δεκτό αυτό; Αλλά και τώρα μήπως κάνουμε κάτι καλύτερο; Κατά μέσον όρο αντιγράφουμε τον μισό πίνακα! Γενικώς η αντιγραφή πίνακα –για διαγραφή ή εισαγωγή μιας τιμής– φαίνεται απαράδεκτη αλλά δεν έχουμε και άλλη επιλογή (πάντως εδώ οι πίνακες δεν είναι και τόσο μεγάλοι).

Δηλαδή το καλό επίπεδο ασφάλειας για τις εξαιρέσεις πληρώνεται με (μεγάλο) υπολογιστικό χρόνο; Όχι απαραιτήτως. Η λύση στο πρόβλημά μας βρίσκεται στην αλλαγή δομής δεδομένων για τις στάσεις που αποτελούν ένα σύνολο: όπως θα δεις αργότερα, χρησιμοποιώντας τα εργαλεία που μας δίνει η STL, θα τις βάλουμε σε ένα σύνολο με στοιχεία τύπου *RouteStop* ή, όπως θα το γράφουμε, "`set<RouteStop>`"· σε αυτή τη δομή οι εισαγωγές και οι διαγραφές στοιχείων γίνονται με ισχυρή εγγύηση ασφάλειας χωρίς να είναι χρονοβόρες.

Επομένως

♦ ***Για να έχουμε το επιθυμητό επίπεδο εγγύησης ασφάλειας πρέπει να σχεδιάσουμε καταλλήλως το πρόγραμμά μας.***

Για να διασφαλίσουμε τη μη-διαρροή πόρων χρησιμοποιούμε τις **try**/**catch**. Η τεχνική RAII είναι σαφώς καλύτερη και ασφαλέστερη. Με τα εργαλεία της STL, που θα μάθουμε αργότερα, η εφαρμογή της θα γίνει απλούστερη.

Ένα εργαλείο, πολύ συχνά χρήσιμο, είναι η (ασφαλής) *swap()*. Μερικές φορές θα πρέπει να συνδυάσεις τη χρήση της με την τεχνική PIMPL (§22.10).

24.10 Σύνοψη

Η C μας δίνει τη δυνατότητα να βρούμε τί δεν πήγε καλά όταν καλέσαμε μια από τις συναρτήσεις της βιβλιοθήκης της

- από την τιμή που επιστρέφει η συνάρτηση και

- την τιμή της καθολικής μεταβλητής *errno*.

Μας δίνει τη δυνατότητα να διακόψουμε την εκτέλεση του προγράμματος (*abort()*, *exit()*, *_Exit()*) και να καθορίσουμε τι θα συμβεί στην περίπτωση αυτή (*atexit()*).

Με την *assert()* μπορούμε να ελέγχουμε τις συνθήκες επαλήθευσης στα διάφορα σημεία εκτέλεσης του προγράμματος. Αλλά, όπως ξέρουμε, αν η συνθήκη δεν ισχύει η «ποινή είναι βαριά».

Για τις εξαιρέσεις της C++ μάθαμε ότι

- Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται η *terminate()* (που καλεί την *abort()*) για να διακόψει την εκτέλεση του προγράμματος.
- Αν παραβιασθεί μια προδιαγραφή εξαιρέσεων –δηλαδή αν από μια συνάρτηση ριχτεί εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση *unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Οι *set_terminate()* και *set_unexpected()* σου επιτρέπουν να αντικαταστήσεις τις *terminate()* και *unexpected()* αντιστοίχως.

Αν έλθουμε στο «πρακτέο» θα πρέπει να πούμε κατ' αρχάς ότι καλό είναι να αφήσεις κατά μέρος τις προδιαγραφές εξαιρέσεων εκτός από την "`throw()`".

Αν γράφεις δικές σου κλάσεις εξαιρέσεων (ή χρησιμοποιείς τις δικές μας) καλό είναι να τις βάζεις να κληρονομούν (αμέσως ή εμμέσως) την *exception*.

Όταν γράφεις το πρόγραμμά σου στόχος σου θα πρέπει να είναι η καλύτερη δυνατή εγγύηση ασφάλειας ως προς τις εξαιρέσεις. Η τεχνική RAII και η ασφαλής *swap()* είναι εργαλεία που μπορείς να χρησιμοποιήσεις.

Πάντως

- η ασφάλεια ως προς τις εξαιρέσεις και
- το πού θα πιάνεις και πώς θα χειρίζεσαι τις εξαιρέσεις

έχουν σχέση (και) με τη σχεδίαση του προγράμματός σου.

κεφάλαιο

25

Περιγράμματα Κλάσεων

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα μάθουμε πώς μπορούμε να υλοποιήσουμε δομές δεδομένων με προγραμματιστικά αντικείμενα της C++: τα περιγράμματα κλάσεων.

Προσδοκώμενα αποτελέσματα:
Όπως με ένα περίγραμμα συνάρτησης μπορείς να υλοποιήσεις προγραμματιστικώς έναν αλγόριθμο, με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων ώστε να μπορείς να έχεις στιγμιότυπα για τους τύπους που σε ενδιαφέρουν.

Έννοιες κλειδιά:

- *περίγραμμα κλάσης*
- *παράμετρος περιγράμματος*
- *εξειδίκευση περιγράμματος*
- *μερική εξειδίκευση περιγράμματος*
- *περιγράμματα και κληρονομιές*
- *περιέχουσα κλάση*
- *περιεχόμενη κλάση*
- *έξυπνα βέλη*
- *κλάσεις χαρακτηριστικών*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Όπως είναι χρήσιμο να έχουμε *περιγράμματα συναρτήσεων*, όπου μεταφράζουμε σε C++ γενικούς αλγόριθμους με τους τύπους των στοιχείων να είναι παράμετροι, είναι χρήσιμο να έχουμε και **περιγράμματα κλάσεων**[1] (class templates) στα οποία μπορούμε να μεταφράσουμε σε C++ δομές δεδομένων και τους αλγόριθμους για τον χειρισμό τους.

Παρόμοιες δυνατότητες προσφέρουν και άλλες γλώσσες προγραμματισμού και –μεταξύ αυτών– τα άλλα «παιδιά της C»:

- Η Java έχει τους **γενικούς τύπους** (generic types).
- Η C# έχει τις **γενικές κλάσεις** (generic classes).

Ο τύπος (*std*::)*string*, που χρησιμοποιούμε συνεχώς στα προγράμματά μας, είναι ένα στιγμιότυπο του περιγράμματος *basic_string*. Ακολουθώντας και εμείς αυτήν την ιδέα(!) θα ξεκινήσουμε την παρουσίαση των περιγραμμάτων κλάσεων ως εξής:

- Από τη *BString* θα πάρουμε το περίγραμμα κλάσης *BStringT*.
- Θα ξαναπάρουμε την αρχική κλάση ως στιγμιότυπο του περιγράμματος.

25.1 Από την Κλάση *BString* στο Περίγραμμα

Στα προηγούμενα κεφάλαια αναπτύξαμε το παράδειγμα της κλάσης *BString*. Επειδή, η κλάση αυτή είναι χρήσιμη γενικότερα, θα θέλαμε να την μετατρέψουμε σε *περίγραμμα* ώστε να μπορούμε να τη χρησιμοποιήσουμε, όχι μόνον με τον `char`, αλλά και με άλλους τύπους, όπως ο `unsigned char` ή ο `wchar_t`. Η C++ μας δίνει τη δυνατότητα να έχουμε, εκτός από περιγράμματα συναρτήσεων, και περιγράμματα κλάσεων.

Στην κλάση, όπως την έχουμε μέχρι τώρα, διαχωρίζουμε τον ερήμην δημιουργό από τον δημιουργό με αρχική τιμή:

```
 0: class BString
 1: { // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
 2: friend bool operator==( const BString& lhs,
 3:                         const BString& rhs );
 4: public:
 5:    BString();
 6:    BString( const char* rhs, int n=0 );
 7:    BString( const BString& rhs );
 8:    ~BString() {  delete[] bsData;  };
 9:    BString& operator=( const BString& rhs );
10:    BString& assign( const BString& rhs )
11:                                    {  return (*this = rhs);  }
12:    const char* c_str() const
13:                     {  bsData[bsLen] = '\0';  return bsData;  }
14:    size_type length() const {  return bsLen;  }
15:    bool empty() const {  return ( bsLen == 0 );  }
16:    char& at( int k ) const;
17:    char& operator[]( int pos ) const {  return bsData[pos];  }
18:    BString& operator+=( const BString& rhs );
19:    BString& append( const BString& rhs )
20:                                   {  return (*this += rhs);  }
21:    void swap( BString& rhs );
22:    int compare( const BString& rhs ) const;
23: private:
24:    static const unsigned int bsIncr = 16;
25:    char*     bsData;
26:    size_type bsLen;
27:    size_type bsReserved;
```

[1] Θα δεις και τον όρο **παραμετρικές** (parametric) κλάσεις.

```
 28:
 29:    static size_type cStrLen( const char* cs );
 30:    static int stringCmpr( const char* lhs, const char* rhs,
 31:                           int n );
 32: }; // BString
```

Ο ορισμός του ερήμην δημιουργού είναι:

```
BString::BString()
{
   try {  bsData = new char[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BString::BString
```

Βάλαμε μέσα στη δήλωση της κλάσης τους ορισμούς των τριών μεθόδων `inline` και του καταστροφέα.

Πέρα από τη φίλη καθολική `operator==()` να θυμηθούμε ότι έχουμε και την καθολική:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

Για να πάρουμε περίγραμμα κλάσης, ας το πούμε *BStringT*, από την ειδική περίπτωση ορμαθών με στοιχεία τύπου `char`, κάνουμε τα εξής:

- Αλλάζουμε το `'\0'` σε "`char(0)`" (στη γρ. 13)
- Αντικαθιστούμε τον τύπο "`char`"[2], όπου τον βρούμε (γρ. 6, 12, 16, 17, 25, 29, 30), με μια παράμετρο, ας την πούμε "`K`".
- Αντικαθιστούμε το "`BString`", όπου τον βρούμε, με "`BStringT`".

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT& lhs,
                        const BStringT& rhs );
public:
   BStringT();
   BStringT( const K* rhs, int n=0 );
   BStringT( const BStringT& rhs );
   ~BStringT() {  delete[] bsData;  };
   BStringT& operator=( const BStringT& rhs );
   BStringT& assign( const BStringT& rhs )
                                        {  return (*this = rhs);  }
   const K* c_str() const
                          {  bsData[bsLen] = K(0);  return bsData;  }
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   K& at( int k ) const;
   K& operator[]( int pos ) const {  return bsData[pos];  }
   BStringT& operator+=( const BStringT& rhs );
   BStringT& append( const BStringT& rhs )
                                       {  return (*this += rhs);  }
   void swap( BStringT& rhs );
   int compare( const BStringT& rhs ) const;
private:
   static const unsigned int bsIncr = 16;
   K*        bsData;
   size_type bsLen;
   size_type bsReserved;
```

[2] Με το "replace" του κειμενογράφου!

```
   static size_type cStrLen( const K* cs );
   static int stringCmpr( const K* lhs, const K* rhs, int n );
}; // BStringT
```

Βλέπεις φυσικά και το –γνωστό μας από τα περιγράμματα συναρτήσεων– «καπέλο» "`template < typename K >`".

Τώρα, αντί για τη δήλωση:

```
   BString s1( "abc" ), s2, s3;
   BString bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

μπορούμε να γράψουμε:

```
   BStringT<char>  s1( "abc" ), s2, s3;
   BStringT<char>  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Λέμε δηλαδή ότι ο τύπος (κλάση) είναι ένα *στιγμιότυπο* του περιγράμματος κλάσης *BStringT* για παράμετρο-τύπο `char`.

Στη βιβλιοθήκη της C++ ορίζεται κατ' αρχάς το *basic_string* που είναι περίγραμμα κλάσης. Η *string* ορίζεται ως στιγμιότυπο του περιγράμματος:

```
typedef basic_string<char> string;
```

Παρομοίως «βαφτίζεται» και ένα άλλο στιγμιότυπο:

```
typedef basic_string<wchar_t> wstring;
```

Μιμούμενοι τα παραπάνω, μετά τον ορισμό του περιγράμματος, θα μπορούσαμε να ορίσουμε:

```
typedef BStringT< char >                  String;
```

και να γράψουμε τις παραπάνω δηλώσεις:

```
   String  s1( "abc" ), s2, s3;
   String  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Η *String* δεν είναι άλλη από τη *BString*.

Μπορούμε ακόμη να ορίσουμε:

```
typedef BStringT< wchar_t >               WString;
typedef BStringT< unsigned char >         UString;
```

και να δηλώσουμε:

```
   WString s1( L"abcd" ), s2( L"abcd" );
```

Να δούμε τώρα πώς γράφονται οι ορισμοί των μεθόδων. Για τον καταστροφέα, τη *c_str()*, τη *length()*, την *empty()*, την *assign()* και την *append()* δεν υπάρχει πρόβλημα· ορίζονται με τις δηλώσεις τους. Ας δούμε την `operator+=()` που επιφορτώνει τον "`+=`" και την ορίζουμε έξω από τη δήλωση της κλάσης. Όπως καταλαβαίνεις, αφού η *BStringT* έγινε περίγραμμα κλάσης και η `operator+=()` είναι πια *περίγραμμα συνάρτησης*. Φυσικά θα έχει τις ίδιες παραμέτρους με την κλάση. Άρα ο ορισμός της θα αρχίζει με τα: "`template < typename K >`".

Αν γράφαμε τη μέθοδό μας για την κλάση `BStringT<char>` η επικεφαλίδα της θα ήταν:

```
BStringT<char>& BStringT<char>::operator+=( const BStringT<char>& rhs )
```

ενώ, αν τη γράφαμε για τη `BStringT<wchar_t>` θα ήταν:

```
BStringT<wchar_t>&
                BStringT<wchar_t>::operator+=( const BStringT<wchar_t>& rhs )
```

Μπορούμε λοιπόν να μαντέψουμε ότι όταν έχουμε στοιχεία κλάσης *K* θα πρέπει να έχουμε:

```
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
```

Πράγματι, το περίγραμμα της μεθόδου (συνάρτησης) θα είναι:

```
template < typename K >
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
```

```
      K* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new K[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringTXptn( "operator+=",
                             BStringTXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BStringT<K>::operator+=
```

Πρόσεξε ότι οι αλλαγές είναι

- στην επικεφαλίδα,
- στη δήλωση "`char* tmp`" που έγινε "`K* tmp`" και
- στη "`new char[tmpRes]`" που έγινε "`new K[tmpRes]`".

Να δούμε πώς γίνεται ο ερήμην δημιουργός, που είδαμε και πιο πάνω:

```
template < typename K >
BStringT<K>::BStringT()
{
   try {  bsData = new K[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringTXptn( "BStringT",
                          BStringTXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BStringT<K>::BStringT<K>
```

Πρόσεξε ότι το όνομα του δημιουργού είναι `BStringT` και όχι `BStringT<K>`.

25.2 Φίλες Συναρτήσεις Περιγραμμάτων

Ας δούμε τώρα τι γίνεται με τις φίλες συναρτήσεις: Κατ' αρχήν, δηλώνονται όπως ακριβώς ξέρουμε:

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Φυσικά, και αυτές γίνονται τώρα περιγράμματα συναρτήσεων και έτσι δίνονται οι ορισμοί τους, π.χ.:

```
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int int fv( BStringT<K>::stringCmpr(lhs.bsData, rhs.bsData,
                                       min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==( const BStringT<K>&
```

Ωραία όλα αυτά, αλλά –όπως είπαμε– «κατ' αρχήν». Το πιο πιθανό είναι ότι ο μεταγλωττιστής σου θα τα απορρίψει! Γιατί; Διότι δεν έχει την «υπομονή» να περιμένει να βρεί τον ορισμό της συνάρτησης `operator==()` και μόλις βρει τη δήλωση `friend` μας λέει ότι δεν ξέρει τέτοια συνάρτηση.

Η πιο απλή λύση στο πρόβλημά μας είναι η εξής: βάζουμε τον ορισμό της συνάρτησης μαζί με τη δήλωση `friend`:

```
template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int fv( stringCmpr(lhs.bsData, rhs.bsData,
                      min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Αυτό θα περάσει από τον μεταγλωττιστή. Πάντως ορισμένοι μεταγλωττιστές (π.χ.: gcc) δεν θα καταλάβουν ότι αυτό είναι περίγραμμα συνάρτησης και θα πρέπει να δηλώσεις:

```
template < typename K1 >
friend bool operator==( const BStringT<K1>& lhs, const BStringT<K1>& rhs )
// . . .
```

Σημείωση:▶
Αν θέλεις να βάλεις τον ορισμό της φίλης συνάρτησης έξω από τον ορισμό της κλάσης θα πρέπει θα *πρέπει να βάλεις πριν από την κλάση προειδοποιητικές δηλώσεις (§22.4) της κλάσης και της συνάρτησης*:

```
template < typename K > class BStringT;
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );

template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
// . . .
```

Στη συνέχεια, μπορείς να βάλεις τον ορισμό της συνάρτησης όπου θέλεις.◀

25.3 Καθολικές Συναρτήσεις για Περιγράμματα

Περιγράμματα συναρτήσεων θα γίνουν και οι καθολικές συναρτήσεις που έχουν σχέση με κάποιο περίγραμμα κλάσης. Δες πώς θα γίνει η (καθολική συνάρτηση) `operator<<()` που επιφορτώνει τον "`<<`":

```
template < typename K >
ostream& operator<<( ostream& tout, const BStringT<K>& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BStringT<K>
```

Παρατήρηση:▶
Θα πρέπει να παρατηρήσουμε εδώ ότι αν πάρουμε στιγμιότυπο του περιγράμματος για `wchar_t`, η `operator<<()` δεν δουλεύει όπως θέλουμε! Θα χρειαστείς εξειδίκευση της συνάρτησης (όχι της κλάσης).◀

25.3.1 Ένα Απλό Παράδειγμα: *pair*

Αν βάλεις στο πρόγραμμά σου "`#include <utility>`" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*`::`*pair* που βάζει σε μια `struct` δύο μεταβλητές και χρησιμοποιείται στην STL. Εδώ θα γράψουμε ένα περίγραμμα κλάσης

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;
// . . .
}; // PairT
```

που θα έχει ακριβώς τις ιδιότητες του *pair*: Μια **struct** με δύο μέλη και λεξικογραφική διάταξη. Τι θα πει αυτό; Ορίζεται ο "<" ως εξής: Αν

```
PairT<T1, T2> x, y;
```

τότε η x < y ισχύει αν και μόνον αν:

$$(x.first < y.first) \;||\; (x.first == y.first \;\&\&\; x.second < y.second)$$

ή, ισοδυνάμως:

$$x.first < y.first \;||\; (!(y.first < x.first) \;\&\&\; x.second < y.second$$

Αυτή η μορφή[3] *δεν έχει σύγκριση για ισότητα* και επομένως είναι προτιμότερη αν ο *T1* είναι τύπος κινητής υποδιαστολής.

Ο ερήμην δημιουργός θα είναι:

```
   PairT()
      :  first( T1() ), second( T2() ) { };
```

πράγμα που *προϋποθέτει την ύπαρξη ερήμην δημιουργών για τους T1 και T2*.

Ένας δημιουργός με αρχικές τιμές θα είναι:

```
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
```

Για να μπορούμε να γράψουμε τα "**first(x)**" και "**second(y)**" θα πρέπει *οι T1 και T2 να έχουν δημιουργούς με αρχική τιμή*.

Για τύπους *U* και *V* που έχουν δημιουργούς μετατροπής του *U* στον *T1* και του *V* στον *T2* μπορούμε να γράψουμε

```
   PairT( const PairT<U,V>& a )
      : first( a.first ), second( a.second ) { };
```

Και αυτό πώς μπαίνει μέσα στη δήλωση του περιγράμματος; Ως περίγραμμα μέσα σε περίγραμμα!

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT
```

Όπως καταλαβαίνεις, κάθε στιγμιότυπο του *PairT* θα είναι μια κλάση που θα περιέχει ένα περίγραμμα δημιουργού.

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε τον "<" ως:

```
template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
                          (!(y.first < x.first) && x.second < y.second) );  }
```

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε και τον "==":

[3] που την προτιμάει το πρότυπο…

```
template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }
```

όπου φυσικά δεν μπορούμε να αποφύγουμε τη σύγκριση για ισότητα όποιοι και αν είναι οι *T1*, *T2*.

Τέλος, το περίγραμμα καθολικής συνάρτησης:

```
template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }
```

ζευγαρώνει δύο τιμές *x*, *y* τύπων *T1*, *T2* αντιστοίχως.

Ας καταγράψουμε τις απαιτήσεις από τους *T1*, *T2* για να μπορούμε να ζευγαρώνουμε τιμές τους με το *PairT*: Θα πρέπει

- Να έχουν ερήμην δημιουργούς.
- Να έχουν δημιουργούς αρχικής τιμής.
- Να έχουν τον τελεστή "`<`".
- Να έχουν τον τελεστή "`==`".

Και κάτι ακόμη: χωρίς να μπορούμε να επιφορτώσουμε τον "`=`" –αφού δεν ξέρουμε τους *T1*, *T2*– θα πρέπει η εκχώρηση να γίνεται σωστά. Αυτό σημαίνει ότι: *ο τελεστής εκχώρησης "`=`" δουλεύει σωστά για τους T1, T2*.

Όσο για τη χρήση ενός τέτοιου περιγράμματος να πούμε τα εξής: Όπως θα καταλάβεις αργότερα, χρησιμοποιώντας τα εργαλεία της STL στο παράδειγμα με τους φοιτητές και τα μαθήματα, θα δημιουργηθούν ζεύγη τύπων όπως:

```
PairT< CourseKey, Course >
PairT< unsigned int, Student >
```

Βέβαια, μπορεί να σου κατέβουν και άλλες ιδέες. Θα μπορούσες να γράψεις τη *minmax* που είδαμε στην §13.9.3 (Παρ. 1) έτσι:

```
PairT< int, int > minmax( int x, int y )
{
   PairT< int, int > fv;

   if ( x < y )
      fv = PairT< int, int >( x, y );
   else // x >= y
      fv = PairT< int, int >( y, x );
   return fv;
} // minmax
```

ή την *pqer* (που είδαμε στις §14.9 και §15.10) ως εξής:

```
PairT< double, double > pqer( double x, double y, double z )
{
   if ( x == y || x == -y || z <= 0 )
      throw ApplicXptn( "pqer", ApplicXptn::paramErr, x, y, z );
   PairT< double, double > fv;
   fv.first = x*y*pow(z, x-y)/(x*x-y*y);
   fv.second = (x*y-1/x)/z;
   return fv;
} // pqer
```

Έτσι, έχουμε συναρτήσεις με τύπο που λέγαμε ότι είναι προτιμότερες από τις "`void`". Βεβαίως, αλλά στην περίπτωση αυτήν ο τύπος είναι κάπως «φτιαχτός»…

25.4 Κατανομή σε Αρχεία

Πριν προχωρήσουμε, ας κάνουμε μια επισήμανση, συμπλήρωμα στην §19.3: Το περίγραμμα μιας κλάσης γράφεται, κατ' αρχήν, σε ένα αρχείο, το "`.h`". Έτσι, για παράδειγμα, το *PairT* θα φυλαχτεί σε ένα αρχείο, το `PairT.h`, που θα έχει:

```
#ifndef _PAIRT_H
#define _PAIRT_H

using namespace std;

template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT

template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }

template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
           (!(y.first < x.first) && x.second < y.second) );  }

template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }

#endif // _PAIRT_H
```

25.5 Παράμετροι και Εξειδικεύσεις

Στην §14.7.1 λέγαμε ότι ένα περίγραμμα (συνάρτησης) «*μπορεί να έχει ως παραμέτρους*

- *έναν ή περισσότερους τύπους ή/και*
- *ακέραιους,*
- *βέλη ή αναφορές.*»

Ας τα δούμε με περισσότερες λεπτομέρειες. Οι παράμετροι ενός περιγράμματος μπορεί να είναι:

- Το λεξικό σύμβολο "`class`" ή "`typename`" ακολουθούμενο από όνομα της παραμέτρου. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμη (τύπο) της παραμέτρου.

  Τα "`class`" και "`typename`" δεν έχουν οποιαδήποτε νοηματική διαφορά. Μπορείς να χρησιμοποιείς όποιο θέλεις.[4]
- Το λεξικό σύμβολο "`template`" ακολουθούμενο από λίστα παραμέτρων περιγράμματος μέσα σε "`<`" και "`>`" ακολουθούμενη από το σύμβολο "`class`" και ένα όνομα. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμή (περίγραμμα) της παραμέτρου. Το παρακάτω παράδειγμα είναι από το πρότυπο C++03:

```
template< typename T > class myarray { /* ... */ };

template< typename K, typename V,
          template< typename T> class C = myarray >
class Map
```

[4] Πάντως το "`typename`" είναι πιο σωστό –παρ' όλο που ορισμένοι μεταγλωττιστές έχουν πρόβλημα με αυτό– αφού μπορείς εκεί να βάλεις "`int`" ή "`double`" (ή άλλα παρόμοια) που δεν είναι κλάσεις.

```
{
   C< K > key;
   C< V > value;
// ...
};
```

Όπως βλέπεις η ερήμην καθορισμένη τιμή "`myarray`" είναι περίγραμμα κλάσης της μορφής "`template< typename T > class C`".

- Παράμετρος ακέραιου ή απαριθμητού τύπου.
- Παράμετρος-βέλος προς αντικείμενο ή συνάρτηση.
- Παράμετρος-αναφοράς αντικειμένου ή συνάρτησης.
- Παράμετρος-βέλος προς μέλος.

Αν κάποιο στιγμιότυπο –ας πούμε το *BStringT*<*AClass*>– που βγαίνει αυτόματως δεν καλύπτει τις απαιτήσεις σου μπορείς, όπως και στα περιγράμματα συναρτήσεων, να κάνεις μια εξειδίκευση:

```
template<> class BStringT<AClass> { /* . . . */ };
```

Αλλά πρόσεχε:

♦ ***Για την εξειδίκευση θα πρέπει να ορίσεις από την αρχή τα πάντα.***

Η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!

Παρατήρηση:▶

Αυτό που θα λύνει το πρόβλημά σου σε πολλές περιπτώσεις είναι η εξειδίκευση μιας ή περισσότερων μεθόδων. Δηλαδή: αφού για κάθε μέθοδο γράφουμε ένα περίγραμμα συνάρτησης μπορούμε να κάνουμε εξειδίκευση του περιγράμματος της μεθόδου για τις περιπτώσεις που μας ενδιαφέρουν.

Έστω για παράδειγμα το περίγραμμα κλάσης:

```
template < typename K > class A
{
public:
   A( const K& b=0 ) {  a = b;  }
   void operator+=( const A& q );
   const K& getA() {  return a;  }
private:
   K a;
}; // template < typename K > class A
```

Για την περίπτωση που παίρνουμε στιγμιότυπο του *A* για `bool` –δηλαδή για την κλάση `A<bool>`– θέλουμε να έχουμε την πράξη "`||`" αντί για τη "`+`" και τη "`&&`" αντί για τη "`*`". Ο τελεστής που έχουμε εδώ θα πρέπει να γίνεται για την κλάση αυτήν "`||=`"(ας πούμε).

Μπορούμε λοιπόν να πούμε:

```
template < typename K >
void A<K>::operator+=( const A<K>& q )
{  a += q.a;  }
```

και να εξειδικεύσουμε:

```
template<>
void A<bool>::operator+=( const A<bool>& q )
{  a = a || q.a;  }
```

Αυτό είναι προτιμότερο από το να γράψουμε μια ολόκληρη κλάση:

```
template<> class A< bool > { /* . . . */ }; // ◀
```

Μια δυνατότητα που υπάρχει για περιγράμματα κλάσεων αλλά όχι για περιγράμματα συναρτήσεων είναι οι *προκαθορισμένες τιμές παραμέτρων του περιγράμματος*. Για παράδειγμα, μετά τον ορισμό:

```
template < typename T1=int, typename T2=char, int n=1 >
class C { /* . . . */ }; // C
```

μπορείς να δώσεις δηλώσεις όπως:

```
C<> ob0;
C< double > ob1;
C< long, int > ob2;
C< unsigned, wchar_t, 2 > ob3;
```

που είναι ισοδύναμες με:

```
C< int, char, 1 > ob0;
C< double, char, 1 > ob1;
C< long, int, 1 > ob2;
C< unsigned int, wchar_t, 2 > ob3;
```

Όπως είναι φυσικό, ισχύει αυτό που είπαμε στην §14.2 για παραμέτρους συναρτήσεων: «*Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.*» Δηλαδή: μη σκεφτείς να γράψεις "`C<3> x;`" επειδή βαριέσαι να γράψεις το σωστό: "`C<int, char, 3> x;`"

25.5.1 Μερική Εξειδίκευση Περιγράμματος Κλάσης

Όπως είδαμε, σε ένα περίγραμμα συνάρτησης μπορείς να επιφορτώσεις ένα ή περισσότερα περιγράμματα συναρτήσεων ή/και μια ή περισσότερες απλές συναρτήσεις. Επιφόρτωση κλάσεων ή περιγραμμάτων κλάσεων δεν υπάρχει αλλά στην εξειδίκευση περιγραμμάτων κλάσεων έχεις μια δυνατοτητα που δεν υπάρχει για την εξειδίκευση περιγραμμάτων συναρτήσεων: η **μερική εξειδίκευση** (partial specialization) *που δίνει περίγραμμα κλάσης και όχι κλάση.*

Δες μερικά παραδείγματα (από το πρότυπο C++03). Ας πούμε ότι έχουμε το πρωτεύον περίγραμμα:

```
template< typename T1, typename T2, int I > class A { /* . . . */ };
```

που έχει δύο παραμέτρους-τύπους και μια ακέραιη παράμετρο.

α) Μια εξειδίκευση είναι η εξής:

```
template< typename T1, typename T2, int I >
class A< T1*, T2, I > { /* . . . */ };
```

Εδώ περιορίζεται ο τύπος της πρώτης παραμέτρου του περιγράμματος: είναι τύπος βέλους.

β) Η εξειδίκευση:

```
template< typename T1, typename T2, int I >
class A< T1, T2*, I > { /* . . . */ };
```

είναι σαν την πρώτη αλλά θέλουμε τύπο βέλους στη δεύτερη παράμετρο.

γ) Μια άλλη εξειδίκευση είναι η εξής:

```
template< typename T, int I > class A< T, T*, I > { /* . . . */ };
```

όπου οι *T1* και *T2* δεν είναι άσχετες μεταξύ τους: η δεύτερη είναι βέλος προς αντικείμενο της πρώτης.

δ) Μια τέταρτη εξειδίκευση είναι η:

```
template< typename T > class A< int, T*, 5 > { /* . . . */ };
```

Εδώ καθορίζεται η πρώτη παράμετρος να είναι ο τύπος "`int`" και η τρίτη να έχει τιμή "`5`".

Θα πρέπει να επιστήσουμε την προσοχή σου στον ορισμό των μεθόδων ενός μερικώς εξειδικευμένου περιγράμματος κλάσης.

Όταν ζητήσουμε ένα στιγμιότυπο ενός περιγράμματος κλάσης τότε γίνεται το εξής: Ο μεταγλωττιστής προσπαθεί να ταιριάσει τις τιμές των παραμέτρων του στιγμιότυπου με αυτά των εξειδικεύσεων.

- Αν βρει μια μόνον εξειδίκευση που να ταιριάζει παράγει το στιγμιότυπο από αυτήν.
- Αν βρει δύο ή περισσότερες εξειδικεύσεις επιλέγει την πιο εξειδικευμένη. Αν δεν μπορεί να διαλέξει βγάζει λάθος.
- Αν δεν βρει εξειδίκευση που να ταιριάζει θα βγάλει το στιγμιότυπο από το αρχικό περίγραμμα.

Ας πούμε ότι έχουμε το περίγραμμα και τις εξειδικεύσεις που είδαμε παραπάνω και έχουμε την

```
A< int, int, 1 > a1;
```

Οι παράμετροι του στιγμιότυπου δεν ταιριάζουν με οποιαδήποτε από τις εξειδικεύσεις. Έτσι, θα παραχθεί από το αρχικό περίγραμμα βάζοντας "`int`" στους *T1*, *T2* και "`1`" στην *I*.

Ας δούμε τώρα τη δήλωση:

```
A< int, int*, 1 > a2;
```

Οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στους *T1* και *T2* και "`1`" στην *I*.
- Με την εξειδίκευση (γ) αν βάλουμε "`int`" στον *T* και "`1`" στην *I*.

Η (γ) είναι πιο εξειδικευμένη και από αυτήν θα παραχθεί το στιγμιότυπο.

Για τη δήλωση:

```
A< int, char*, 5 > a3;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`5`" στην *I*.
- Με την εξειδίκευση (δ) αν βάλουμε "`char`" στον *T*.

Το στιγμιότυπο θα παραχθεί από τη (δ) που είναι πιο εξειδικευμένη.

Για τη δήλωση:

```
A< int, char*, 1 > a4;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν μόνο με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`1`" στην *I*.

Τέλος, για τη δήλωση:

```
A< int*, int*, 2 > a5;
```

οι παράμετροι ταιριάζουν με την (α) και τη (β) που όμως είναι το ίδιο εξειδικευμένες. Εδώ ο μεταγλωττιστής, μη μπορώντας να δημιουργήσει τον τύπο της *a5*, θα βγάλει λάθος.

Το πρόγραμμα που ακολουθεί δοκιμάζει τα παραπάνω:

```
#include <iostream>

using namespace std;

template< typename T1, typename T2, int I >
struct A
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1, T2, I >::f()  {   cout << "f 0" << endl;  }

template<> void A< int, int, 0 >::f()
{   cout << "f 0 ovrld" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1*, T2, I >
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1*, T2, I >::f()  {  cout << "f a" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1, T2*, I >
{
   void f();
};
```

```
template< typename T1, typename T2, int I >
void A< T1, T2*, I >::f()  {  cout << "f b" << endl;  }

template< typename T, int I >
struct A< T, T*, I >
{
   void f();
};

template< typename T, int I >
void A< T, T*, I >::f()  {  cout << "f c" << endl;  }

template< typename T >
struct A< int, T*, 5 >
{
   void f();
};

template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }

int main()
{
   A< int, int, 0 > a0;
   A< int, int, 1 > a1;
   A< int, int*, 1 > a2;
   A< int, char*, 5 > a3;
   A< int, char*, 1 > a4;
   A< int*, int*, 2 > a5;

   a0.f();
   a1.f();
   a2.f();
   a3.f();
   a4.f();
   a5.f();
}
```

Για την "`A< int*, int*, 2 > a5;`" ο μεταγλωττιστής[5] θα μας βγάλει λάθος: "`Too many candidate template specializations from 'A<T1,T2,2>' in function main()`". Αν διαγράψουμε αυτήν τη δήλωση καθώς και την εντολή "`a5.f();`" όλα πάνε καλά και παίρνουμε αποτέλεσμα:

```
f 0 ovrld
f 0
f c
f d
f b
```

Τώρα κάνουμε την εξής δοκιμή: αφαιρούμε την

```
template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }
```

Αυτήν τη φορά θα διαμαρτυρηθεί ο συνδέτης: "`Error: Unresolved external 'A<int, char *, 5>::f()' referenced from …`" Όπως λέγαμε πιο πριν «*η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!*» Το ίδιο ισχύει και για τα περιγράμματα που προκύπτουν από μερικές εξειδικεύσεις ενός πρωτεύοντος περιγράμματος: *δεν κληρονομούν ότιδήποτε από αυτό.*

[5] Borland C++ v.5.5.

25.6 Περιγράμματα και Κληρονομιές

Τι δυνατότητες κληρονομιάς έχουν τα περιγράμματα κλάσεων; Ό,τι μπορείς να σκεφτείς.

25.6.1 Κληρονομιά Κλάσης από Περίγραμμα Κλάσης

Μια κλάση μπορεί να κληρονομεί *στιγμιότυπο* περιγράμματος κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

class D :  public B< char >
{
public:
   D( char a1=0, int a2=0 )
      :  B<char>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<char>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Πρόσεξε την προετοιμασία που κάναμε στη βασική κλάση (περίγραμμα) για να μπορεί να κληρονομηθεί: έχουμε δηλώσει **virtual** τον καταστροφέα. Ακόμη κάναμε το ίδιο και στη μέθοδο *display()* αφού περιμένουμε –σε περίπτωση κληρονομιάς– να υπάρξει μέθοδος της παράγωγης κλάσης με το ίδιο όνομα που θα πρέπει να υπερισχύσει.

25.6.2 Κληρονομιά Περιγράμματος Κλάσης από Κλάση

Μερικές φορές μπορεί να θέλεις να χρησιμοποιήσεις –σε ένα περίγραμμα που γράφεις– εργαλεία που έχεις σε κάποια κλάση. Ένας τρόπος είναι και η κληρονομια. Για παράδειγμα:

```
class B
{
public:
   B( int a=0 ) :  bCount( a ) { };
   virtual ~B() { };
   unsigned int getCount() const {  return bCount;  }
// . . .
protected:
   unsigned int bCount;
}; // B

template < typename C, size_type sz=10 >
class D :  public B
{
public:
   D( int a=sz )
   {  if ( a <= 0 )  throw a;
      dArr = new C[a];   bCount = a;  };
   virtual ~D() {  delete[] dArr;  };
// . . .
private:
   C* dArr;
}; // D
```

25.6.3 Κληρονομιά Περιγράμματος Κλάσης από Περίγραμμα

Η πιο ενδιαφέρουσα από τις τρεις περιπτώσεις είναι η κληρονομιά περιγράμματος κλάσης από άλλο περίγραμμα κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

template < typename C >
class D :  public B< C >
{
public:
   D( C a1=0, int a2=0 )
      :  B<C>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<C>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Όπως βλέπεις και εδώ προετοιμάζουμε για κληρονομιά το περίγραμμα κλάσης *B*, από το οποίο θα προέλθει η βασική κλάση. Από αυτά που έχουμε μέχρι εδώ δεν έχει ορισθεί κάποια κλάση.

Κλάση δημιουργείται όταν εκτελεσθεί η

```
   D<char> x( 'x', 7 );
```

όπου ζητείται κλάση-στιγμιότυπο του *D* για τιμή παραμέτρου "`char`". Πριν από αυτήν θα πρέπει να δημιουργηθεί η `B<char>` την οποία θα κληρονομήσει η `D<char>`.

Ας δούμε και ένα πολύ απλό παράδειγμα από την STL. Για να μπορούμε να περάσουμε μια συνάρτηση (κατηγόρημα) θα πρέπει να τη «μετατρέψουμε» σε κλάση· για την ακρίβεια σε συναρτησοειδές (§22.6.2). Η C++ έχει έτοιμα μερικά περιγράμματα τέτοιων συναρτησοειδών και μπορείς να τα χρησιμοποιήσεις βάζοντας στο πρόγραμμά σου "`#include <functional>`". Εκεί μπορείς να βρεις τα εξής περιγράμματα (μεταξύ άλλων):

```
template < typename Arg1, typename Arg2, typename Result >
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};


template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Το *binary_function* είναι ένα «άδειο» περίγραμμα ενώ το δεύτερο είναι περίγραμμα συναρτησοειδούς. Μέσα στο πρόγραμμά σου μπορείς να γράψεις:

```
#include <functional>
// . . .
using namespace std;
// . . .
   less<int> lt;
```

Η "`less<int>`" είναι κλάση (συναρτησοειδές) και η "`lt(1,-1)`" θα δώσει τιμή "`false`".

Δες και μιαν άλλην περίπτωση:

```
template < typename C1, typename C2 >
class D1 :  public B< C1 >
{
public:
   D1( C1 a1=0, C2 a2=0 )
      :  B<C1>( a1 ), md( a2 ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B<C1>::display( tout );  tout << "  " << md;  }
private:
   C2 md;
}; // D1
```

Εδώ, η πρώτη παράμετρος (*C1*) καθορίζει το στιγμιότυπο της βασικής το οποίο θα κληρονομήσει η παράγωγη ενώ η δεύτερη (*C2*) αναφέρεται μόνον στην παράγωγη.

Τέλος, θα αναφέρουμε δύο πιθανές δυσάρεστες εκπλήξεις που σε περιμενουν στην περίπτωση κληρονομιάς περιγράμματος από περίγραμμα.

1. Ας πούμε ότι στο «βασικό περίγραμμα» *B* έχεις μια συνάρτηση-μέλος

```
template < typename T >
class B
{
public:
   void f() {  /* . . . */  }
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* καλεί την *f()*:

```
   void g() { /* . . . */ f(); /* . . . */ }
```

Είναι πιθανό ότι ο μεταγλωττιστής σου (π.χ. gcc) δεν θα το επιτρέψει.

Πώς θα τον πείσουμε να το δεχτεί; Γράφοντας την κλήση της *f()* ως "`this->f()`" ή "`B<C>::f()`".

2. Ας πούμε ότι στο «βασικό περίγραμμα» *B* ορίζεις έναν τοπικό δικό σου τύπο, π.χ. μια κλάση:

```
template < typename T >
class B
{
public:
   struct Qaz {  /* . . . */  };
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ σε κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* δηλώνεις μια μεταβλητή τύπου *Qaz*:

```
   void g() { Qaz x; /* . . . */ }
```

Και αυτό μπορεί να απορριφθεί από τον μεταγλωττιστή σου.

Πώς διορθώνεται; Έτσι:

```
   void g() { typename B<C>::Qaz x; /* . . . */ }
```

Για περισσότερα παραπέμπουμε στο (Cline 1999), §35.18 και 35.19.

25.7 Περιέχουσες Κλάσεις

Περιέχουσα κλάση (container class) είναι *περίγραμμα* για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων. **Περιέχον** (container) είναι ένα αντικείμενο στιγμιοτύπου περιέχουσας κλάσης. Παραδείγματα:

- Η κλάση *SList* που είναι συλλογή αντικειμένων κλάσης *GrElmn* (§21.11) θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.

- Οι κλάσεις *CourseCollection*, *StudentCollection* και *StudentInCourseCollection* –που είδαμε στα Project 4 και 6 και είναι συλλογές αντικειμένων κλάσης *Course*, *Student* και *StudentInCourse* αντιστοίχως– θα μπορούσαν να είναι στιγμιότυπα μιας τέτοιας κλάσης.[6]
- Όλοι οι πίνακες που έχουμε δει στα παραδείγματά μας είναι συλλογές αντικειμένων του ίδιου τύπου. Βέβαια είναι λιγο δύσκολο να τους δούμε ως στιγμιότυπα ενός περιγράμματος κλάσης.

Τώρα θα πεις: Ένα αντικείμενο κλάσης *CourseCollection* περιέχει πίνακα με (βέλη προς) αντικείμενα κλάσης *Course*. Τι έχουμε εδώ; Περιέχον μέσα σε περιέχον; Όχι! Το περιέχον είναι ο πίνακας. Το αντικείμενο που τον περιέχει είναι το εργαλείο μας για να κάνουμε τον χειρισμό του πίνακα σύμφωνα με ορισμένους κανόνες (που προκύπτουν από την αναλλοίωτη). Λέμε ότι η κλάση *CourseCollection* (ακριβέστερα: το περίγραμμα κλάσης του οποίου στιγμιότυπο θα ήταν η *CourseCollection*) είναι **προσαρμογέας περιέχοντος** (container adaptor) ή **περιέχον-προσαρμογέας**. Αυτά ξεκαθαρίζουν στο παρακάτω

Παράδειγμα – Μια στοίβα σε πίνακα

Με τις

```
K   arr[sz];
int top;
```

και με δύο συναρτήσεις –*push()* και *pop()*– μπορούμε να υλοποιήσουμε μια **στοίβα** ((LIFO) stack) με στοιχεία τύπου *K*. Σε κάθε στιγμιότυπο όπου καθορίζεται ο τύπος *K* (π.χ. `int` ή *Date* ή *Course* κλπ) ο πίνακας είναι μια συλλογή αντικειμένων αυτού του τύπου. Έχουμε λοιπόν ένα περιέχον.

Αλλά το περιέχον μας έχει πρόβλημα: Πολλά μπορούμε να κάνουμε με τα στοιχεία ενός πίνακα! Έτσι:

- Σε μια στοίβα η θέση της κορυφής (*top*) μπορεί να μεταβάλλεται μόνον από τις *push()* και *pop()* κατά "`+1`" ή "`-1`" αντιστοίχως. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`top = 47`" ή "`top += 19`".
- Ακόμη, σε μια στοίβα μπορούμε να «δούμε» μόνο το στοιχείο που δείχνει η κορυφή. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`arr[47] = ...`" ή "`q = arr[29]`".

Για να εμποδίσουμε τέτοια λάθη «κρύβουμε» τα `arr` και `top` μέσα σε έναν προσαρμογέα περιέχοντος:

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
// . . .
   void push( const K& v );
   void pop();
   const K& getTop() const;
// . . .
private:
   K   arr[sz];
   int top;
}; // StackT
```

Ολόκληρο το περίγραμμα μπορεί να είναι:[7]

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
   StackT() { top = -1; }  // empty stack
```

[6] Θα το δούμε σε επόμενα Project.

[7] Το κατηγόρημα για την άδεια στοίβα θα το βαφτίζαμε *isEmpty()*, αλλά προτιμούμε να ακολουθήσουμε την ονοματολογία της STL.

```
   StackT( const StackT& aSt );
   void push( const K& v );
   void pop();
   const K& getTop() const;
   bool empty() const {  return ( top == -1 );  }
   bool isFull() const {  return ( top == sz-1 );  }
private:
   K   arr[sz];
   int top;
}; // StackT

struct StackTXptn :  public exception
{
   static const int fnLength = 100;
   enum { stackFull, stackEmpty };
   char funcName[fnLength];
   int  errorCode;

   StackTXptn( const char* fn, int ec ) :  errorCode(ec)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
}; // StackTXptn

template < typename K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   if ( top == sz-1 )
      throw StackTXptn( "push", StackTXptn::stackFull );
   ++top;  arr[top] = v;
} // push

template < typename K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
} // pop

template < typename K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( top == -1 )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return arr[top];
} // getTop
```

Το περίγραμμα είναι πολύ απλό, αλλά όχι και τόσο χρήσιμο. Πρόβλημα; Το σταθερό μέγεθος του πίνακα. Μπορούμε να τον κάνουμε δυναμικό και να χρησιμοποιήσουμε τη *renew()*. Αλλά, αν πρέπει να κάνουμε κάθε τόσο αντιγραφές ολόκληρου του πίνακα οι καθυστερήσεις θα είναι μεγάλες.

Να και ένα προγραμματάκι που δοκιμάζει την παραπάνω υλοποίηση:

```
int main()
{
   StackT<int> intStack;

   cout << "Pushing integers on intStack" << endl;
   for ( int i = 0; i < 5; i++ )
   {
      intStack.push( i );       // put items in the stack
      cout << i << ' ';
   }
   cout << endl;

   cout << endl << "Popping integers from intStack" << endl;
   while ( !intStack.empty() )
   {
```

```
      cout << intStack.getTop() << ' ';
      intStack.pop();                    // remove items
   }
   cout << endl;

   StackT<char> charStack;
   cout << endl << endl
        << "Pushing characters on charStack" << endl;
   for ( char c = 'A'; c < 'E'; c++ )
   {
      charStack.push( c );      // put items in the stack
      cout << c << ' ';
   }
   cout << endl;

   cout << endl << "Popping characters from charStack" << endl;
   while ( !charStack.empty() )
   {
      cout << charStack.getTop() << ' ';
      charStack.pop();   // remove items
   }
   cout << endl;
} // main
```

Αυτή η στοίβα ήταν το «ορεκτικό». Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.

Παρατήρηση: ▶

Όλος ο κόσμος θα έγραφε ένα περίγραμμα σαν

```
template < typename K, size_type sz >
const K& StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
   return arr[top+1];
} // pop
```

Γιατί εδώ γράψαμε δύο χωριστές μεθόδους-περιγράμματα; Για ασφάλεια στις εξαιρέσεις!

Ας εξηγηθούμε: έστω ότι η *v* είναι μεταβλητή κλάσης *K*, το *aStack* είναι αντικείμενο κλάσης `StackT<K>` και ρίχνεται εξαίρεση κατά την εκτέλεση της:

```
v = aStack.pop();
```

Έστω ότι η εκχώρηση στην κλάση *K* έχει ισχυρή εγγύηση ασφάλειας πράγμα που σημαίνει ότι η τιμή της *v* δεν θα αλλάξει. Αλλά τι γίνεται με το *aStack*; Η τιμή που αφαιρέθηκε από την κορυφή του χάθηκε!

Με την υλοποίηση που δώσαμε η ίδια δουλειά θα γίνει με τις:

```
v = aStack.getTop();
aStack.pop();
```

Αν κατά την εκτέλεση της πρώτης εντολής ριχτεί εξαίρεση η δεύτερη εντολή δεν θα εκτελεσθεί.

Πάντως αν το πρόγραμμά σου έχει σχεδιαστεί για πολυνηματική εκτέλεση και δύο ή περισσότερα νήματα παίρνουν τιμές από το *aStack* υπάρχει άλλο πρόβλημα. Οι δύο λειτουργίες θα πρέπει να γίνονται από *μια* συνάρτηση που θα πρέπει να έχει και **ατομικότητα** (atomicity), που με απλά λόγια σημαίνει ότι εκτελείται σαν *μια εντολή* (πριν τελειώσει η εκτέλεσή της δεν μπορεί να αλλάξει νήμα εκτέλεσης). Η C++11, αλλά και οι API των ΛΣ, σου δίνουν εργαλεία για να λύσεις παρόμοια προβλήματα.◀

25.7.1 Το Περίγραμμα μιας Λίστας

Πιο πάνω είπαμε «*Η κλάση SList, που είναι συλλογή αντικειμένων κλάσης GrElmn (§21.11), θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.*» Ας το δούμε λοιπόν. Θα ξεκινήσουμε από την τελευταία μορφή της *SList*, όπως την είδαμε στην §21.11 και στην §22.9 όπου κάναμε εισαγωγή των προσεγγιστών.

Όπως στη *BString* αντικαταστήσαμε το "`char`" με "*K*" εδώ θα αντικαταστήσουμε το "*GrElmn*" με "*K*". Πριν από αυτό όμως χρειάζονται μερικές αλλαγές. Οι μέθοδοι *find1Elmn()*, *get1Elmn()* και *delete1Elmn()* έχουν μια παράμετρο τύπου `int` αφού περιμένουν εκεί ένα όρισμα, τον ατομικό αριθμό, που είναι και το κλειδί για τη *GrElmn*. Αν τις αφήσουμε όπως είναι το περίγραμμα θα μπορεί να δώσει μόνον στιγμιότυπα-συλλογές αντικειμένων με κλειδί κάποιον ακέραιο. Φυσικά δεν θέλουμε κάτι τέτοιο και τις αλλάζουμε σε:

```
bool SList::find1Elmn( const GrElmn& aDtItem ) const
{
   return ( findPtr(aDtItem) != slTail );
} // SList::find1Elmn

const GrElmn& SList::get1Elmn( const GrElmn& aDtItem ) const
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::get1Elmn

void SList::delete1Elmn( const GrElmn& aDtItem )
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::delete1Elmn
```

Στη συνέχεια θα ξαναδούμε αυτές τις μεθόδους.

Παρατήρηση: ▶

Τουλάχιστον η *get1Elmn()* φαίνεται κάπως περίεργη: Ζητούμε να μας επιστρέψει κάτι που το δίνουμε ως όρισμα όταν την καλούμε; Όχι βέβαια. Η κλήση της θα γίνεται συνήθως ως εξής:

```
tmp = lst.get1Elmn( GrElmn(aa) );
```

Δηλαδή στο *aDtItem* θα περνάει ένα αντικείμενο που θα έχει μόνο το κλειδί. Για τα υπόλοιπα θα στηριζόμαστε στον ορισμό της ισότητας (`operator==()`) που έχουμε για τη *GrElmn*.◀

Κάνοντας τώρα την αντικατάσταση του "*GrElmn*" με "*K*" έχουμε:

```
template < typename K >
class SListT
{
private:
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      K& operator*();
      K* operator->();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
```

```
      ListNode* iPNode;
   }; // Iterator

   SListT();
   SListT( const SListT& rhs );
   ~SListT();
   SListT& operator=( const SListT& rhs );
   void swap( SListT& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
   bool find1Elmn( const K& aDtItem ) const;
   const K& get1Elmn( const K& aDtItem ) const;
   void insert1Elmn( const K& aDtItem );
   void delete1Elmn( const K& aDtItem );
   void save( ostream& bout,
              void (*wrProc)(const K&, ostream&) ) const;
   void display( ostream& tout ) const;
private:
   ListNode* slHead;
   ListNode* slTail;

   ListNode* findPtr( const K& aDtItem ) const;
   void push_front( const K& aData );
}; // SList
```

και η αντίστοιχη κλάση εξαιρέσεων:

```
struct SListTXptn
{
   enum { allocFailed, notFound, listEnd };
   char  funcName[100];
   int   errorCode;
   int   errIntVal;
   SListTXptn( const char* fn, int ec, int iv=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errIntVal = iv;  }
}; // SListXptn
```

Ας δούμε τα περιγράμματα συναρτήσεων που επιφορτώνουν τους τελεστές "++", "*" και "->" για τον *SListT*<*K*>::*Iterator*:

```
template < typename K >
SListT<K>::Iterator& SListT<K>::Iterator::operator++()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator++",
                        SListTXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SListT<K>::Iterator::operator++

template < typename K >
K& SListT<K>::Iterator::operator*()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator*",
                        SListTXptn::listEnd );
   return ( iPNode->lnData );
} // SListT<K>::Iterator::operator*

template < typename K >
K* SListT<K>::Iterator::operator->() const
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator->",
                       SListXptn::listEnd );
   return ( &(iPNode->lnData) );
} // SList::Iterator::operator->
```

Στη συνέχεια θα δούμε μερικές από τις μεθόδους του περιγράμματος ξεκινώντας με την

```
template < typename K >
void SListT<K>::push_front( const K& aData )
{
   ListNode* pnln;
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListTXptn( "push_front", SListTXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = slHead;
   slHead = pnln;
} // SList::push_front
```

Εδώ τώρα πρόσεξε τα εξής:

- Για να εκτελεσθεί η "`new ListNode`" θα πρέπει να δημιουργηθεί και το μέλος *lnData* από τον ερήμην δημιουργό της *K*.
- Εκτέλεση της "`pnln->lnData = aData`" σημαίνει εκτέλεση του (αντιγραφικού) τελεστη εκχώρησης της *K*.

Επομένως:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***

Βέβαια θα πεις ότι θα μπορούσαμε να αποφύγουμε τον ερήμην δημιουργό και τον τελεστή εκχώρησης αν εφοδιάσουμε την *ListNode* με δημιουργούς:

```
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
      ListNode()
         : lnNext( 0 ) { };
      explicit ListNode( const K& aData )
         : lnData( aData ), lnNext( 0 ) { };
   }; // ListNode
```

και αντί για:

```
   pnln = new ListNode;  pnln->lnData = aData;
```

να γράφουμε

```
   pnln = new ListNode( aData );
```

Εδώ καλείται ο δεύτερος δημιουργός που –με τη σειρά του– θα καλέσει τον δημιουργό αντιγραφής (`lnData(aData)`) της *K*.

Σωστό! Αλλά πρόσεξε το εξής: Εμείς κάνουμε αυτές τις επισημάνσεις για να κατάλάβεις τις απαιτήσεις που (θα δούμε ότι) βάζει η STL για τις κλάσεις που πιθανόν θα θελήσεις να βάλεις σε κάποιο από τα περιέχοντά της. Εκεί δεν θα έχεις επιλογή για το πώς θα γράψεις το περιέχον· είναι έτοιμο. Έτσι, η παρατήρηση που κάναμε πιο πάνω όχι μόνον δεν αναιρεί τους περιορισμούς που είχαμε αλλά βάζει έναν ακόμη:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***

Ας σκεφτούμε τώρα λιγάκι τη *findPtr()*. Κατ' αρχήν θα μπορούσαμε να γράψουμε το περίγραμμα:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   slTail->lnData = aDtItem;
   fv = slHead;
   while ( (fv->lnData) != aDtItem )  fv = fv->lnNext;
   return fv;
} // SListT<K>::findPtr
```

αλλά εκείνη η "`slTail->lnData = aDtItem`" –αντιγραφή ολόκληρου αντικειμενου κλάσης *K*–[8] έχει πρόβλημα. Αν μπορούμε να θεωρήσουμε ότι ένα αντικείμενο *GrElmn* είναι μικρό και η αντιγραφή του δεν είναι χρονοβόρα δεν μπορούμε να πούμε το ίδιο πράγμα για το αντικείμενο οποιασδήποτε κλάσης *K* θα κληθεί να φιλοξενήσει το *SListT*.

Εγκαταλείπουμε λοιπόν την αντιγραφή στον φρουρό και επιστρέφουμε στις διπλές συγκρίσεις:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   for ( fv = slHead; fv != slTail; fv = fv->lnNext )
      if ( fv->lnData == aDtItem ) break;
   return fv;
} // SListT<K>::findPtr
```

Αυτό το περίγραμμα όμως είναι σαν να μας λέει: «Άλλαξε τα βέλη, σε προσεγγιστές!» Ας το κάνουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::findIt( const K& aDtItem ) const
{
   Iterator fv;
   for ( fv = begin(); fv != end(); ++fv )
      if ( *fv == aDtItem ) break;
   return fv;
} // SListT<K>::findIt
```

Αν χρησιμοποιήσουμε τη νέα *findIt()* στη *get1Elmn()* θα έχουμε:

```
template < typename K >
const K& SListT<K>::get1Elmn( const K& aDtItem ) const
{
   Iterator itToNode( findIt(aDtItem) );
   if ( itToNode == end() )
      throw SListTXptn( "get1Elmn", SListTXptn::notFound,
                        aDtItem.getANumber() );
   return *itToNode;
} // SList<K>::get1Elmn
```

Εδώ όμως η *findIt()* δεν μας χρειάζεται· μπορούμε να χρησιμοποιήσουμε την *std*::*find()* –που είδαμε στην §22.9– και να γράψουμε:

```
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
```

Μπορείς να τροποποιήσεις με τον ίδιο τρόπο και τις *insert1Elmn()* και *delete1Elmn()*. Για την τελευταία θα χρειαστείς κάτι ακόμη: Στην περιοχή "`private`" της *Iterator* ορίζουμε:

```
// . . .
   private:
      ListNode* iPNode;
      ListNode* getPNode() const {  return iPNode;  }
   }; // Iterator
```

Έτσι, στην αρχή της *delete1Elmn()* θα έχουμε:

```
template < typename K >
void SListT<K>::delete1Elmn( const K& aDtItem )
{
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
   if ( itToNode != end() ) // υπάρχει
   {
      ListNode* ptrToNode( itToNode.getPNode() );
      ListNode* pnln( ptrToNode->lnNext );
   // . . .
```

[8] Ενώ χρειαζόμαστε μόνο το κλειδί!

Παρατήρηση:▶
Σχετικώς με τον φρουρό: οι αντιγραφές δεν είναι το μόνο πρόβλημα. Αν τα αντικείμενα της περιεχόμενης κλάσης είναι μεγάλα η ύπαρξη του φρουρού είναι και σπατάλη μνήμης.◀

25.7.1.1 Ο Καταστροφέας

Τέλος, ας πούμε και δυο λόγια για τον καταστροφέα. Το μόνο πράγμα που άλλαζει είναι η επικεφαλίδα από "`SList::~SList()`" σε "`template <typename K> SListT<K>::~SListT()`". Υπάρχει όμως ένα άλλο ερώτημα σχετικώς με τη δήλωση: Μήπως θα έπρεπε να τον δηλώσουμε "`virtual`" ώστε να υπάρχει δυνατότητα να κληρονομηθούν οι κλάσεις-στιγμιότυπα του περιγράμματος; Αυτό είναι ένα ερώτημα που τίθεται από πολλούς για τα περιέχοντα της SLT και μάλιστα με κάπως διαφορετικό τρόπο: πώς «ξέφυγε» μια τόσο σοβαρή παράλειψη και οι καταστροφείς των περιεχόντων της STL δεν είναι "`virtual`";

Φυσικά δεν πρόκειται για λάθος ούτε για παράλειψη και ας το σκεφτούμε. Στη συνέχεια θα δούμε υλοποίηση στοίβας με μια λίστα. Για να δηλώσουμε μια στοίβα ακεραίων μπορούμε να κάνουμε ένα από τα εξής:

- Να ορίσουμε το περίγραμμα:

```
template < typename K >
class StackT
{
public:
// . . .
private:
   SListT<K> cont;
}; // StackT
```

και στη συνέχεια να δηλώσουμε:

```
StackT< int > aStack;
```

- Να ορίσουμε:

```
class IntStack : public SListT<int>  { /*. . . */ }
```

και στη συνέχεια:

```
IntStack aStack;
```

Τι θα πρέπει να προτιμήσουμε; Για να απαντήσουμε θα πρέπει να καταλάβουμε το νόημα της κάθε επιλογής:

- Στην πρώτη περίπτωση έχουμε το περίγραμμα όπως το είχαμε αρχικώς (εκτός από τη δεύτερη παράμετρο "`size_type sz = 100`") όπου αλλάξαμε το περιέχον από πίνακα σε λίστα.
- Στη δεύτερη περίπτωση λέμε ότι μια στοίβα ακεραίων είναι μια (`is_a`) λίστα ακεραίων.

Η δεύτερη επιλογή έρχεται σε αντίθεση με αυτό που λέγαμε στην §23.14: «*οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και τη σωστή νοηματική σχέση.*» Η στοίβα *μπορεί να υλοποιηθεί* με μια λίστα. Αλλά *δεν είναι λίστα*· ήδη την έχουμε υλοποιήσει με πίνακα.[9]

Στις περισσότερες περιπτώσεις που θα πάει το μυαλό σου σε κληρονομιά από περιέχον ένα σκεπτικό σαν το παραπάνω έχει νόημα. Δεν βάζουμε λοιπόν το "`virtual`" για να αποθαρρύνουμε τέτοιες σκέψεις.

Δηλαδή η κληρονομιά από το περίγραμμα λίστα δεν έχει νόημα ποτέ; Θα είχε νόημα αν κάναμε μια νέα δομή δεδομένων που θα ήταν ειδική περίπτωση της λίστας με απλή σύνδεση.

---

[9] Με την ορολογία της §23.15: αυτά είναι επιχειρήματα υπέρ της σχέσης "`has_a`"· επομένως και υπέρ της κληρονομιάς "`private`".

25.7.2 Η Περιεχόμενη Κλάση

Να καταγράψουμε απολογιστικά τις απαιτήσεις για την περιεχόμενη κλάση που βρήκαμε από το παράδειγμά μας μέχρι τώρα:

- ***Πρώτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Δεύτερη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει δημιουργό αντιγραφής.***
- ***Τρίτη Απαίτηση: Για τον περιεχόμενο τύπο K πρέπει να έχει ορισθεί ο τελεστής εκχώρησης.***

Φυσικά, χωρίς να καταγράψουμε, χρησιμοποιήσαμε το γεγονός ότι:

- ***Τέταρτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

Αυτές οι απαιτήσεις τίθενται για όλα τα περιέχοντα της STL. Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Αν, ας πούμε, θέλεις να κάνεις αναζητήσεις τιμών θα πρέπει να έχεις τη δυνατότητα για συγκρίσεις.

25.7.3 Ένα Πρόβλημα, μια Λύση και ένα Άλλο Πρόβλημα

Το περίγραμμα *slist*[10] (αλλά και το *list* της STL) έχει μέθοδο:

```
iterator erase( iterator pos )
```

Αν *a* είναι αντικείμενο κλάσης *slist*<*K*> και *p* ένας *slist*<*K*>::*iterator* τότε η εντολή

```
a.erase( p );
```

καταστρέφει τον κόμβο-στόχο του *p*. Το μήκος της λίστας μειώνεται κατά 1 χωρίς να αλλάζουν οι σχετικές θέσεις των άλλων κόμβων. Η μέθοδος επιστρέφει προσεγγιστή με στόχο το στοιχείο της λίστας που ακολουθεί αυτό που διαγράφηκε.

Αν θέλουμε να γράψουμε και εμείς μια μέθοδο

```
Iterator erase( Iterator pos );
```

για το *SlistT* θα αντιμετωπίσουμε το εξής πρόβλημα: «*θα πρέπει να αλλάξουμε την τιμή του lnNext του προηγούμενου κόμβου που δεν ξέρουμε ποιος είναι!*» Στην §21.11 δώσαμε μια λύση –που φαίνεται στο Σχ. 21-1– γράφοντας την *SList*::*erase1Elmn*. Στην περίπτωσή μας θα έχουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::erase( SListT<K>::Iterator pos )
{
   if ( pos == end() )
      throw SListTXptn( "erase", SListTXptn::listEnd );
   ListNode* ptrToNode( pos.getPNode() );
   ListNode* pnln( ptrToNode->lnNext );
   ptrToNode->lnNext = pnln->lnNext;
   if ( pnln == slTail ) slTail = ptrToNode;
                    else ptrToNode->lnData = pnln->lnData;
   delete pnln;
   return pos;
} // SListT<K>::erase
```

Όπως βλέπεις, θα χρειαστούμε μια μέθοδο (`private`) της κλάσης *Iterator* που θα μας δίνει το βέλος που κρύβει ο προσεγγιστής:

```
      ListNode* getPNode() const {  return iPNode;  }
```

Φαίνεται λοιπόν ότι λύσαμε το πρόβλημά μας αλλά πρόσεξε τα προβλήματα της παραπάνω λύσης:

---

[10] της SGI. Στο C++11 προδιαγράφεται περιέχουσα κλάση με το όνομα "`forward_list`" και είναι λίστα με απλή σύνδεση. Η *forward_list* δεν έχει *erase()* με τα παραπάνω χαρακτηριστικά. Έχει μέθοδο *erase_after()* που καταστρέφει τον κόμβο που ακολουθεί τον κόμβο-στόχο.

- Που δείχνει ο `pos`; Στον κόμβο που έδειχνε αλλά που τώρα έχει ως περιεχόμενο αυτό του κόμβου που ακολουθούσε αυτόν που διαγράψαμε! Δηλαδή το `*pos` αλλάζει με τη διαγραφή. Ναι, αλλά ο `pos` είναι παράμετρος τιμής. Αυτή είναι μια περίεργη κατάσταση.
  Αν χρησιμοποιήσεις άλλη μέθοδο και ανακυκλώσεις ακριβώς τον κόμβο-στόχο του `pos` τότε τα πράγματα είναι χειρότερα.
- Αν δεν έχουμε φτάσει στον φρουρό κάνουμε μια αντιγραφή παραπάνω: "`ptrToNode->lnData = pnln->lnData`". Αν τα αντικείμενα του *K* είναι μεγάλα αυτή η αντιγραφή κοστίζει.

Το δεύτερο πρόβλημα μπορείς να το λύσεις –με άλλον αλγόριθμο– αλλά το πρώτο δεν λύνεται. Η διαγραφή κόμβου *ακυρώνει* τον προσεγγιστή. Αυτό είναι ένα γενικότερο πρόβλημα με τις περιέχουσες κλάσεις: *οι διαγραφές (αλλά και οι εισαγωγές) τιμών μπορεί να ακυρώνουν κάποιους προσεγγιστές*.

25.7.4 Μια Άλλη Στοίβα

Υποσχεθήκαμε ότι «*Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.*» Θα την υλοποιήσουμε χρησιμοποιώντας –όχι πίνακα– αλλά λίστα με απλή σύνδεση.

Στην προηγούμενη υποπαράγραφο είδαμε ένα «άδειο» σχέδιο του περιγράμματος (*StackT*) που θα γράψουμε. Το μοναδικό κρυμμένο μέλος είναι:

```
   SListT<K> cont;
```

Ο ερήμην δημιουργός είναι απλούστατος:

```
   StackT() { };
```

αφού μόνη η δήλωση της λίστας δημιουργεί μια κενή στοίβα.

Ο δημιουργός αντιγραφής

```
   StackT( const StackT& aSt );
```

ορίζεται ως:

```
template < class K, size_type sz >
StackT<K,sz>::StackT( const StackT<K,sz>& other )
   :  cont( other.cont )  {  }
```

και έχει λυμένα τα προβλήματά του από τον δημιουργό αντιγραφής της λίστας.

Ο (σωστός) τελεστής εκχώρησης της λίστας μας επιτρέπει να μην γράψουμε τελεστή εκχώρησης για τη στοίβα.

Η *push()* ορίζεται ως εξής:

```
template < class K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   try {  cont.push_front( v );  }
   catch( SListTXptn& x )
   {  throw StackTXptn( "push", StackTXptn::stackFull );  }
} // StackT<K,sz>::push
```

αλλά εδώ έχουμε ένα πρόβλημα: Η *SListT*::*push_front()* είναι δηλωμένη "`private`"· θα πρέπει να την κάνουμε "`public`".

Για να ορίσουμε την

```
template < class K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( cont.empty() )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   cont.pop_front();
} // StackT<K,sz>::pop
```

θα πρέπει να ορίσουμε την *SListT*::*pop_front()*:

```
template < typename K >
void SListT<K>::pop_front()
{
   if ( slHead == slTail )
      throw SListTXptn( "pop_front", SListTXptn::listEmpty );
   ListNode* pnln( slHead );
   slHead = slHead->lnNext;
   delete pnln;
} // SListT<K>::pop_front
```

Τώρα, η *getTop()* ορίζεται ως εξής:

```
template < class K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( cont.empty() )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return *cont.begin();
} // StackT<K,sz>::getTop
```

Τέλος, τα δύο κατηγορήματα ορίζονται `inline` ως εξής:

```
   bool isEmpty() const {  return cont.empty();  }
   bool isFull() const {  return false;  }
```

Το δεύτερο είναι λίγο περίεργο; Όχι και τόσο αφού για να γεμίσει η στοίβα θα πρέπει να πάρουμε όλη τη διαθέσιμη μνήμη του υπολογιστή μας. Και τότε, αν αποπειραθούμε να καλέσουμε την *push()* θα πάρουμε εξαίρεση με κωδικό *StackTXptn*::*stackFull* που θα προκληθεί από τη σύλληψη εξαίρεσης με κωδικό *SListTXptn*::*allocFailed* που θα ρίξει η *push_front()*.

Δηλαδή όλες οι αλλαγές είναι στην υλοποίηση και το τμήμα διεπαφής δεν αλλάζει; Ακριβώς! Και δεν θα πρέπει να ξεφορτωθούμε τη δεύτερη παράμετρο ("`size_type sz = 100`") καθώς και την *isFull()*; Μπορούμε να το κάνουμε και έτσι· σε «πραγματικές» εφαρμογές όμως θα πρέπει να σκεφτείς το εξής: αν θέλουμε να περάσουμε στην καινούρια υλοποίηση, θα αλλάξουμε όλα τα προγράμματα που έχουμε γράψει και χρησιμοποιούν το περίγραμμα της στοίβας; Η απάντηση στο ερώτημα αυτό είναι συνήθως «όχι». Βέβαια, σε κάποιο πρόγραμμα που καλεί την *isFull()* μπορεί να συμβεί το εξής: οι

```
   if ( !aStack.isFull() )
      aStack.push( v );
```

μπορεί να προκαλέσουν έγερση εξαίρεσης *StackTXptn* με κωδικό *stackFull*! Ε, συμβαίνουν αυτά όταν κάνουμε βελτιώσεις…

Αλλαγές όμως θα έχουμε στο περίγραμμα *SListT*:

- Η *push_front()* έγινε "`public`" και
- Γράψαμε άλλη μια μέθοδο, την *pop_front()* που είναι και αυτή "`public`".

Αυτές μάλλον δεν προκαλούν προβλήματα σε εφαρμογές που χρησιμοποιούν το περίγραμμα.[11]

Παρατήρηση:▶

Για το πρόγραμμα του παραδείγματος της §25.7 η υλοποίηση με τον πίνακα είναι μάλλον πιο κατάλληλη. Για να το καταλάβεις υπολόγισε τη (σπατάλη σε) μνήμη αν υλοποιήσουμε τις δύο στοίβες με λίστα.◀

25.8 Έξυπνα Βέλη

Στην §16.7 εντοπίσαμε ορισμένα προβλήματα που μπορεί να έχουμε όταν διαχειριζόμαστε δυναμική μνήμη με μεταβλητές-βέλη.

---

[11] Αν κάποιος έγραψε κλάση ή περίγραμμα που να κληρονομεί το *SListT*… Εδώ μπορεί να υπάρχει πρόβλημα.

- **Διαρροή μνήμης** (memory leakage): «*το πρόγραμμά μας έχει δυναμική(-ές) μεταβλητή(-ές) αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί (ούτε να την ανακυκλώσει).*»
- **Μετέωρο βέλος** (pending ή dangling pointer): αν δύο ή περισσότερα βέλη δείχνουν την ίδια δυναμική μεταβλητή και την ανακυκλώσουμε τότε κάποιο(-α) βέλος(-η) μένει(-ουν) *μετέωρο*(-α) με την έννοια ότι δείχνει θέση της μνήμης που δεν ελέγχεται από το πρόγραμμά μας.
- **Πολλαπλή διαγραφή (ανακύκλωση):** Αν το βέλος *p* δείχνει κάποια περιοχή της δυναμικής μνήμης οι εντολές "`delete p; delete p;`" μπορεί να προκαλέσουν καταστροφικό λάθος στο σύστημα διαχείρισης της δυναμικής μνήμης.

Τα **έξυπνα βέλη** (smart pointers) είναι προγραμματιστικά εργαλεία που χρησιμοποιούμε για να ξεπεράσουμε τα προβλήματα που απαριθμήσαμε παραπάνω.

Παράδειγμα έξυπνου βέλους μας δίνει η STL της C++. Αναφερόμαστε στο περίγραμμα κλάσης *auto_ptr*. Αυτό το περίγραμμα κλάσης θα μελετήσουμε εδώ χωρίς να δεσμευόμαστε από κάποια συγκεκριμένη υλοποίηση. Για να μην υπάρχουν μπερδέματα, θα ονομάσουμε το δικό μας περίγραμμα *AutoPtr*.[12]

Η ιδέα είναι να κρύψουμε το βέλος μέσα σε ένα αντικείμενο που θα το κάνουμε να συμπεριφέρεται σαν βέλος: Μετά τις

```
Date*           pd( new Date(2010, 11, 26) );
AutoPtr< Date >  apd( pd );
```

το "`*apd`" και το "`*pd`" θα πρέπει να είναι το ίδιο πράγμα. Το ίδιο ισχύει και για τα "`apd->getYear()`", "`apd->getMonth()`", "`apd->getDay()`" και τα "`pd->getYear()`", "`pd->getMonth()`", "`pd->getDay()`" αντιστοίχως.

Κατά τα άλλα θα ρυθμίσουμε τη συμπεριφορά του *AutoPtr*<*K*> για να αντιμετωπίσουμε τα προβλήματα που παραθέσαμε παραπάνω.

Αυτό το κάναμε ήδη μια φορά με τους προσεγγιστές. Εκεί είδαμε ότι για να έχει το αντικείμενο συμπεριφορά βέλους θα πρέπει η (παραμετρική) κλάση μας να είναι:

```
template < typename K >
class Iterator
{
public:
   explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
   K& operator*() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListTXptn( "Iterator::operator*",
                           SListTXptn::listEnd );
      return ( iPNode->lnData );
   } // operator*
   K* operator->() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListXptn( "Iterator::operator->",
                          SListXptn::listEnd );
      return ( &(iPNode->lnData) );
   } // SList::Iterator::operator->
private:
   ListNode* iPNode;
   ListNode* getPNode() const {  return iPNode;  }
}; // Iterator
```

Ξεκινούμε και εδώ το περίγραμμα κλάσης ως εξής:

---

[12] Στο (Alexandrescu 2001) εξετάζονται άλλες επιλογές που υπάρχουν για τη σχεδίαση μιας παραμετρικής κλάσης για έξυπνα βέλη. Μπορείς να βρεις το Κεφ. 7 (για τα έξυπνα βέλη) αυτού του βιβλίου στο `http://ptgmedia.pearsoncmg.com/images/9780201704310/samplepages/0201704315.pdf`.

```
template < typename K >
class AutoPtr
{
// . . .
private:
   K* ap;
}; // AutoPtr
```

Η διαφορά μας από τον προσεγγιστή είναι ότι εκεί το βέλος *iPNode* στόχευε έναν κόμβο μέσα στον οποίο υπάρχει το αντικείμενο που μας ενδιαφέρει ενώ τώρα στοχεύει το ίδιο το αντικείμενο.

Ο δημιουργός είναι ολόϊδιος:

```
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
```

Αλλά τώρα θα χρειστούμε –η πρώτη σημαντική διαφορά– και *καταστροφέα που θα ανακυκλώνει τον στόχο*:

```
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

Για την *getPNode()* υπάρχουν δυο διαφορές (με βάση αυτά που ισχύουν για το *auto_ptr*):

- Η πιο σημαντική διαφορά: είναι "`public`".
- Η λιγότερο σημαντική: θα της αλλάξουμε το όνομα σε "*get*".

```
   K* get() const {  return ap;  }
```

Στην επιφόρτωση του "`operator*`" και του "`operator->`" έχουμε πρόβλημα: Όλες οι μέθοδοι της *auto_ptr* έχουν προδιαγραφή εξαιρέσεων "`throw()`". Το C++03 δίνει τις εξής προδιαγραφές για την `operator*()` του *auto_ptr* που ακολουθούμε:

```
X& operator*() const throw();
```

Προϋπόθεση: `get() != 0`

Επιστρέφει: `*get()`

Για την `operator->()` μας λέει:

```
X* operator->() const throw();
```

Επιστρέφει: `get()`

Οι συναρτήσεις που γράψαμε για τον προσεγγιστή μπορεί να ρίξουν εξαίρεση! Εδώ τι θα κάνουμε; Θα ακολουθήσουμε το πρότυπο:

```
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
```

Έτσι, αν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα αντικείμενο *AutoPtr* που έχει *ap* `==` 0 θα έχεις –όταν εκτελείται η `operator*()`– το πρόβλημα που έχεις όταν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα σύνηθες μηδενικό βέλος. Στην περίπτωση του "`->`" η εκτέλεση της `operator->()` γίνεται κανονικώς και το πρόβλημα θα παρουσιαστεί όταν θα δράσει ο "`->`" στο αποτέλεσμα της συνάρτησης.

Ας πούμε ότι έχουμε δηλώσει:

```
   AutoPtr<string> aps( new string("abc") );
```

Οι εντολές

```
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;
```

θα δώσουν:

```
abc
qwerty
```

Μέχρι εδώ έχουμε:

```
template < typename K >
class AutoPtr
{
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

```
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
private:
   K* ap;
}; // AutoPtr
```

Τι έχουμε κερδίσει με αυτά; Αν έχουμε:

```
void f( int v )
{
   int* p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
   delete p;
} // f
```

παρ' όλο που φροντίσαμε να βάλουμε τη "`delete p`", αν ριχτεί εξαίρεση θα έχουμε διαρροή μνήμης. Αν όμως βάλουμε:

```
void f( int v )
{
   AutoPtr<int> p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
} // f
```

τα πράγματα αλλάζουν: Είτε η εκτέλεση της συνάρτησης τελειώσει κανονικώς είτε διακοπεί λόγω ρίψης εξαίρεσης η καταστροφή της *p* ανακυκλώνει και τη μνήμη που κατέχει η *p*. Όπως καταλαβαίνεις, δεν χρειαζόμαστε πια τον `delete`, αφού έχουμε αυτόματη ανακύκλωση.

Θα δούμε τώρα δύο χαρακτηριστικές μεθόδους της *AutoPtr*. Η πρώτη είναι η *release()*. Το πρότυπο μας λέει για τη *release()* της *auto_ptr*:

```
X* release() throw();
```

Επιστρέφει: `get()`

Απαίτηση: το αντικείμενο (`*this`) έχει το μηδενικό βέλος (`NULL`).

Δηλαδή: επιστρέφει την τιμή του «κρυμμένου» βέλους αλλά επιπλέον το μηδενίζει.

```
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
```

Ας δούμε τη διαφορά της από τη *get()* με ένα παράδειγμα. Μετά τη δήλωση

```
   auto_ptr< Date >  apd( new Date(2010, 11, 26) );
```

οι εντολές:

```
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
```

δίνουν:

```
26.11.2010
26.11.2010
26.11.2010
it is NULL
```

Η τελευταία γραμμή δείχνει ότι, μετά την *apd.release()*, το βέλος που κρύβεται μέσα στο *apd* μηδενίζεται. Βέβαια η τιμή του πέρασε στο *pd1*, όπως φαίνεται από την προτελευταία

γραμμή. Με τη "`Date* pd0(apd.get())`" αντιγράφεται η τιμή του βέλους του *apd* στο *pd0* χωρίς να αλλάζει, όπως δείχνουν οι δύο πρώτες γραμμές.

Η δεύτερη μέθοδος είναι η *reset()*:

```
void reset( X* p=0 ) throw();
```

Ενέργεια: Αν `get() != p` τότε `delete get()`.
Απαίτηση: το αντικείμενο (`*this`) έχει το βέλος *p*.
Με αυτήν αλλάζουμε το κρυμμένο βέλος. Αλλά προσοχή: *αν το παλιό βέλος έχει την κυριότητα κάποιου αντικειμένου αυτό ανακυκλώνεται πριν από την αλλαγή της τιμής του βέλους.*

```
   void reset( K* p=0 )
   {  if ( ap != p ) {  delete ap;  ap = p;  }  }
```

Πρόσεξε ότι αν δώσουμε:

```
   api1.reset();
```

ανακυκλώνεται το αντικείμενο που κατέχει το *api1*. Δηλαδή, μπορούμε έτσι να ανακυκλώνουμε τη μνήμη όποτε θέλουμε.

Ας δούμε τώρα τον «δημιουργό αντιγραφής» και τον τελεστή εκχώρησης.

Το C++03 μας λέει:

```
auto_ptr( auto_ptr& a ) throw();
```

Ενέργεια: Εκτελείται η κλήση "`a.release()`".
Απαίτηση: Το `*this` κρατάει το βέλος που επιστρέφει η `a.release()`.

Για το *AutoPtr* γράφουμε:

```
   AutoPtr( AutoPtr& other ) {  ap = other.release();  }
```

που ισοδυναμεί με:

```
   AutoPtr( AutoPtr& other ) {  ap = other.ap;  other.ap = 0;  }
```

Τι δημιουργός αντιγραφής είναι αυτός; Η μοναδική εντολή φέρνει στο *ap* την τιμή του *other.ap* αλλά, επί πλέον, μηδενίζει το *other.ap*· για να γίνεται αυτό *δεν βάλαμε* "`const`" στην παράμετρο. Στην πραγματικότητα εδώ έχουμε έναν **δημιουργό μεταβίβασης** (move constructor).[13]

Για τον τελεστή εκχώρησης οι προδιαγραφές είναι:

```
auto_ptr& operator=( auto_ptr& a ) throw();
```

Προϋπόθεση: Μπορεί να εκτελεσθεί η "`delete get()`".
Ενέργεια: `reset( a.release() )`.
Επιστρέφει: `*this`.

Για το *AutoPtr* μπορούμε να γράψουμε:

```
   AutoPtr& operator=( AutoPtr& rhs )
   {  reset( rhs.release() );  return *this;  }
```

που είναι ισοδύναμο με:

```
   AutoPtr& operator=( AutoPtr& rhs )
   {  if ( ap != rhs.ap )
      {  delete ap;  ap = rhs.ap;  rhs.ap = 0;  }
      return *this;  }
```

Τώρα μπορείς να καταλάβεις γιατί πολλές φορές γράφαμε για «*αντιγραφικό τελεστή εκχώρησης*»· όπως βλέπεις εδώ έχουμε έναν *μεταβιβαστικό* τελεστή εκχώρησης. Αφού η *reset()* βάζει ως νέο βέλος αυτό που επιστρέφει η `rhs.release()` το βέλος του *rhs* δεν αντιγράφεται αλλά μηδενίζεται. Για να είναι αυτό δυνατό η παράμετρος δεν έχει "`const`".

Όπως βλέπεις, όλες οι μέθοδοι έχουν τα χαρακτηριστικά που πρέπει για να γλυτώσουμε από τα προβλήματα που παραθέσαμε στην αρχή:

- Δεν επιτρέπονται αντίγραφα βελών.

[13] Στο C++11 ο δημιουργός μεταβίβασης ορίζεται με πιο συγκεκριμένο τρόπο και έχει την ίδια θέση με τον συνήθη δημιουργό αντιγραφής.

- Όταν αλλάζουμε την τιμή βέλους που έχει την κυριότητα κάποιου δυναμικού αντικειμενου ανακυκλώνεται και το αντικείμενο.
- Κάθε φορά που ανακυκλώνεται το δυναμικό αντικείμενο μηδενίζεται η τιμή του βέλους που το είχε.

Υπάρχει όμως μια εξαίρεση: η *get()*!

Να λοιπόν πώς θα είναι το **AutoPtr.h**:

```
#ifndef _AUTOPTR_H
#define _AUTOPTR_H

template < typename K >
class AutoPtr
{
public:
   explicit AutoPtr( K* p=0 )  {  ap = p;  }
   AutoPtr( AutoPtr& other ) {  ap = other.release();  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
   AutoPtr& operator=( AutoPtr& rhs )
   {  reset( rhs.release() );  return *this;  }
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
   void reset( K* p=0 )
   {  if ( ap != p ) {  delete ap;  ap = p;  }  }
private:
   K* ap;
}; // AutoPtr

#endif // _AUTOPTR_H
```

Το παρακάτω πρόγραμμα περιέχει όλες τις δοκιμές που είδαμε παραπάνω:

```
#include <iostream>
#include <string>

#include "AutoPtr.h"
#include "Date.cpp"

using namespace std;

int main()
{
   AutoPtr< string > aps( new string("abc") );
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;

   AutoPtr< Date >  apd( new Date(2010, 11, 26) );
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
}
```

Πρόσεξε το εξής: Ενώ έχουμε δύο "**new**" δεν υπάρχουν "**delete**" διότι δεν χρειάζονται την ανακύκλωση των δυναμικών αντικειμένων τη φροντίζουν οι καταστροφείς. Είδαμε λοιπόν έναν τρόπο για να εφοδιάσουμε τη C++ με μηχανισμό «**αποκομιδής απορριμάτων**» (garbage collection). Βέβαια δεν είναι πλήρης, αφού δεν καλύπτει και δυναμικούς πίνακες αλλά είναι ένα πολύ καλό εργαλείο.

25.8.1 *std*: : *auto_ptr*

Αν στο παραπάνω πρόγραμμα με τα παραδείγματα αλλάξεις:

- το "`#include "AutoPtr.h"`" σε "`#include <memory>`" και
- το "`AutoPtr`" σε "`auto_ptr`"

θα πάρεις τα ίδια αποτελέσματα· αυτήν τη φορά όμως με χρήση του περιγράμματος (*std*::)*auto_ptr* που μας δίνει η C++.

Δες την υλοποίηση του `auto_ptr` στο `memory` της C++ που χρησιμοποιείς και πιθανότατα θα δεις ότι είναι σαν αυτήν του `AutoPtr` που δώσαμε πιο πάνω. Τώρα που είδες πώς και γιατί γράφηκε αυτή η παραμετρική κλάση ας δούμε πώς μπορείς να τη χρησιμοποιείς.

1. Μπορείς να δηλώσεις μεταβλητές ως εξής:

```
auto_ptr<int> api1( new int(375) );
auto_ptr<string> aps1( new string("F.C.Barcelona") );
auto_ptr<int> api2;
auto_ptr<string> aps2( aps1 );
```

- Η πρώτη δήλωση έχει ως αποτέλεσμα: η *api1* να έχει κυριότητα μιας δυναμικής μεταβλητής (τύπου `int`) με έχει τιμή 375.
- Η δεύτερη δήλωση έχει αποτέλεσμα: η *aps1* κατέχει μια δυναμική μεταβλητή (τύπου *string*) που έχει τιμή `"F.C.Barcelona"`.
- Το αποτέλεσμα της τρίτης δήλωσης: η *api2* να δεν κατέχει κάποια δυναμική μεταβλητή.
- Η τέταρτη δήλωση έχει ως αποτέλεσμα η *aps2* να πάρει την κυριότητα της δυναμικής μεταβλητής (τύπου `std::string`) που έχει τιμή `"F.C.Barcelona"` και ήταν αρχικώς στην κατοχή της *aps1*. Η *aps1* δεν κατέχει πια κάποια δυναμική μεταβλητή.

Προσοχή όμως!

α) *Τα παρακάτω απαγορεύονται*:

```
int  k( 255 );
auto_ptr<int> api0( &k );
```

Δεν μπορείς να δώσεις στην κατοχή ενός έξυπνου βέλους μια μεταβλητή που δεν είναι δυναμική. Γιατί; Διότι η ώρα του "`delete`" θα έλθει οπωσδήποτε! Δυστυχώς, ο μεταγλωττιστής δεν θα σε αποτρέψει! Όλα θα είναι στη δική σου την ευθύνη!

β) Κατ' αρχήν μπορείς να συνεχίσεις να χειρίζεσαι μια δυναμική μεταβλητή με «παραδοσιακό» βέλος και μετά την κατοχή της από ένα έξυπνο βέλος. Π.χ. οι παρακάτω:

```
int* pi1( new int(511) );
auto_ptr<int> api( pi1 );
cout << *api << endl;
*pi1 = 375;
cout << *api << endl;
int* pi2( api.get() );
*pi2 = 101;
cout << *api << endl;
```

δίνουν:

```
511
375
101
```

Πάντως *κάτι τέτοιο είναι εξαιρετικά επικίνδυνο*!

2. Μπορείς να ελέγξεις την τιμή του κρυμμένου βέλους παίρνοντάς το με τη μέθοδο *get()*. Μπορείς να κάνεις αποπαραπομπή με το "`*`" όπως και με τα απλά βέλη.

Μετά τις παραπάνω δηλώσεις οι παρακάτω εντολές

```
if ( api1.get() != 0 ) cout << *api1 << endl;
                  else cout << " api1 NULL" << endl;
if ( aps1.get() != 0 ) cout << *aps1 << endl;
                  else cout << " aps1 NULL" << endl;
if ( api2.get() != 0 ) cout << *api2 << endl;
                  else cout << " api2 NULL" << endl;
```

```
   if ( aps2.get() != 0 ) cout << *aps2 << endl;
                      else cout << " aps2 NULL" << endl;
```

θα δώσουν:

```
375
 aps1 NULL
 api2 NULL
F.C.Barcelona
```

3. Να ξαναγυρίσουμε στις ιδιαιτερότητες του «δημιουργού αντιγραφής» που, όπως είπαμε, είναι *δημιουργός μεταβίβασης*. Μετά την τέταρτη περίπτωση του σημείου 1, δες και αυτό:

```
   cout << *api1 << endl;
   cout << *(f(api1)) << endl;
   if ( api1.get() != 0 ) cout << *api1 << endl;
                      else cout << " api1 NULL" << endl;
```

όπου:

```
auto_ptr<int> f( auto_ptr<int> x )
{
   *x += 1;
   return x;
} // f
```

Αποτέλεσμα:

```
375
376
 api1 NULL
```

Αρχικώς η *api1* κατέχει βέλος προς δυναμική μεταβλητή που έχει τιμή 375. Όταν καλείται η *f()*, η *api1* περνάει ως παράμετρος τιμής. Αυτό σημαίνει ότι χρησιμοποιείται ο δημιουργός αντιγραφής με αποτέλεσμα η *api1* να χάνει την κατοχή του βέλους προς τη δυναμική μεταβλητή. Την κατοχή έχει πια η τοπική μεταβλητή της συνάρτησης *x*.

Με την εκτέλεση της `return` και η *x* χάνει την κατοχή του βέλους, αφού και πάλι εκτελείται ο δημιουργός αντιγραφής, αλλά αυτό είναι κάτι που δεν μπορούμε να το δούμε ούτε και μας ενδιαφέρει!

4. Μπορείς να αλλάξεις την τιμή μιας τέτοιας μεταβλητής είτε με τη *reset()* είτε με τον (μεταβιβαστικό) τελεστή εκχώρησης "`=`".

Ας θυμίσουμε την επικεφαλίδα της *reset()*:

```
   void reset( K* p = 0 )
```

Δηλαδή, η *reset()* περιμένει όρισμα που είναι σύνηθες βέλος. Έτσι, οι εντολές:

```
   api1.reset( new int(1023) );
   cout << *api1 << endl;
   cout << *aps2 << endl;
   aps2.reset( new string("Kavala F.C.") );
   cout << *aps2 << endl;
```

θα δώσουν:

```
1023
F.C.Barcelona
Kavala F.C.
```

Όπως στη δήλωση, έτσι και στη *reset()*,

α) *δεν επιτρέπεται να περάσεις βέλος προς μια μη δυναμική μεταβλητή,*

β) παρ' όλο που μπορείς να χειρίζεσαι μια δυναμική μεταβλητή και με το απλό βέλος, *από τη στιγμή που το πέρασες σε ένα έξυπνο βέλος μην το ξαναπειράξεις* (το απλό βέλος).

Ο τελεστής "`=`" δουλεύει ως εξής:

```
   auto_ptr& operator=( auto_ptr& rhs ) throw()
   {  reset( rhs.release() );  return *this;  }
```

Πρόσεξε ότι η παράμετρος είναι αναφοράς. Επομένως, μπορείς να μεταβιβάσεις την κατοχή βέλους από άλλη μεταβλητή-έξυπνο-βέλος. Μπορείς λοιπόν να βάλεις:

```
double* pd2( new double(7.5) );
auto_ptr<double> apd2( pd2 );
auto_ptr<double> apd3;
apd3 = apd2;
```

αλλά όχι:

```
apd3 = auto_ptr<double>( new double(7.5) );
```

ούτε:

```
apd3 = f( pd2 );
```

Φυσικά, το σπουδαιότερο που πρέπει να θυμάσαι είναι ότι με *τον τελεστή αυτόν γίνεται μεταβίβαση τιμής και όχι εκχώρηση όπως την ξέρουμε*. Με την:

```
apd3 = apd2;
```

η *apd3* γίνεται κάτοχος του βέλους που είχε η *apd2* αλλά η *apd2* χάνει την κατοχή και καθαρίζεται (*apd2*.*ap* = 0).

5. Με τη *reset()* μπορείς να ανακυκλώσεις τη δυναμική μνήμη που διαχειρίζεται ένα έξυπνο βέλος (περίπου ό,τι κάνεις με τον "`delete`" για τα συνήθη βέλη). Π.χ.:

```
apd3.reset();
```

6. Με τη *release()* μπορείς να ξαναπάρεις τον έλεγχο από το έξυπνο βέλος:

```
double* pd( apd3.release() );
```

25.8.2 Συνελόντι Ειπείν...

Τα έξυπνα βέλη είναι πολύ καλά εργαλεία για να γράφεις πιο σίγουρα προγράμματα με λιγότερα λάθη. Η πιο απλή περίπτωση είναι τα βέλη που παράγονται από το περίγραμμα `auto_ptr`, που βρίσκεται στην πάγια βιβλιοθήκη της C++.

Αν προσπαθήσεις να τα μάθεις και να τα χρησιμοποιείς, τα σημεία που πρέπει να προσέξεις είναι τα εξής:

- Ο δημιουργός αντιγραφής και ο "=" κάνουν μεταβίβαση τιμής και όχι συνήθη αντιγραφή.
- Ο "`delete`" συνήθως δεν χρειάζεται αλλά όπου είναι απαραίτητος αντικαθίσταται από κλήση της "`reset()`".

Μια σοβαρή έλλειψη του *auto_ptr* είναι ότι δεν μπορεί να χειρισθεί δυναμικούς πίνακες.

Πάντως, στο C++11 αποθαρρύνεται η χρήση του *auto_ptr* (που μπορεί να μην υπάρχει σε επόμενες τυποποιήσεις) αφού δίνεται ένα νέο περίγραμμα κλάσης για έξυπνα βέλη, το *unique_ptr*, με πολύ περισσότερες δυνατότητες.

Το C++11 δίνει ακόμη δύο περιγράμματα κλάσεων για έξυπνα βέλη, τα *shared_ptr* και *weak_ptr*, που είναι περίπου αυτά που υπάρχουν στη βιβλιοθήκη Boost με τα ίδια ονόματα.[14]

25.9 Ένα Χρήσιμο Περίγραμμα Κλάσης

Βάζοντας στο πρόγραμμά σου "`#include <limits>`" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*::*numeric_limits*. Η δήλωσή του είναι:

```
template< typename T >
class numeric_limits
{
public:
   static const bool is_specialized = false;
   static T min() throw();
   static T max() throw();
   static const int digits = 0;
   static const int digits10 = 0;
```

[14] `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`

```
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 0;
    static T epsilon() throw();
    static T round_error() throw();
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();
    static const bool is_iec559 = false;
    static const bool is_bounded = false;
    static const bool is_modulo = false;
    static const bool traps = false;
    static const bool tinyness_before = false;
    static const float_round_style round_style = round_toward_zero;
};
```

Μα τι κλάσεις θα βγάλουμε από εδώ; *Τα πάντα είναι "*`static`*"!* Θα το καταλάβεις από τα παραδείγματα που θα δεις στη συνέχεια.

Τα ονόματα των μελών περιγράφουν αρκετά καλά το νόημά τους. Για μερικά από αυτά μπορεί να πρέπει να γυρίσεις στο Κεφ. 17. Στο `limits` υπάρχουν και εξειδικεύσεις για τους βασικούς τύπους `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double` και `long double`.

Ορισμένα μέλη –όπως *min_exponent*, *min_exponent10*, *max_exponent*, *max_exponent10* κλπ και οι συναρτήσεις *epsilon()*, *round_error()* κλπ– έχουν νόημα μόνο για τύπους κινητής υποδιαστολής, δηλαδή τους `float`, `double` και `long double` (και άλλους που τυχόν δίνει κάποια υλοποίηση). Αν κάποιο μέλος δεν έχει νόημα για κάποιον τύπο τότε στην εξειδίκευση αφήνουμε τιμή "`0`" ή "`false`"όπως φαίνεται στην παραπάνω δήλωση.

Με χρήση των κατάλληλων εξειδικεύσεων μπορούμε να ξαναγράψουμε το προγραμματάκι (§2.5):

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
    cout << "INT_MIN = " << INT_MIN << "    INT_MAX = "
         << INT_MAX << endl;
}
```

ως εξής:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
    cout << "INT_MIN = " << numeric_limits<int>::min()
         << "    INT_MAX = " << numeric_limits<int>::max() << endl;
}
```

Δηλαδή:

- Αντί για "`#include <climits>`" βάλαμε "`#include <limits>`".
- Αντί για "`INT_MIN`" βάλαμε "`numeric_limits<int>::min()`".

- Αντί για "`INT_MAX`" βάλαμε "`numeric_limits<int>::max()`".

Ο πρώτος τρόπος γραφής είναι η κληρονομιά από τη C. Ο δεύτερος τρόπος, με το πολύ γράψιμο, είναι «καθαρή C++».

Παρομοίως, το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << DBL_MIN
        << "    DBL_MAX = " << DBL_MAX << endl;
   cout << "DBL_EPSILON = " << DBL_EPSILON << endl;
}
```

θα γίνει:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << numeric_limits<double>::min()
        << "    DBL_MAX = " << numeric_limits<double>::max()
        << endl;
   cout << "DBL_EPSILON = " << numeric_limits<double>::epsilon()
        << endl;
}
```

Από εδώ και πέρα θα προτιμούμε να γράφουμε πιο πολλά και να είμαστε «πιο C++» εκτός από την εξής περίπτωση: όταν ελέγχουμε τα αποτελέσματα συναρτήσεων της C, όπως αυτά που είδαμε στην §24.1.3. Εκεί θα πρέπει να βάζουμε τις προκαθορισμένες σταθερές της C.

Οι κλάσεις που προκύπτουν από τις εξειδικεύσεις της *numeric_limits* είναι παραδείγματα **κλάσεων χαρακτηριστικών** (traits) και η εισαγωγή τους δεν έγινε μόνο για να γράφουμε προγράμματα «πιο C++». Θα δούμε τη χρησιμότητά τους με ένα

Παράδειγμα

Ας πούμε ότι θέλουμε να γράψουμε ένα

```
template < typename T >
class MyClass
{
   void f( ... )
   { . . . }
}
```

που θα μας δίνει στιγμιότυπα για ακέραιους τύπους.

Στη μέθοδο *f* θέλουμε τη μέγιστη τιμή του τύπου *T*. Με αυτά που ξέρουμε μέχρι τώρα θα πρέπει να γράψουμε μια πολλαπλή `if`:

```
if ( T είναι ο int )  {  T mt( INT_MAX ); ... }
else if ( T είναι ο long )  {  T mt( LONG_MAX ); ... }
else . . .
```

Οι συνθήκες "`T είναι ο int`" και οι άλλες παρόμοιες μεταφράζονται σε C++ με τη βοήθεια του `typeinfo`.

Χρησιμοποιώντας το περίγραμμα που είδαμε πιο πάνω τα πράγματα απλουστεύονται πάρα πολύ:

```
T mt( numeric_limits<T>::max() ); ...
```

☜☜☜

Όπως καταλαβαίνεις, οι κλάσεις χαρακτηριστικών είναι πολύτιμο εργαλείο όταν έχουμε να γράψουμε περιγράμματα.

Παρατήρηση:▶

Αξίζει να πούμε κάτι ακόμη, σχετικώς με το πρώτο μας παράδειγμα, το περίγραμμα *BStringT*: Η C++ μας δίνει το περίγραμμα:

```
template< typename charT > struct char_traits
```

και δύο εξειδικεύσεις του:

```
template<> struct char_traits< char >;
template<> struct char_traits< wchar_t >;
```

Με τη χρήση τους θα μπορούσαμε να γράψουμε το *BStringT* πιο εύκολα (και να το κάνουμε πιο αποδοτικό.)◀

25.10 Ανακεφαλαίωση

Με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων με όλες τις λειτουργίες της που να έχει ως παράμετρο τον τύπο του *περιεχομένου*. Η υλοποίηση της δομής για συγκεκριμένο τύπο προκύπτει ως στιγμιότυπο του περιγράμματος.

Οι πιθανές ιδιαιτερότητες που παρουσιάζονται στα στιγμιότυπα για ορισμένους τύπους αντιμετωπίζονται με *εξειδικεύσεις* ή *μερικές εξειδικεύσεις*.

Ενδιαφέρον έχουν οι κληρονομιές περιγραμμάτων:

- Μια κλάση μπορεί να κληρονομεί στιγμιότυπο περιγράμματος κλάσης.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί κάποια κλάση ώστε να χρησιμοποιεί εργαλεία που έχει η κλάση.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί άλλο περίγραμμα κλάσης.

Δύο περιγράμματα κλάσεων που «περιτυλίγουν» βέλη ώστε να εμπλουτίσουν ή/και να αλλάξουν τις ιδιότητές τους είναι: οι *προσεγγιστές* (που ήδη ξέραμε) και τα *έξυπνα βέλη* (μια σύνοψη στην §25.8.2).

Οι *κλάσεις χαρακτηριστικών* όπως η *numeric_limits* είναι πολύ χρήσιμες για να γράφεις περιγράμματα κλάσεων.

Ασκήσεις

25-1 «Μάζεψε» όλα τα περιγράμματα συναρτήσεων για διαχείριση ψηφιοπινάκων σε περίγραμμα κλάσης *Bitmap*.

κεφάλαιο

26

Βιβλιοθήκη Παγίων Περιγραμμάτων – Standard Template Library (STL)

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα κάνουμε μια σύντομη γνωριμία με την STL.
Προσδοκώμενα αποτελέσματα:
Θα μπορείς να χρησιμοποιείς στα προγράμματά σου έτοιμες και καλές λύσεις για πολλά προβλήματα και δεν θα χρειάζεται να γράφεις τα πάντα από την αρχή. Όπως θα κατάλάβεις, θα είναι σαν να γράφεις τα προγράμματά σου σε γλώσσα προγραμματισμού πολύ υψηλού επιπέδου.
Έννοιες κλειδιά:

- *περιέχουσα κλάση, περιέχον*
- *προσεγγιστής*
- *αλγόριθμοι*
- *συναρτησιακό αντικείμενο*
- *ακολουθίες*
- *συνειρμικά περιέχοντα*
- *επιλογή περιέχοντος*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Η **Βιβλιοθήκη Παγίων Περιγραμμάτων** (Standard Template Library – STL) περιλαμβάνει τέσσερα είδη συνιστωσών:

- *περιέχουσες κλάσεις,*
- *προσεγγιστές,*
- *αλγόριθμους,*
- *συναρτησιακά αντικείμενα.*

Για την υλοποίησή τους χρησιμοποιούνται

- **παραχωρητές μνήμης** (allocators) και
- κλάσεις χαρακτηριστικών.

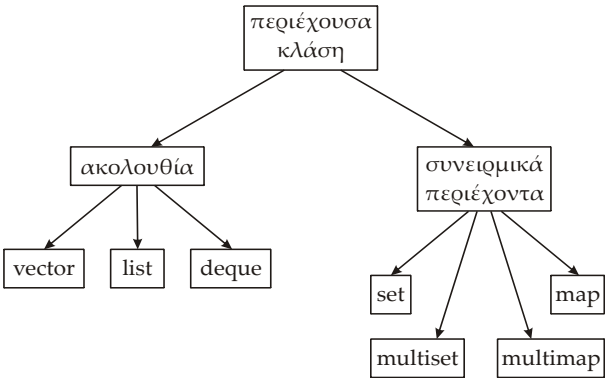
Στα παραδείγματα κλάσεων με δυναμικούς πίνακες βάλαμε έναν μηχανισμό που ήταν παντού ο ίδιος (με ή χωρίς τη *renew()*). Ο πειρασμός να γράψουμε ένα περίγραμμα κλάσης για δυναμικό μονοδιάστατο πίνακα με αυτορυθμιζόμενο μέγεθος είναι μεγάλος! Δεν χρειάζεται: η STL έχει τέτοιο εργαλείο.

Ας δούμε ένα απλό πρόβλημα και ας το λύσουμε με αυτά που ξέρουμε:

Θέλουμε ένα πρόγραμμα που θα διαβάζει από το πληκτρολόγιο άγνωστο πλήθος φυσικών, θα τους ταξινομεί και θα τους δείχνει στην οθόνη.

Χρησιμοποιώντας τη *renew()* από το `MyTmpltLib.h` και την *qsort()* που είδαμε στην §16.10 έχουμε:

```
#include <iostream>
#include <cstdlib>

#include "MyTmpltLib.h"

using namespace std;

int compare( const void* a, const void* b )
{
   return ( *reinterpret_cast<const unsigned int*>(a) -
            *reinterpret_cast<const unsigned int*>(b) );
} // compare

int main()
{
   const unsigned int inc( 5 );
   size_type size( inc );
   unsigned int* array( new unsigned int[inc] );

   int x;
   size_type count( 0 );
   cin >> x;
   while ( x >= 0 )
   {
      if ( count >= size )
      {
         renew( array, count, size+inc );
         size += inc;
      }
      array[count] = x;
      ++count;
      cin >> x;
   } // while
   qsort( array, count, sizeof(int), compare );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```



Σχ. 26-1 Οι περιέχουσες κλάσεις της STL.

Στη συνέχεια θα δούμε διαφορετικές λύσεις που μπορούμε να γράψουμε με τα εργαλεία της STL.

26.1 Περιέχοντα, Προσεγγιστές και Αλγόριθμοι

Ένας άλλος τρόπος να λύσουμε το πρόβλημα που δώσαμε στην εισαγωγή είναι ο εξής:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   vector< unsigned int > array;

   int x;
   cin >> x;
   while ( x >= 0 )
   {
      array.push_back( x );
      cin >> x;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```

Όπως βλέπεις, αυτό είναι αρκετά απλούστερο από το αρχικό πρόγραμμα.

Το "`vector`" είναι *περιέχουσα κλάση* της STL και εδώ χρησιμοποιούμε ένα στιγμιότυπο της για περιεχόμενο "`unsigned int`".

Ο `array` είναι ένας δυναμικός πίνακας. Πρόσεξε ότι το μόνο σημείο που αναφερόμαστε στο μέγεθος του πίνακα είναι το "`array.size()`" με το οποίο ζητάμε από τον `array` να μας πει το μέγεθός του. Η "`array.push_back(x)`" αντιγράφει την τιμή της *x* στο τέλος του πίνακα. Αν το μέγεθος του πίνακα δεν είναι αρκετό φροντίζει να τον μεγαλώσει.

Το "`sort`" είναι ένα περίγραμμα συνάρτησης που περιμένει δύο προσεγγιστές που να ορίζουν μια περιοχή μέσα σε κάποιο περιέχον και ταξινομεί τις τιμές μέσα στην περιοχή. Η ταξινόμηση γίνεται με βάση τον τελεστή "`<`" που πρέπει να είναι ορισμένος για τον περιεχόμενο τύπο.

Τα "`array.begin()`" και "`array.end()`" είναι κλήσεις μεθόδων του `array` που μας δίνουν (σταθερούς) *προσεγγιστές* προς την αρχή και το τέλος του. Θυμίσου τις μεθόδους με τα ίδια ονόματα που είδαμε στην *SListT*.

26.1.1 Περιέχουσες Κλάσεις

Όπως λέγαμε και στο προηγούμενο κεφάλαιο (§25.7) «*Περιέχον (container) ή περιέχουσα κλάση (container class) είναι περίγραμμα για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων.*» Στη συνέχεια θα χρησιμοποιούμε τον όρο «περιέχον» και για στιγμιότυπα κάποιας περιέχουσας κλάσης.

Μπορούμε να δούμε τα περιέχοντα της STL χωρισμένα σε δύο κατηγορίες (Σχ. 26-1):

- **Ακολουθίες** (sequences) στις οποίες η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Στις ακολουθίες ανήκουν τα
  - *vector* (διάνυσμα). Με τη δήλωση "`vector<K> vt;`" ο *vt* δηλώνεται ως δυναμικός μονοδιάστατος πίνακας (διάνυσμα) με στοιχεία τύπου *K*.
  - *list*. Με τη δήλωση "`list<K> lt;`" η *lt* δηλώνεται ως λίστα με διπλή σύνδεση με στοιχεία τύπου *K*.
  - *deque* (*d*ouble *e*nded *que*ue, ουρά δύο άκρων). Μπορεί να γίνεται εισαγωγή και εξαγωγή από τα δύο άκρα.

  και οι προσαρμογείς
  - *priority_queue* (ουρά προτεραιότητας). Γενικώς, τα στοιχεία εισάγονται μαζί με βαθμό προτεραιότητας και εξάγεται πρώτο το στοιχείο με τη μεγαλύτερη προτεραιότητα. Στην STL πρέπει να οριστεί η διάταξη "<" (για την περιεχόμενη κλάση) και πρώτο εξάγεται το «μεγαλύτερο».
  - *queue* (ουρά) FIFO. Επιτρέπει εισαγωγή μόνον στο τέλος της ουράς και εξαγωγή μόνον από την αρχή. Μπορείς να τη δεις σαν ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *φθίνουσα* προτεραιότητα.
  - *stack* (στοίβα) LIFO. Επιτρέπει εισαγωγή και εξαγωγή μόνον από ένα άκρο. Μπορείς να τη θεωρήσεις ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *αύξουσα* προτεραιότητα.
- **Συνειρμικά Περιέχοντα** (associative containers) στα οποία η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση κάποιο κλειδί. Αυτά είναι:
  - *set* (σύνολο).
  - *multiset* (πολυσύνολο).
  - *map* (απεικόνιση), σύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).
  - *multimap* (πολυαπεικόνιση), πολυσύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).

  Όλα τα παραπάνω περιέχοντα έχουν:
- Ερήμην δημιουργό.
- Δημιουργό αντιγραφής.
- Καταστροφέα.
- *Αντιγραφικό* τελεστή εκχώρησης.
- Μέθοδο *size()* που επιστρέφει το πλήθος στοιχείων της συλλογής.
- Μέθοδο *max_size()* που επιστρέφει το μέγιστο πλήθος στοιχείων που μπορεί να αποθηκευτεί στη συλλογή.
- Μέθοδο *empty()* που επιστρέφει τιμή `true` αν η περιεχόμενη συλλογή δεν έχει στοιχεία.
- Μέθοδο *swap()* για ανταλλαγή τιμών δύο περιεχόντων (του ίδιου τύπου).
- Μέθοδο *get_allocator()* που επιστρέφει αντίγραφο του παραχωρητή μνήμης του περιέχοντος.

Έχουν ακόμη τέσσερις μεθόδους που τις βλέπουμε στη συνέχεια.

Με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι

- "`==`", "`!=`" (το $a \neq b$ υπολογίζεται ως !(a `==` b)),
- "`<`", "`>=`" ((a `>=` b) ≡ !(a `<` b)),
- "`>`", "`<=`" ((a `>` b) ≡ (b `<` a)).

Η σύγκριση "`<`" γίνεται *λεξικογραφικώς* (§22.7.3). Για να γίνει αυτό θα πρέπει να έχουμε (εκτός από τον "`==`") και τον "`<`" για την περιεχόμενη κλάση.

Κάθε μια από τις περιέχουσες κλάσεις έχουν τις δικές τους *κλάσεις προσεγγιστών*. Πρώτη η *iterator* που είναι σαν αυτήν που είδαμε στην *SListT*. Και αυτή έχει επιφορτωμένους τους "`++`", "`*`", "`->`", "`==`", "`!=`". Ακόμη, έχει επιφορτωμένον και τον "`--`": αυτοί οι προσεγγιστές είναι **αμφίδρομοι** (bidirectional) ενώ της *SListT* ήταν *μονόδρομοι*.

Η δεύτερη κλάση είναι η *const_iterator*. Αν δηλώσεις

```
C<T>::const_iterator it;
```

δεν έχεις δικαίωμα να αλλάξεις την τιμή του "`*it`".

Δύο μέθοδοι μας δίνουν την αρχή και το τέλος των περιεχομένων της περιέχουσας κλάσης:

- Μέθοδος *begin()*: που επιστρέφει προσεγγιστή προς το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
iterator begin();
const_iterator begin() const;
```

- Μέθοδος *end()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) μετά το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
iterator end();
const_iterator end() const;
```

(Η δεύτερη μορφή –*const_iterator*– για αντικείμενα-συλλογές "`const`".) Αν κάποιο περιέχον a είναι *κενό* –η *a.empty()* επιστρέφει `true`– τότε έχουμε *a.begin()* `==` *a.end()*.

Φυσικά, θα πεις: το «*πρώτο στοιχείο*» και η «*πρώτη θέση(!) μετά το τελευταίο στοιχείο*» μπορεί να έχουν νόημα για πίνακα, μπορεί να έχουν νόημα και για λίστα (όπως είδαμε στα προηγούμενα κεφάλαια, π.χ. §25.7.1)· τι νόημα μπορεί να έχουν για το σύνολο; Θα κατάλάβεις στη συνέχεια…

Για οποιαδήποτε περιέχουσα κλάση C της STL, με τη:

```
for ( C<T>::iterator it( co.begin() ); it != co.end(); ++it )
{ /* . . . */ }
```

μπορείς να περάσεις από όλα τα περιεχόμενα αντικείμενα τύπου *T* του αντικειμένου-συλλογή *co*. Αν δεν θέλεις να αλλάζεις τις τιμές των περιεχομένων μπορείς να χρησιμοποιήσεις προσεγγιστή "`C<T>::const_iterator`" ώστε να κάνεις το πρόγραμμά σου κάπως ταχύτερο.

Ας πούμε ότι η:

```
list< int > il;
```

έχει περιεχόμενο τους 0, 1, 2, 3, 4. Οι

```
for ( list<int>::iterator it( il.begin() );
      it != il.end(); ++it )
   cout << *it << "  ";
cout << endl;
```

θα δώσουν:

```
0  1  2  3  4
```

Δύο άλλες κλάσεις προσεγγιστών είναι οι *reverse_iterator* και *const_reverse_iterator*. Με τέτοιους προσεγγιστές διασχίζεις τη συλλογή από το τέλος προς την αρχή. Δύο μέθοδοι σου δίνουν αρχή και τέλος για τέτοια διάσχιση:

- Μέθοδος *rbegin()*: που επιστρέφει προσεγγιστή προς το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

- Μέθοδος *rend()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) πριν το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Οι εντολές

```
for ( list<int>::reverse_iterator rit( il.rbegin() );
      rit != il.rend(); ++rit )
   cout << *rit << "  ";
cout << endl;
```

για τη λίστα που είδαμε παραπάνω θα δώσουν:

```
4  3  2  1  0
```

Πρόσεξε ότι διασχίζουμε τη λίστα με την "`++rit`".

Αυτά προς το παρόν για τις περιέχουσες κλάσεις. Στη συνέχεια θα δούμε και άλλα που ισχύουν ειδικότερα για συγκεκριμένες κατηγορίες τους.

Υπάρχουν όμως και ορισμένες απαιτήσεις από τον τύπο της *περιεχόμενης* κλάσης. Στην §25.7.2 συνοψίσαμε μερικές από αυτές που επισημάναμε γράφοντας την περιέχουσα κλάση *SListT*:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Ας πούμε,

- ***Αν χρειάζονται αναζητήσεις τιμών ο τύπος περιεχομένου K θα πρέπει να έχει τη δυνατοτητα για συγκρίσεις.***

26.1.2 Περί Προσεγγιστών...

Πριν προχωρήσουμε στην παρουσίαση των περιεχουσών κλάσεων της STL θα πούμε λίγα ακόμη για τους προσεγγιστές.

Η έννοια του προσεγγιστή είναι γενίκευση της έννοιας του βέλους. Με ποιον στόχο; Τη χρήση πάγιων τεχνικών και αλγορίθμων των μονοδιάστατων πινάκων σε άλλες δομές δεδομένων.

Αρχικώς, στην §22.9, γνωρίσαμε τον *πρόσθιο προσεγγιστή*. Σε αυτόν επιφορτώνουμε τους τελεστές "`*`", "`->`", "`++`", "`==`", "`!=`".

Ο *αμφίδρομος προσεγγιστής*, που είδαμε πιο πάνω, έχει όλα τα χαρακτηριστικά του *πρόσθιου προσεγγιστή* αλλά σε αυτόν επιφορτώνουμε και τον "`--`".

Στη συνέχεια θα δούμε τους **προσεγγιστές τυχαίας πρόσβασης** (random access iterators) που δεν έχουν να ζηλέψουν κάτι από τα βέλη: έχουν όλα τα χαρακτηριστικά του αμφίδρομου προσεγγιστή και επιπλέον αριθμητική σαν αυτήν των βελών (§12.3): "*προσεγγιστής θ ακέραιος*" όπου *θ* κάποιος από τους "`+`", "`-`", "`+=`", "`-=`" ή "*προσεγγιστής* `-` *προσεγγιστής*" (που μας δίνει την «απόσταση» των στοιχείων που δείχνουν οι δύο προσεγγιστές).

Ας δούμε τώρα μια τρίτη μορφή του παραδείγματός μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <iterator>

using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   for ( int k(0); k < count; ++k )
      *coutIt = array[k];
   cout << endl;
}
```

και ένα παράδειγμα εκτέλεσης:

```
19  0  17  3  11  5  7 e
0  3  5  7  11  17  19
```

Βάζοντας στο πρόγραμμα την "`#include <iterator>`" μπορούμε να χρησιμοποιήσουμε το *ostream_iterator* που είναι ένα περίγραμμα που μπορεί να μας δώσει στιγμιότυπα για οποιονδήποτε τύπο *T* για τον οποίον υπάρχει επιφορτωμένος ο "`<<`" προς μορφοποιημένο αρχείο. Ο *coutIt* είναι ένας προσεγγιστής *ostream*. Με τη δήλωση του *coutIt* λέμε ότι θα είναι ένας προσεγγιστής που θα γράφει τιμές τύπου `unsigned int` στο *cout* και μετά από κάθε τιμή θα «γράφει» δύο διαστήματα (`"  "`). Πώς γράφει μια τιμή *v*; Η εντολή: "`*coutIt = v;`" Ισοδυναμεί με "`cout << v << "  ";`". Έτσι, η **for** του προγράμματος βγάζει το αποτέλεσμα που είδες παραπάνω.

Ο *ostream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εξόδου** (output iterator). Ένας προσεγγιστής εξόδου έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και κατάστροφέα. Ακόμη μπορεί να «αυξηθεί» (π.χ. με τον "`++`").
- Με την αποπαραπομπή του παίρνουμε *τιμή-l* (§11.3). Στην πραγματικότητα η αποπαραπομπή ενός προσεγγιστή εξόδου εμφανίζεται πάντοτε στο αριστερό μέρος μιας εκχώρησης όπως φαίνεται και στο παράδειγμά μας ("`*coutIt = array[k];`").

Όπως καταλαβαίνεις και οι τρεις κατηγορίες προσεγγιστών που είδαμε –πρόσθιοι, αμφίδρομοι, τυχαίας πρόσβασης– έχουν τα χαρακτηριστικά του προσεγγιστή εξόδου.

Όπως πιθανότατα μαντεύεις, υπάρχουν και *προσεγγιστές istream*. Έτσι μπορούμε να διαβάσουμε από κάποιο *istream* ως εξής:

```
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
```

Ο *cinIt* είναι ένας προσεγγιστής κλάσης `istream_iterator<unsigned int>`: Διαβάζει τιμές τύπου `unsigned int` από ένα ρεύμα *istream*, στην περίπτωσή μας από το *cin*. Δηλώνοντας έναν *istream_iterator* χωρίς ορίσματα –όπως τον *cinEoStream*– παίρνεις έναν προσεγγιστή που όμως «δείχνει μετά το τέλος του ρεύματος τιμών `unsigned int`» και χρησιμοποιείται για τον τερματισμό της ανάγνωσης.

Η τιμή που διαβάζουμε δίνεται με την αποπαραπομπή του προσεγγιστή (`*cinIt`). Ο προσεγγιστής διαβάζει με τον "`>>`". Προχωρούμε στην επόμενη τιμή με τον τελεστή "`++`".

Αν φτάσουμε σε τέλος αρχείου ή σε τέλος ρεύματος (δηλαδή –στην περίπτωσή μας– τιμή που δεν είναι `unsigned int`) η τιμή του *cinIt* θα γίνει ίση με αυτήν του *cinEoStream*.

Ο *istream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εισόδου** (input iterator) που έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και κατάστροφέα. Αν δεν έχει φτάσει στο τέλος ρεύματος, μπορεί να «αυξηθεί» (π.χ. με τον "`++`"). Ακόμη, μπορεί να συγκριθεί με τον "`==`" και τον "`!=`".
- Με την αποπαραπομπή του παίρνουμε *τιμή-r* (μπορεί να εμφανιστεί μόνον στο δεξιό μερος της εκχώρησης και όχι στο αριστερό).

Τα χαρακτηριστικά του προσεγγιστή εισόδου τα βρίσκουμε και στους προσεγγιστές που είδαμε: εξόδου, πρόσθιους, αμφίδρομους και τυχαίας πρόσβασης.

Βάζοντας στο πρόγραμμά σου "`#include <iterator>`" μπορείς να χρησιμοποιήσεις το περίγραμμα συνάρτησης

```
template < typename InputIterator, typename Distance >
void advance( InputIterator& it, Distance n );
```

που δίνει τη λειτουργικότητα του "`+=`" σε οποιονδήποτε προσεγγιστή εισόδου. Ο *Distance* τι τύπος είναι; Κατ' αρχήν είναι ακέραιος τύπος. Αν ο *InputIterator* είναι τύπος προσεγγιστή αμφίδρομου ή τυχαίας πρόσβασης ο *Distance* μπορεί να είναι και ο `int`. Αλλιώς πρέπει να είναι `unsigned int`.

Η "`advance(it, n);`"

- αν ο *it* είναι τυχαίας πρόσβασης, θα εκτελεσθεί ως "`it += n;`"
- αλλιώς θα εκτελεσθεί με επαναληπτική (*n* φορές) δράση του "`++it;`" ή "`--it;`".

26.1.3 Αλγόριθμοι

Έχουμε ήδη μάθει ότι βάζοντας στο πρόγραμμά μας "`#include <algorithm>`" μπορούμε να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων:

- (*std*::)*max()*, (*std*::)*min()*,
- (*std*::)*swap()*,
- (*std*::)*find()*,
- (*std*::)*sort()*.

Στη συνέχεια θα δούμε μερικά ακόμη, αλλά δεν θα παραθέσουμε και τα πάντα. Ψάξε τα εγχειρίδια της C++ που χρησιμοποιείς για να δεις τι υπάρχει.

Εδώ θα δούμε άλλη μια μορφή του παραδείγματός μας χρησιμοποιώντας το περίγραμμα:

```
template< class InputIterator, class OutputIterator >
OutputIterator copy( InputIterator first, InputIterator last,
                     OutputIterator result );
```

που αντιγράφει τα στοιχεία

από την περιοχή `[` *first*, *last* `)` στην περιοχή `[` *result*, *result* + (*last* - *first*) `)`.

Κάνει δηλαδή τις εκχωρήσεις "`*result = *first`", "`*(result+1) = *(first +1)` ", κ.ο.κ. με αυτήν τη σειρά. Επιστρέφει ως τιμή το *result* + (*last* - *first*)· δηλαδή έναν προσεγγιστή προς μια «θέση» μετά το τελευταίο στοιχείο που ήλθε από την αντιγραφή.

Δες πώς γράφεται το πρόγραμμά μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
```

```
using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;
   back_insert_iterator< vector<unsigned int> > dest( array );
   copy( cinIt, cinEoStream, dest );
   sort( array.begin(), array.end() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   copy( array.begin(), array.end(), coutIt );
   cout << endl;
}
```

Πρόσεξε πώς γίνεται η έξοδος των στοιχείων: με αντιγραφή όλων των στοιχείων του πίνακα –αυτά περιλαμβάνονται στην περιοχή [*array*.*begin*(), *array*.*end*())– στο ρεύμα *cout*. Αυτό είναι το εύκολο.

Η δυσκολία υπάρχει στην ανάγνωση. Θα μπορούσαμε να διαβάσουμε τα στοιχεία με την *copy()*; Να γράψουμε δηλαδή κάτι σαν:

```
   copy( cinIt, cinEoStream, array.begin() );
```

Όχι! Εδώ έχουμε πρόβλημα: Ο *array* δεν έχει την απαιτούμενη μνήμη και –γενικώς– δεν ξέρουμε πόσα είναι τα στοιχεία για να την κρατήσουμε πριν ζητήσουμε την αντιγραφή.

Η λύση είναι η εξής:

```
   back_insert_iterator< vector<unsigned int> > dest( array );
   copy( cinIt, cinEoStream, dest );
```

Ο *back_insert_iterator* είναι ένας *προσαρμογέας*: από τη μια είναι ένας προσεγγιστής, όπως τον θέλει η *copy()*, από την άλλη φροντίζει να παίρνει την απαραίτητη μνήμη ώστε να γίνεται η εισαγωγή μιας τιμής στον *array* χωρίς πρόβλημα.

Πριν προχωρήσουμε, αξίζει να «θαυμάσουμε» αυτή τη μορφή του προγράμματος που μοιάζει με ψευδοκώδικα! copy από την είσοδο – sort – copy στην έξοδο! Ή, αλλιώς: έχουμε προγραμματισμό σε ψηλότερο επίπεδο!

Παρατηρήσεις:▶

1. Η STL έχει περιγράμματα συναρτήσεων που δρουν σε *περιοχές* ενός περιέχοντος (ή σε *ολόκληρο* περιέχον). Δηλαδή, για ένα πρόβλημα σαν αυτό που είδαμε εδώ, η αντιπροσωπευτική λύση είναι η τελευταία και όχι οι προηγούμενες με τις **while** και **for**.
2. Όπως χρησιμοποιείς τα περιέχοντα της STL μπορείς να χρησιμοποιείς και τα περιέχοντα της C++ –δηλαδή τους πίνακες– και περιέχοντα δικά σου. Δες μια άλλη μορφή του παραδείγματος που είδαμε πιο πάνω:

```
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
   int  array[100];
   int* pEnd;

   pEnd = copy( istream_iterator<int>(cin), istream_iterator<int>(),
                array );
   sort( array, pEnd );
   copy( array, pEnd, ostream_iterator<int>(cout, "\n") );
}
```

Εδώ, περιέχον είναι ένας συνήθης πίνακας. Ως προσεγγιστές χρησιμοποιούμε δύο συμβατικά βέλη, τα `array` και `pEnd`. Κατά τα άλλα χρησιμοποιούμε τα περιγράμματα *copy()* και *sort()*.◀

Πριν προχωρήσουμε είναι καλό να ρίξουμε μια ματιά στις προδιαγραφές των περιγραμμάτων που ήδη είδαμε και χρησιμοποιούμε.

Ξεκινούμε από τα *min()* και *max()* που δίνονται με δύο μορφές:

```
template< class T >
const T& min( const T& a, const T& b );
template< class T, class Compare >
const T& min( const T& a, const T& b, Compare comp );
```

και

```
template< class T >
const T& max( const T& a, const T& b );
template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp );
```

Για τις απλές περιπτώσεις ο τύπος *T* θα πρέπει να έχει

- τον τελεστή "`<`" και
- δημιουργό αντιγραφής.

Έτσι, για παράδειγμα η *min()* μπορεί να είναι:

```
template < class T >
inline const T& min( const T& a, const T& b )
{  return b < a ? b : a;  }
```

Οι μορφές με τις τρεις παραμέτρους μπορεί να χρησιμοποιηθούν όταν ο *T* δεν έχει τον "`<`" ή θέλουμε να κάνουμε σύγκριση με άλλον τρόπο· αρκεί να δοθεί η *comp()* που μας λέει ποια από δύο τιμές *T* είναι η «μικρότερη» ή «προηγείται». Στην περίπτωση αυτήν η *min()* μπορεί να είναι:

```
template < class T, class Compare >
inline const T& min( const T& a, const T& b, Compare comp )
{  return comp(b, a) ? b : a;  }
```

Επομένως η *Compare* είναι ένα συναρτησοειδές, δηλαδή κλάση στην οποίαν έχουμε επιφορτώσει τον τελεστή "`()`" ώστε να δέχεται δύο παραμέτρους τύπου *T* και να επιστρέφει `true` αν η πρώτη είναι η «μικρότερη».

Αν έχεις να υπολογίσεις τον ελάχιστο από δύο ακέραιους χρησιμοποιείς την απλή μορφή: "`min( i1, i2 )`".

Αν όμως έχεις να συγκρίνεις τα `"abc"` και `"aggr"` θα χρειαστείς τη δεύτερη.[1] Για να τη χρησιμοποιήσεις έχεις δύο επιλογές:

- Να γράψεις κλάση που να ταιριάζει με την "`class Compare`":

```
struct CStrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {  return strcmp(cs1, cs2) < 0;  }
}; // CStrLT
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLT)`" όπου το *cstrLT* είναι (συναρτησιακό) αντικείμενο κλάσης *CStrLT*.

- Να γράψεις συνάρτηση (κατηγόρημα):

```
bool cstrLTf( const char cs1[], const char cs2[] )
{  return strcmp(cs1, cs2) < 0;  }
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLTf)`".

[1] Βέβαια μπορείς να βολευτείς με την απλή μορφή αν γράψεις "`min( string("abc"), string("aggr") )`".

Παρόμοια ισχύουν και για τη *max()* αλλά προσοχή: μη θεωρήσεις ότι εδώ θα πρέπει να ασχοληθείς με τον "`>`"· και πάλι ο "`<`" θα κάνει τη δουλειά μας. Για παράδειγμα η "`max("abc", "aggr", cstrLTf)`" θα μας δώσει `"aggr"`.

Ας έλθουμε τώρα στο περίγραμμα:

```
template< class InputIterator, class T >
InputIterator find( InputIterator first, InputIterator last,
                    const T& value );
```

Στον τύπο *T* πρέπει να είναι δυνατές οι συγκρίσεις "`==`" και "`!=`".

Πάντως υπάρχει και το περίγραμμα *find_if()* που είναι γενίκευση του *find()*:

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Μας επιστρέφει προσεγγιστή *it* προς την πρώτη τιμή για την οποίαν το `pred(*it)` θα επιστρέψει `true`. Αν δεν βρει τέτοια τιμή επιστρέφει *last*.

Για το *pred* έχουμε τις ίδιες επιλογές που είχαμε και για το *comp* προηγουμένως αλλά το *pred* είναι κατηγόρημα με μια παράμετρο.

Και η *sort()* έχει δύο μορφές:

```
template< class RandomAccessIterator >
void sort( RandomAccessIterator first, RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void sort( RandomAccessIterator first, RandomAccessIterator last,
           Compare comp );
```

Όπως λέει και το όνομα της κλάσης-παραμέτρου πρέπει να έχουμε προσεγγιστές τυχαίας πρόσβασης. Στην πρώτη περίπτωση η ταξινόμηση γίνεται με βάση τον τελεστή "`<`" ενώ στη δεύτερη με βάση το κατηγόρημα (δύο παραμετρων) *comp()* για το οποίο ισχύουν αυτά που είπαμε πιο πάνω.

Τώρα θα πεις, αφού μας δίνει εργαλείο για ταξινόμηση δεν θα μας δώσει και ένα εργαλείο για δυαδική αναζήτηση; Μας δίνει και όχι μόνον ένα.

```
template< class ForwardIterator, class T >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value );
template< class ForwardIterator, class T, class Compare >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value, Compare comp );
```

Εδώ έχεις μια απογοήτευση: αν βρεθεί στο `[`*first*, *last*`)` κάποιος *it* τέτοιος που να έχουμε `*`*it* `==` *value* η *binary_search()* θα σου επιστρέψει `true`. Αυτό συνήθως δεν είναι αρκετό.

Ας δούμε μιαν άλλη συνάρτηση (δηλαδή *δύο άλλες*):

```
template< class ForwardIter, class T >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value );
template< class ForwardIter, class T, class Compare >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value, Compare comp );
```

Ο προσεγγιστής που επιστρέφεται –ας τον πούμε *rv*– δείχνει την πρώτη θέση στην οποία μπορεί να εισαχθεί η *value* χωρίς να χαλάσει η ταξινόμηση· δείχνει το πρώτο στοιχείο που δεν είναι μικρότερο από *value*. Δηλαδή για κάθε προσεγγιστή *it* στο `[`*first*, *rv*`)` έχουμε `*`*it* `<` *value*. Αν *rv* `!=` *last* για κάθε *it* στο `[`*rv*, *last*`)` έχουμε `*`*it* `>=` *value* (ακριβέστερα: `!(*`*it* `<` *value*`)`.)

Αν χρησιμοποιήσεις τη δεύτερη μορφή τότε στο `[`*first*, *rv*`)` έχουμε `comp(*it, value)` (`==` `true`) ενώ στο `[`*rv*, *last*`)` έχουμε `!comp(*it, value)`.

Αν η αναζητούμενη τιμή υπάρχει στο `[`*first*, *last*`)` *τότε θα τη δείχνει ο rv δηλαδή θα έχουμε:* `*`*rv* `==` *value* (ακριβέστερα: `!(*`*rv* `<` *value*`)` `&&` `!(`*value* `<` `*`*rv*`)`.)

Παρατηρήσεις:▶

1. Και οι τέσσερις συναρτήσεις έχουν πολυπλοκότητα ανάλογη του *log*(*last*–*first*).

2. Η *comp* πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.
3. Υπάρχει και ένα ζεύγος περιγραμμάτων συναρτήσεων *upper_bound()* που επιστρέφουν προσεγγιστή προς τη μικρότερη τιμή που είναι μεγαλύτερη από αυτήν που ψάχνουμε.◀

26.1.3.1 Τρεις Συναρτήσεις για «τα Πάντα»

Πριν προχωρήσουμε στα συναρτησιακά αντικείμενα θα πούμε λίγα λόγια για τρία περιγράμματα συναρτήσεων που *δρουν σε όλα τα αντικείμενα μιας περιοχής*: πρόκειται για τις *for_each(), transform()* και *random_shuffle()*.

Η συνάρτηση

```
template< class InputIterator, class Function >
Function for_each( InputIterator first, InputIterator last, Function f );
```

σαρώνει την περιοχή **[***first, last***)** με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή του υπολογίζει το "**f(*it)**". Αν η *f()* δεν είναι **void** το αποτέλεσμα που επιστρέφει αγνοείται.

Τα αντικείμενα κλάσης *Function* έχουν επιφορτωμένο τον "**()**" (συναρτησιακά αντικείμενα) έτσι ώστε να δέχονται μοναδική παράμετρο τα αντικείμενα που προκύπτουν από την αποπαραπομπή προσεγγιστών κλάσης *InputIterator*. Το όνομα της κλάσης-παραμέτρου μας δείχνει ότι τα χαρακτηριστικά των προσεγγιστών: είναι *προσεγγιστές εισόδου*. Αν πάρουμε υπόψη μας ότι η αποπαραπομπή τους είναι *τιμή-r* καταλαβαίνουμε ότι *η "**f(*it)**" δεν μπορεί να αλλάζει την τιμή του αντικειμένου που δείχνει ο it*.

Η *for_each()* επιστρέφει το συναρτησιακό αντικείμενο *f* όπως αυτό διαμορφώθηκε μετά την εφαρμογή σε όλα τα αντικείμενα της περιοχής.

Το παρακάτω πρόγραμμα

```
#include <iostream>
#include <algorithm>

using namespace std;

template< typename T >
struct Print
{
   void operator()( T x )
   {  cout << x << ' ';  }
}; // Print

int main()
{
   int        intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   Print<int> pr;

   for_each( intArr, intArr + 7, pr );
   cout << endl;
}
```

θα δώσει:

```
34 72 -23 8 -11 31 41
```

Θα μπορούσαμε να πάρουμε τα ίδια αποτελέσματα με τις:

```
   for_each( intArr, intArr+7, printF<int> );
   cout << endl;
```

όπου

```
template< typename T >
void printF( T x )  {  cout << x << ' ';  }
```

Για να δεις πώς μπορούμε να χρησιμοποιήσουμε αυτό που επιστρέφει η *for_each()* αλλάζουμε λίγο την *Print* (SGI 1999):

```
template< typename T >
struct Print
```

```
{
   Print() :  count(0) {};
   void operator()( T x )
   {  cout << x << ' ';
      ++count;  }
   unsigned int count;
}; // Print
```

και τον τρόπο που την καλούμε:

```
   pr = for_each( intArr, intArr+7, pr );
   cout << endl << pr.count << " numbers printed" << endl;
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
7 numbers printed
```

Αν θέλεις να αλλάξεις «τα πάντα» θα χρησιμοποιήσεις ένα από τα δύο περιγράμματα:

```
template< class InputIter, class OutputIter,
          class UnaryOperation >
OutputIterator transform( InputIter first, InputIter last,
                          OutputIter result, UnaryOperation op );
template< class InputIter1, class InputIter2,
          class OutputIter, class BinaryOperation >
OutputIterator transform( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, OutputIter result,
                          BinaryOperation binary_op );
```

Στα στιγμιότυπα της πρώτης μορφής σαρώνεται η περιοχή **[***first, last***)** με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή υπολογίζεται το **op(*it)** που φυλάγεται σε διαδοχικές θέσεις ξεκινώντας από αυτήν που δείχνει ο *result*. Ο *result* μπορεί να δείχνει την ίδια θέση που δείχνει και ο *first*. Επιστρέφει προσεγγιστή προς την πρώτη θέση μετά την τελευταία όπου αποθηκεύτηκε αποτέλεσμα.

Ένα απλό παράδειγμα χρήσης της πρώτης μορφής είναι το παρακάτω:

```
#include <iostream>
#include <algorithm>

using namespace std;

template< class T >
struct Sqr
{
   T operator()( T x )  {  return x*x;  }
}; // Sqr

int main()
{
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   int      intArr1[7];
   Sqr<int> sqr;

   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   int* res1( transform(intArr, intArr+7, intArr1, sqr) );
   copy( intArr1, intArr1+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res1-intArr1) << endl;

   int* res( transform(intArr, intArr+7, intArr, sqr) );
   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res-intArr) << endl;
}
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
1156 5184 529 64 121 961 1681
7
1156 5184 529 64 121 961 1681
```

7

Το συναρτησιακό αντικείμενο *Sqr*[2] υπολογίζει και επιστρέφει το τετράγωνο της μοναδικής παραμέτρου του.

Η πρώτη κλήση της *transform()* έχει ως αποτέλεσμα να υπολογιστούν τα τετράγωνα των στοιχείων του `intArr` και να φυλαχτούν στα αντίστοιχα στοιχεία του `intArr1`. Ο προσεγγιστής (στην περίπτωσή μας απλό βέλος) που επιστρέφει (*res1*) απέχει από την αρχή του `intArr1` 7 θέσεις τύπου `int`.

Στη δεύτερη κλήση της *transform()* κάθε τετράγωνο που υπολογίζεται αντικαθιστά το αρχικό στοιχείο του `intArr`. Ο προσεγγιστής που επιστρέφει (*res*) απέχει από την αρχή του `intArr` 7 θέσεις τύπου `int`.

Πρόσεξε τώρα τη δεύτερη μορφή: το συναρτησιακό αντικείμενο που έχουμε εδώ έχει δύο παραμέτρους και δρα στα αντίστοιχα στοιχεία δύο περιοχών με ίσα μήκη. Το αποτέλεσμα κατά τη φύλαξή του μπορεί να αντικαταστήσει είτε την πρώτη περιοχή είτε τη δεύτερη (είτε καμία από τις δύο). Στις παραμέτρους υπάρχει μια «ασυμμετρία»:

- Για την πρώτη περιοχή μας δίνεται αρχή και τέλος: `[`*first1*, *last1*`)`.
- Για τη δεύτερη περιοχή μας δίνεται μόνο η αρχή *first2* και το τέλος συνάγεται από το ότι θα πρέπει να έχει το ίδιο μήκος με την πρώτη.

Αν στο προηγούμενο παράδειγμα ορίσουμε επί πλέον

```
template< typename T >
struct Sum
{
   T operator()( T x, T y )  {  return x + y;  }
};
```

και δηλώσουμε:

```
   Sum<int> sum;
   int      intArr2[7];
```

τότε οι:

```
   transform( intArr, intArr+7, intArr1, intArr2, sum );
   copy( intArr2, intArr2+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
```

θα μας δώσουν:

```
1190 5256 506 72 110 992 1722
```

Όπως βλέπεις, το 1190 είναι 34+1156, 5256 == 72+5184 κ.ο.κ. Δηλαδή, η *sum* καλείται 7 φορές για να υπολογίσει τα `intArr[k] + intArr1[k]`. Το αποτέλεσμα φυλάγεται στο `intArr2[k]`. Θα μπορούσαμε, αντί για `intArr2`, να είχαμε βάλει `intArr` ή `intArr1`.

Η τρίτη «συνάρτηση για τα πάντα» είναι η *random_shuffle()* που έχει δύο μορφές:

```
template< class RandomAccessIter >
void random_shuffle( RandomAccessIter first, RandomAccessIter last);
template< class RandomAccessIter, class RandomNumberGenerator >
void random_shuffle( RandomAccessIter first, RandomAccessIter last,
                     RandomNumberGenerator& rand );
```

Αυτή ανακατεύει με τυχαίο πρόπο τις τιμές που δείχνουν οι προσεγγιστές της περιοχής `[`*first1*, *last1*`)`. Στη δεύτερη μορφή έχεις τη δυνατότητα να ορίσεις δική σου γεννήτρια τυχαίων αριθμών με μορφή συναρτησιακού αντικειμένου *rand*, με επιφορτωμένο τον "`()`", τέτοια ώστε η "`rand(n)`" να επιστρέφει τυχαίο ακέραιο στο `[`0`..`*n*`)`.

Ως παράδειγμα δες το ανακάτεμα της τράπουλας:

```
#include <iostream>
#include <algorithm>

using namespace std;
```

[2] Δεν είναι απαραίτητο να είναι περίγραμμα.

```
int main()
{
   char* cardDeck[52] = { "S2","S3","S4","S5","S6","S7","S8","S9","S10","SJ",
                                                          "SQ","SK","SA",
                          "H2","H3","H4","H5","H6","H7","H8","H9","H10","HJ",
                                                          "HQ","HK","HA",
                          "D2","D3","D4","D5","D6","D7","D8","D9","D10","DJ",
                                                          "DQ","DK","DA",
                          "C2","C3","C4","C5","C6","C7","C8","C9","C10","CJ",
                                                          "CQ","CK","CA" };
   random_shuffle( cardDeck, cardDeck+52 );
   for ( int k(0); k < 52; ++k )
   {
      cout << cardDeck[k] << ' ';
      if ( k%13 == 12 )  cout << endl;
   }
}
```

Αποτέλεσμα:[3]

```
DK SK D5 HQ H7 D3 C4 CQ H9 H4 S6 D8 C10
S7 C9 C2 D9 DA H3 D10 CJ S10 H10 CA S9 H8
C3 SA C6 SJ H2 H5 HJ D6 D7 CK S2 C7 DJ
HA D2 H6 C8 S4 C5 S8 S3 SQ DQ HK S5 D4
```

26.1.4 Συναρτησιακά Αντικείμενα

Στη συνέχεια θα δούμε το περίγραμμα κλάσης "*set*" που είναι:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

Αφού στα περιγράμματα κλάσεων δεν μπορούμε να έχουμε επιφόρτωση, όπως έχουμε στα περιγράμματα συναρτήσεων, αλλά μπορούμε να έχουμε ερήμην «τιμές» των παραμέτρων του περιγράμματος, αντί να έχουμε δύο περιγράμματα "*set*", όπως έχουμε δύο περιγράμματα *sort()*, έχουμε αυτό το "`class Compare = less<Key>`" που μας λέει ότι αν γράψω "`set<T>`" εννοώ ότι οι συγκρίσεις θα γίνονται με το συναρτησιακό αντικείμενο `less<T>` όπου (§25.6.3):

```
template < typename T >
struct less
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Βάζοντας στο πρόγραμμά σου "`#include <functional>`" έχεις τη δυνατοτητα να χρησιμοποιείς συναρτησιακά αντικείμενα σαν το παραπάνω και τα άλλα παρόμοια περιγράμματα που υπάρχουν εκεί.

Όλα ξεκινούν με τα[4]

```
template < class Arg, class Result >
struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type;
}; // unary_function

template < typename Arg1, typename Arg2, typename Result >
```

[3] Πού είναι η τράπουλα; Βλέπε το "Sx" ως "♠x", το "Hx" ως "♥x", το "Dx" ως "♦x" και το "Cx" ως "♣x".

[4] Στο C++11 αποθαρρύνεται η χρήση των *unary_function*, *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

```
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
}; // binary_function
```

Τα άλλα περιγράμματα κληρονομούν αυτά τα δύο, για παράδειγμα:

```
template < class T >
struct negate : public unary_function< T, T >
{
   T operator() ( const T& x ) const { return -x; }
};

template < class T >
struct plus : public binary_function<T, T, T>
{
   T operator() ( const T& x, const T& y ) const { return x + y; }
};

template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

- Το *binary_function* κληρονομούν τα
  - *plus()* ("+"), *minus()* ("-"), *multiplies()* ("*"), *divides()* ("/"), *modulus()* ("%")
  - *equal_to()* ("=="), *not_equal_to()* ("!="), *greater()* (">"), *less()* ("<"), *greater_equal()* (">="), *less_equal()* ("<=")
  - *logical_and()* ("&&"), *logical_or()* ("||")
- Το *unary_function* κληρονομούν τα
  - *negate()* ("-")
  - *logical_not()* ("!")

Γυρνώντας στο παράδειγμα για τη δεύτερη μορφή της *transform()* βλέπουμε ότι δεν χρειάζεται να γράψουμε το περίγραμμα *Sum*. Βάζουμε στο πρόγραμμα την "`#include <functional>`", αλλάζουμε τη δήλωση του *sum* σε:

```
   plus<int> sum;
```

και παίρνουμε τα ίδια αποτελέσματα.

Τα εργαλεία αυτά μας βοηθούν να χρησιμοποιούμε τις συναρτήσεις του `algorithm` και τα περιέχοντα της STL. Για τον λόγο αυτόν μας δίνονται και διάφοροι προσαρμογείς. Με ένα παράδειγμα ας δούμε εργαλεία που μας δίνονται ώστε να χρησιμοποιούμε συναρτησεις με δύο παραμέτρους εκεί που απαιτείται συνάρτηση με μια.

Παράδειγμα

Στο Project04, θέλουμε να εφοδιάσουμε την κλάση *CourseCollection* με μια μέθοδο:

```
Course* CourseCollection::findDep( string code )
```

που θα μας επιστρέφει βέλος προς στοιχείο (τύπου *Course*) που έχει ως προαπαιτούμενο (*prereq*) το στοιχείο με κωδικό *code*. Αν δεν υπάρχει τέτοιο στοιχείο θα επιστρέφει "`0`" (*NULL*).

Ένας τρόπος να λύσουμε το πρόβλημα είναι αυτός που έχουμε στη μέθοδο *delete1-Course()*· εδώ όμως θέλουμε λύσουμε το πρόβλημα χρησιμοποιώντας το

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Το στιγμιότυπο που μας ενδιαφέρει θα είναι:

```
Course* find_if( ccArr, ccArr+ccNOfCourses, pred???? );
```

Τι θα είναι εκείνο το "`pred????`"; Εμείς ψάχνουμε ένα στοιχείο *s* για το οποίο "`s.prereq == code`". Αυτό θα μπορούσαμε να το κωδικοποιήσουμε με ένα κατηγόρημα:

```
struct EqPrereq
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Αλλά εδώ έχουμε ένα κατηγόρημα με δύο παραμέτρους ενώ το "`pred????`" πρέπει να έχει μια παράμετρο που θα αντικαθίσταται με τα `ccArr[0]`, `ccArr[1]` κ.ο.κ. Αυτό μπορεί να προκύψει από το *EqPrereq* ως εξής:

```
bind2nd(EqPrereq(),code)
```

που σημαίνει: συνάρτηση με μια παράμετρο που προκύπτει από την *EqPrereq()* αν βάλουμε τη δεύτερη παράμετρο ίση με "`code`".

Για να δουλέψει η *bind2nd()* θα πρέπει το (πρώτο) όρισμά της να κληρονομεί ένα στιγμιότυπο της *binary_function*. Γράφουμε λοιπόν:

```
struct EqPrereq :  binary_function< Course, string, bool >
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Η συνάρτηση που θέλουμε να γράψουμε θα είναι:

```
Course* CourseCollection::findDep( string code )
{
   Course* k;
   k = find_if( ccArr, ccArr+ccNOfCourses, bind2nd(EqPrereq(),code) );
   if ( k == ccArr+ccNOfCourses )  k = 0;
   return k;
} // findDep
```

Εκτός από το περίγραμμα *bind2nd()* υπάρχει και το *bind1st()* που μας επιτρέπει να «καρφώσουμε» το πρώτο όρισμα. Τα δύο περιγράμματα στηρίζονται στα περιγράμματα-προσαρμογείς *binder2nd* και *binder1st* αντιστοίχως. Να επαναλάβουμε ότι για να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων σε μια συνάρτηση *f()* με δύο παραμέτρους θα πρέπει η αντίστοιχη συναρτησιακή κλάση *f* να κληρονομεί την *binary_function*.[5]

26.2 Ακολουθίες

Όπως είπαμε παραπάνω στις *ακολουθίες* η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Έτσι:

- Ένα στιγμιότυπο της *vector* είναι κλάση μονοδιάστατων δυναμικών πινάκων. Αν έχουμε "`vector<K> vt`" τότε το "`vt[j]`" μας δίνει πρόσβαση στο αντίστοιχο στοιχείο του πίνακα με την ίδια ευκολία για οποιαδήποτε (νόμιμη) τιμή του "`j`".
- Ένα στιγμιότυπο της *list* είναι κλάση που κάθε της αντικείμενο είναι λίστα με διπλή σύνδεση. Αν έχουμε "`list<K> lt`" μπορείς να αρχίσεις να διασχίζεις την *lt* από την αρχή της (με τον *lt.begin()*) ή από το τέλος της (με τον *lt.end()*). Αν έχεις κάποιον προσεγγιστή *it* προς κάποιο στοιχείο της λίστας μπορείς να τον μετακινήσεις προς το επόμενο στοιχείο (με την "`++it`") ή προς το προηγούμενο (με την "`--it`") αν, φυσικά, υπάρχουν.

[5] Στο C++11 αποθαρρύνεται η χρήση των *bind2nd()*, *bind1st()*, *binder2nd*, *binder1st* όπως και της *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

- Ένα στιγμιότυπο της *deque* είναι κλάση που κάθε της αντικείμενο είναι ουρά δύο άκρων. Η λειτουργικότητα ενός τέτοιου αντικειμένου είναι παρόμοια με αυτήν ενός αντικειμένου *vector*, έχουμε όμως και τη δυνατότητα να εισάγουμε και να εξάγουμε αντικείμενα της περιεχόμενης κλάσης από την αρχή.

Όλες οι ακολουθίες `S<K>` έχουν –πέρα από αυτά που είπαμε για τα περιέχοντα γενικώς– και άλλες συναρτήσεις-μέλη που είναι:

- **Δημιουργοί πλήρωσης** (fill constructors).[6]

  ```
  explicit S<K>( size_type sz, const K& val = K() );
  ```

  Με τη δήλωση:

  ```
  S<K> a( n, c );
  ```

  –όπου $n \geq 0$ και *c* αντικείμενο κλάσης *K*– το *a* περιέχει *n* αντίγραφα του *c*. Για παράδειγμα, μετά την

  ```
  list< Date >  ld( 3, Date(2010, 1, 1) );
  ```

  η *ld* είναι λίστα με τρία αντίγραφα της "`1.1.2010`". Παρόμοια λίστα (χωρίς όνομα) δημιουργείται από την παράσταση:

  ```
  list< Date >( 3, Date(2010, 1, 1) )
  ```

  Με τη δήλωση:

  ```
  S<K> a( n );
  ```

  το *a* περιέχει *n* αντίγραφα του *K()*. Έτσι, αν δηλώσουμε:

  ```
  vector< Date >  vd( 5 );
  ```

  ο *vt* είναι πίνακας με πέντε αντίγραφα της "`1.1.1`". Παρόμοιος πίνακας (χωρίς όνομα) δημιουργείται και από την παράσταση: "`vector<Date>(5)`"

- **Δημιουργοί περιοχής** (range constructors).

  ```
  template < class InputIterator >
  S<K>( InputIterator first, InputIterator last );
  ```

  Με τη δήλωση:

  ```
  S<K> a( j, k );
  ```

  –όπου τα *j*, *k* είναι προσεγγιστές σε κάποιο άλλο περιέχον (ή απλά βέλη προς στοιχεία απλού πίνακα) και η `[j, k)` είναι μια μη κενή περιοχή με αντικείμενα κλάσης *K*– το *a* περιέχει αντίγραφα όλων των αντικειμένων της περιοχής. Για παράδειγμα, μετά τις

  ```
  int v0[10] = { 0,1,2,3,4,5,6,7,8,9 };
  int* p1( &v0[5] );  int* p2( &v0[8] );
  deque< int > dqt( p1, p2 );
  ```

  η *dqt* περιέχει 3 στοιχεία: τα "`5`", "`6`", "`7`".

  Ακόμη, στις τρεις «νέες» μορφές του παραδείγματος με το οποίο ξεκινήσαμε, μπορείς να προσθέσεις άλλη μια όπου η ανάγνωση των στοιχείων γίνεται με τη δήλωση του πίνακα: αφού δηλώσεις τους δύο *istream_iterator*s χρησιμοποίησε αυτόν τον δημιουργό για να δηλώσεις τον `array`:

  ```
     istream_iterator< unsigned int > cinIt( cin );
     istream_iterator< unsigned int > cinEoStream;
     vector< unsigned int > array( cinIt, cinEoStream );
     sort( array.begin(), array.end() );
  // . . .
  ```

- Μέθοδοι για την **κεφαλή** (head) του περιέχοντος:

  ```
  K& front();
  const K& front() const;
  ```

  (η δεύτερη για αντικείμενα `const`.) Επιστρέφει (αναφορά προς) το πρώτο αντικείμενο (κεφαλή) του περιέχοντος. Με άλλα λόγια το: `a.front()` είναι ίσο με `*(a.begin())`.

[6] Παραλείπουμε την τρίτη παράμετρο (παραχωρητής μνήμης).

- Μέθοδοι εισαγωγής στοιχείων:

```
S<K>::iterator insert( S<K>::iterator pos, const K& val );
void insert( S<K>::iterator pos, size_type sz, const K& val );
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

Η παράμετρος "`S<K>::iterator pos`" πρέπει να είναι προσεγγιστής στο περιέχον που κάνουμε την εισαγωγή.

Η πρώτη μέθοδος εισάγει αντίγραφο της *val* πριν από το στοιχείο που δείχνει ο *pos* και αυξάνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που εισάχθηκε.

Η δεύτερη εισάγει *sz* αντίγραφα της *val* πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *sz*.

Η τρίτη (περίγραμμα) εισάγει ολόκληρη την περιοχή [*first*, *last*) πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *last - first*. Η προς αντιγραφή περιοχή μπορεί να βρίσκεται στο ίδιο ή σε άλλο αντικείμενο.

Προσοχή: Η εισαγωγή στοιχείων σε μια ακολουθία μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας. Αυτό ισχύει και για το *vector* αφού με την εισαγωγή στοιχείου μπορεί να γίνει νέα παραχώρηση μνήμης. Έτσι, ο προσεγγιστής που επιστρέφει η πρώτη μορφή της *insert()* δεν είναι απαραίτητο να ισούται με τον (εισερχόμενο) *pos*.

- Μέθοδοι διαγραφής στοιχείων:

```
S<K>::iterator erase( S<K>::iterator pos );
S<K>::iterator erase( S<K>::iterator first,
                      S<K>::iterator last );
void clear();
```

Η πρώτη μέθοδος διαγράφει το στοιχείο που δείχνει ο *pos* και μειώνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που ακολουθούσε αυτό που διαγράφηκε.

Η δεύτερη μέθοδος διαγράφει όλα τα στοιχεία της περιοχής [*first*, *last*). Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που αρχικώς έδειχνε ο *last*.

Η τρίτη διαγράφει όλα τα περιεχόμενα. Η `a.clear()` είναι ισοδύναμη με `a.erase( a.begin(), a.end() )`.

Προσοχή: Και η διαγραφή στοιχείων ακολουθίας μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας.

- Μέθοδος αλλαγής μήκους

```
void resize( size_type sz, K val = K() );
```

Δίνοντας "`vt.resize( n, kv );`"

αν *n* < *vt*.*size*`()` είναι σαν να δίνεις

"`vt.erase( vt.begin()+n, vt.end() );`"

ενώ αν *n* > *vt*.*size*`()` είναι σαν δίνεις

"`vt.insert( vt.end(), n-vt.size(), kv );`"

Τα τρία περιγράμματα ακολουθιών έχουν μερικές ακόμη ιδιότητες (που δεν υπάρχουν για τα επιπλέον περιγράμματα που προδιαγράφονται στο C++11):

- Είναι **αναστρέψιμα** (reversible) περιέχοντα. Όπως είπαμε παραπάνω, αυτό σημαίνει ότι έχουν τύπο *reverse_iterator* και τις μεθόδους *rbegin()*, *rend()*.
- Δίνουν εργαλεία για **εισαγωγή** (και **διαγραφή**) **στο τέλος** (back insertion).
  - Μέθοδο για να πάρουμε το τελευταίο στοιχείο του περιέχοντος (αν δεν είναι κενό):

```
K& S<K>::back();
```

```
const K& S<K>::back();
```

Το "`a.back()`" είναι ισοδύναμο με το "`*(--a.end())`".

- Μέθοδο για εισαγωγή στο τέλος της συλλογής του περιέχοντος:

```
void S<K>::push_back( const K& );
```

Μετά την "`a.push_back( c );`" η "`a.back()`" μας δίνει το *c*.

- Μέθοδο για τη διαγραφή του τελευταίου στοιχείου της συλλογής του περιέχοντος (αν δεν είναι κενό):

```
void S<K>::pop_back();
```

26.2.1 Το Περίγραμμα "*vector*"

Πολλοί προγραμματιστές «βολεύονται» απλώς και μόνον με δυναμικούς πίνακες. Έτσι, «περιφρονούν» τις άλλες περιέχουσες κλάσεις της STL, χρησιμοποιούν μόνον τη *vector* (διάνυσμα) που γίνεται η ευρύτερα χρησιμοποιούμενη περιέχουσα κλάση της STL. «Βολεύονται» μόνο από τεμπελιά; Όχι!

- ***Το περίγραμμα "vector" είναι η μόνη περιέχουσα κλάση της STL[7] που εγγυάται αποθήκευση των στοιχείων σε συναπτές θέσεις της μνήμης.***

Έτσι, μπορεί κανείς να χρησιμοποιεί απλά βέλη –αλλά και συναρτήσεις– που έχουν γραφεί για συνήθεις (μονοδιάστατους) πίνακες για να χειριστεί πίνακες *vector*. Ας δούμε ένα

Παράδειγμα

Στην §12.3.5 είχαμε δει τη συνάρτηση

```
double vectorSum( const double x[], int n, int from, int upto )
{
   if ( x == 0 && n > 0 )
   {  cerr << "η vectorSum κλήθηκε με ανύπαρκτο πίνακα" << endl;
      exit( EXIT_FAILURE );  }
// άλλοι έλεγχοι
   double sum( 0 );
   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

που υπολογίζει το άθροισμα των στοιχείων από `x[from]` μέχρι και `x[upto]` ενός πίνακα με *n* στοιχεία τύπου `double`. Μπορούμε να τη χρησιμοποιήσουμε για τον

```
   vector< double > vx;
```

Βεβαίως, και να πώς:

```
ut = vectorSum( &vx[0], vx.size(), 1, 2 );
```

Η αρχή του πίνακα βρίσκεται στη `&vx[0]`[8] και το πλήθος των στοιχείων του είναι `vx.size()`.

Το περίγραμμα *vector* δηλώνεται ως εξής:

```
template < class K, class Alloc = allocator<K> >
class vector;
```

Να δούμε τώρα τις δυνατότητες –για την ακρίβεια: τις μεθόδους– που προσφέρει η *vector* πέρα από αυτές που έχουν οι ακολουθίες γενικώς:

- Η πρόσβαση σε κάθε στοιχείο γίνεται (θυμίσου και τη *string*) με τον τελεστή "`[]`" και τη μέθοδο *at()*.

```
K& operator[]( size_type n );
const K& operator[]( size_type n ) const;
```

---

[7] Αυτό για τη C++03. Η STL της C++11 έχει και άλλες.

[8] Όπως θα δεις στη συνέχεια, στη *vector* επιφορτώνουμε τον "`[]`".

```
K& at( size_type n );
const K& at( size_type n ) const;
```

(Οι περιπτώσεις "`const`" για σταθερά αντικείμενα.) Όπως και στη *string*, η διαφορά έγκειται στο ότι η *at()* ελέγχει την τιμή του *n* και αν είναι ≥ *size()* ρίχνει εξαίρεση (*std*::)*out_of_range*.

- Όπως για τη *string* (§21.10.1) έτσι και για κάθε στιγμιότυπο του *vector* έχει νόημα η **χωρητικότητα** (capacity): το πλήθος αντικειμένων τύπου *K* που μπορούμε να αποθηκεύσουμε στη μνήμη που έχει ήδη δεσμευθεί για ένα αντικείμενο `vector<K>`. Μας τη δίνει η μέθοδος:

  ```
  size_type capacity() const;
  ```

  Προφανώς, για οποιοδήποτε αντικείμενο *a* τέτοιου τύπου θα έχουμε:

  `a.size() ≤ a.capacity()`

  Αν το `a.size()` πρέπει υπερβεί την τρέχουσα `a.capacity()` τότε θα πρέπει να γίνει νέα παραχώρηση μνήμης ώστε να μεγαλώσει η χωρητικότητα και να γίνουν οι απαραίτητες αντιγραφές (αυτά που κάνουμε με τη *renew()*). Βάζοντας αρκετά μεγάλη τιμή χωρητικότητας –με τη μέθοδο *reserve()* που ακολουθεί– περιορίζεις αυτές τις χρονοβόρες διαδικασίες.

- Η χωρητικότητα ενός αντικειμένου *vector* αλλάζει με τη

  ```
  void reserve( size_type n );
  ```

  Αν η *n* έχει τιμή μεγαλύτερη από την τρέχουσα χωρητικότητα (`a.capacity()`) τότε δεσμεύεται περισσότερη μνήμη ώστε να έχουμε `a.capacity()` ≥ *n*. Αυτό συνήθως απαιτεί αντιγραφή των περιεχομένων του *a*.

Επειδή με τις *size()*, *resize()*, *capacity()*, *reserve()* μπορεί να υπάρχουν μπερδεματα ας προσπαθήσουμε να τα ξεκαθαρίσουμε με ένα προγραμματάκι:

Παράδειγμα

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    vector<int> av( 3 );
 8:
 9:    cout << "capacity: " << av.capacity()
10:         << "   size: " << av.size() << endl;
11:    for ( int i(0); i <= 6; ++i )
12:    {
13:       if ( i < av.size() )  av[i] = i*10;
14:                       else  av.push_back( i*10 );
15:       cout << "i: " << i << "   capc: " << av.capacity()
16:            << "   size: " << av.size() << "   ";
17:       for ( int k(0); k < av.size(); ++k )
18:          cout << av[k] << ' ';
19:       cout << endl;
20:    }
21:
22:    av.reserve( 31 );
23:    av.resize( 4 );
24:    cout << "capacity: " << av.capacity()
25:         << "   size: " << av.size() << endl;
26:    for ( int k(0); k < av.size(); ++k )  cout << av[k] << ' ';
27:    cout << endl;
28:    try { cout << av.at(5) << endl; }
29:    catch( out_of_range& x )
30:    {  cout << x.what() << endl;  }
31: }
```

που μας δίνει:

```
capacity: 3   size: 3
i: 0   capc: 3   size: 3   0 0 0
i: 1   capc: 3   size: 3   0 10 0
i: 2   capc: 3   size: 3   0 10 20
i: 3   capc: 6   size: 4   0 10 20 30
i: 4   capc: 6   size: 5   0 10 20 30 40
i: 5   capc: 6   size: 6   0 10 20 30 40 50
i: 6   capc: 12   size: 7   0 10 20 30 40 50 60
capacity: 31   size: 4
0 10 20 30
index out of range in function: vector:: at(size_t)
 index: 5 is greater than max_index: 4
```

Με τη δήλωση της γραμμής 7 δηλώνουμε έναν (δυναμικό) πίνακα ακεραίων με αρχικό μέγεθος 3, δηλαδή με 3 στοιχεία. Από τη γραμμή 9 παίρνουμε το αποτέλεσμα "`capacity: 3  size: 3`". Η χωρητικότητα έγινε αυτομάτως `3`. Αυτή είναι η ελάχιστη τιμή για την οποίαν ικανοποιείται η: `av.size() ≤ av. capacity()`.

Στις πρώτες 3 εκτελέσεις (*i*: 0, 1, 2) της **for** δίνουμε τιμές στα `va[i]` με εκχώρηση (γραμμή 13) αφού τα στοιχεία αυτά υπάρχουν.

Για την εισαγωγή της 4ης τιμής ("`3`") χρησιμοποιούμε την "`av.push_back(i*10)`" διότι θέλουμε να μεγαλώσουμε ταυτοχρόνως τον πίνακα. Το ίδιο ισχύει και για τις επόμενες τιμές.

Όπως φαίνεται στη γραμμή "`i: 3`" η *av.size()* επέστρεψε τιμή "`4`" –αυξήθηκε κατά 1– αλλά η *av.capacity()* επέστρεψε "`6`"· διπλασιάστηκε με τη νέα παραχώρηση μνήμης που μεσολάβησε! Στη γραμμή "`i: 6`" βλέπουμε τι έγινε για να μπορέσουμε να βάλουμε και έβδομη τιμή: το μέγεθος αυξήθηκε κατά 1 αλλά η χωρητικότητα ξαναδιπλασιάστηκε· είχαμε και πάλι νέα παραχώρηση. Έτσι δουλεύει ο παραχωρητής μνήμης του *vector* και αυτό το έχουμε συζητήσει ήδη στην §16.13.3.

Η γραμμή "`capacity: 31  size: 4`" μας δείχνει τα αποτελέσματα των εντολών που δώσαμε στις γραμμές 22 και 23. Το αποτέλεσμα της **for** της γραμμής 26 μας επιβεβαιώνει το "`size: 4`".

Άλλη μια επιβεβαίωση για το μέγεθος μας δίνει και η απόπειρα χρήσης του 6ου στοιχείου του πίνακα με την "`av.at(5)`" μέσα σε μια `try`. Ρίχνεται εξαίρεση *out_of_range* από τη *what()* της οποίας παίρνουμε τις δύο τελευταίες γραμμές.

Τώρα θα πούμε και μερικά πράγματα για τους προσεγγιστές του *vector*. Αν ψάξεις λίγο στο αρχείο `vector` της C++ που χρησιμοποιείς θα δεις κάτι σαν:

```
template < class T, class Allocator = allocator<T> >
class vector
{
// . . .
public:
   typedef T           value_type;
// . . .
   typedef value_type* iterator;
// . . .
   typedef ptrdiff_t   difference_type;
// . . .
```

Δηλαδή: ο προσεγγιστής "`vector<T>::iterator`" είναι βέλος "`T*`" προς στοιχείο πίνακα. Έτσι, γίνεται αυτομάτως προσεγγιστής τυχαίας πρόσβασης και έχουμε δυνατότητα για αριθμητική βελών.

Στο παραπάνω παράδειγμα θα μπορούσαμε, μετά τη γραμμή 20, να βάλουμε:

```
   cout << *(av.begin()+3) << endl;
   vector<int>::iterator it1( av.begin() ),
                         it2( av.begin() );
   it1 += 2;
```

```
   it2 += 6;
   vector<int>::difference_type dist( it2 - it1 );
   cout << "dist: " << dist << endl;
```

και να πάρουμε:

```
30
*it1: 20   *it2: 60
dist: 4
```

Και κάτι για τις ακυρώσεις προσεγγιστών:

- ***Αν έχεις προσεγγιστές προς στοιχεία κάποιου αντικειμένου* vector *και κάνεις κάποια εισαγωγή (insert) ή διαγραφή (erase) όσοι προσεγγιστές έδειχναν από το πρώτο στοιχείο που εισάχθηκε (ή διαγράφηκε) και μετά ακυρώνονται (δεν δείχνουν αυτό που έδειχναν).***

Αν το ξεχάσεις μπορεί να «υποφέρεις» από μερικά δύσκολα προγραμματιστικά λάθη.

26.2.1.1 Εξαιρέσεις, Απόδοση και Άλλα

Με πόση ασφάλεια και πόσο γρήγορα γίνονται οι πράξεις σε αντικείμενα *vector;* Ας τις δούμε μια προς μια.

Πρώτα οι διαγραφές: Ας πούμε ότι έχεις ένα αντικείμενο *v* τύπου `vector <K>` –με *n* στοιχεία τύπου *K*– και δίνεις την εντολή "`v.erase(f, l);`".

Αν στα προς διαγραφή στοιχεία περιλαμβάνεται και το τελευταίο στοιχείο (αν δηλαδή *l* `==` *v*`.end()`) δεν γίνονται αντιγραφές. Απλώς ο παραχωρητής παίρνει πίσω τα τελευταία *l - f* στοιχεία και καταστρέφει τα αντικείμενά τους καλώντας τον `~K()`. Στην περίπτωση αυτή (καλώς εχόντων των πραγμάτων) δεν εγείρεται εξαίρεση. Ο χρόνος που απαιτείται είναι αυτός που χρειάζεται για την καταστροφή των *l - f* αντικειμένων.

Έστω τώρα ότι ο *f* δείχνει το `v[j]` και ο *l* δείχνει το `v[k]` (όπου *k - j = l - f*), δύο ενδιάμεσα στοιχεία του πίνακα. Τι πρέπει να γίνει;

- Να αντιγραφούν τα `v[k]..v[n-1]` στα στοιχεία που ξεκινούν από το `v[j]` (`copy(l, v.end(), f)`).
- Να καταστραφούν τα τελευταία *l - f* στοιχεία και να μειωθεί το μέγεθος του πίνακα κατά *l - f*.

Δηλαδή: έχουμε γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-1-*k*) των στοιχείων από το `v[k+1]` μέχρι το τέλος του πίνακα.

Από τις αντιγραφές μπορεί να έχουμε εξαίρεση. Πάντως δεν θα έχουμε διαρροή μνήμης και το *v* θα είναι διαχειρίσιμο (βασική εγγύηση).

Και για να επανέλθουμε σε αυτό που λέμε πιο πάνω: *αν έχεις προσεγγιστές που –πριν τη διαγραφή– δείχνουν σε στοιχεία από* `v[j]` *μέχρι το τέλος του πίνακα, μετά τη διαγραφή ακυρώνονται.*

Ας πάμε τώρα στην εισαγωγή. Έστω ότι θέλουμε να εισαγάγουμε *m* αντικείμενα ξεκινώντας από τη θέση `v[k]`. Θα πρέπει:

- Να μεγαλώσει το μέγεθος του πίνακα κατά *m* (σε *n*+*m*). Υποθέτουμε ότι δεν θα χρειαστεί νέα παραχώρηση μνήμης.
- Να αντιγραφούν τα `v[k]..v[n-1]` στα `v[k+m]..v[n+m-1]` αντιστοίχως.
- Να αντιγραφούν τα *m* εισαγόμενα αντικείμενα στις `v[k]..v[k+m-1]`.

Έχουμε δηλαδή και εδώ γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-*k*) των στοιχείων από το `v[k]` μέχρι το τέλος του πίνακα και το πλήθος *m* των εισαγόμενων στοιχείων. Αλλά:

- Αν δεν ισχύει η υπόθεση που κάναμε και *έχουμε* νέα παραχώρηση μνήμης τότε θα πρέπει να γίνουν και *n* αντιγραφές. Στην περίπτωση αυτήν μπορεί –κατ' αρχήν τουλάχιστον– να πάρουμε *bad_alloc*.

- Ο χρόνος εισαγωγής στο τέλος εξαρτάται μόνο από το πλήθος των στοιχείων που εισάγονται. Φυσικά, για εισαγωγή ενός στοιχείου είτε με την *push_back()* είτε με κάποια *insert()* ο χρόνος είναι σταθερός. Σταθερός είναι ο χρόνος και την *pop_back()*.
- Για την τρίτη συνάρτηση (περίγραμμα) εισαγωγής

```
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

τα πράγματα μπορεί να είναι πιο πολύπλοκα. Οι *first* και *last* είναι –γενικώς– προσεγγιστές εισόδου. Και αν μεν είναι πρόσθιοι μπορούν να γίνουν αυτά που είπαμε παραπάνω. Αν όμως είναι, ας πούμε, *istream_iterator*s τότε δεν μπορούμε να μετρήσουμε και να βρούμε το *m* εκ των προτέρων. Τα στοιχεία θα εισάγονται ένα προς ένα και θα γίνουν *m* φορές αυτά που είπαμε πιο πάνω (αν τα εξειδικεύσεις για $m = 1$).

Εκτός από τη *bad_alloc*, που προαναφέραμε, μπορεί να έχουμε εξαίρεση από τις αντιγραφές. Σε κάθε περίπτωση έχουμε και εδώ ασφάλεια επιπέδου βασικής εγγύησης.

26.2.1.2 Και μια Εξειδίκευση: "vector<bool>"

Όπως έχουμε παρατηρήσει και σε προηγούμενο κεφάλαιο, από τα 8 ψηφία της ψηφιολέξης με την οποίαν παριστάνεται μια τιμή τύπου `bool` χρησιμοποιούμε μόνον το ένα. Για να γίνει η σχετική οικονομία, η STL προσφέρει μια εξειδίκευση του *vector* για τον τύπο `bool`:

```
template <class Allocator>
class vector<bool, Allocator>
```

Πέρα από την οικονομία μνήμης, η εξειδίκευση έχει δύο επιπλέον συναρτήσεις:

- Τη μέθοδο:

```
void flip();
```

που αλλάζει τις τιμές όλων των περιεχομένων από *b* σε *~b*· δηλαδή αλλάζει όλα τα `false` σε `true` και όλα τα `true` σε `false`.

- Πέρα από τη μέθοδο *swap()*, που ανταλλάσσει τις τιμές δύο πινάκων, τη στατική συνάρτηση:

```
static void swap( vector<bool>::reference ref1,
                  vector<bool>::reference ref2 );
```

που ανταλλάσσει τις τιμές δύο στοιχείων του πίνακα. Το "`reference`" να το σκέφτεσαι ως "`bool&`" (αλλά διάβασε παρακάτω.)

Παρατήρηση:▶

Το περίγραμμα *vector* έχει γενικώς έναν τύπο "`reference`" που για το περίγραμμα `vector<K>` ορίζεται ως "`K&`". Ειδικώς όμως για το `vector<bool>` υπάρχει πρόβλημα: Όπως έχουμε πει, οι αναφορές είναι «κρυμμένα» βέλη, δηλαδή διευθύνσεις. Αν λοιπόν «στριμώξουμε» το κάθε στοιχείο μιας τέτοιας κλάσης σε ένα δυαδικό ψηφίο τότε την ίδια διεύθυνση τη μοιράζονται 8 (ή περισσότερα) στοιχεία. Για να λυθεί αυτό το πρόβλημα ορίζεται (μέσα στο `vector<bool>`) ολόκληρη κλάση `reference`.◀

Δες το παρακάτω προγραμματάκι:

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: void display( ostream& tout, const vector<bool>& bv )
 5: {
 6:    tout << boolalpha;
 7:    for ( int k(0); k < bv.size(); ++k ) tout << bv[k] << ' ';
 8:    tout << endl;
 9: }
10:
11: int main ()
12: {
```

```
13:    vector<bool> bv1( 3 ) , bv2( 2 );
14:
15:    bv1[0] = true;  bv1[1] = false;  bv1[2] = true;
16:    cout << "bv1: ";  display( cout, bv1 );
17:    bv1.flip();
18:    cout << "bv1: ";  display( cout, bv1 );
19:
20:    bv2[0] = true;  bv2[1] = false;
21:    cout << "bv2: ";  display( cout, bv2 );
22:
23:    vector<bool>::swap( bv1[0], bv1[1] );
24:    cout << "bv1: ";  display( cout, bv1 );
25:    vector<bool>::swap( bv2.front(), bv2.back() );
26:    cout << "bv2: ";  display( cout, bv2 );
27:    vector<bool>::swap( bv1[1], bv2[1] );
28:    cout << "bv1: ";  display( cout, bv1 );
29:    cout << "bv2: ";  display( cout, bv2 );
30:
31:    bv1.swap( bv2 );
32:    cout << "bv1: ";  display( cout, bv1 );
33:    cout << "bv2: ";  display( cout, bv2 );
34: }
```

που βγάζει:

```
bv1: true false true
bv1: false true false
bv2: true false
bv1: true false false
bv2: false true
bv1: true true false
bv2: false false
bv1: false false
bv2: true true false
```

Πρόσεξε τα εξής:

- Η αλλαγή του *bv1* από την πρώτη γραμμή στη δεύτερη προκλήθηκε από την "`bv1.flip();`" της γραμμής 17.
- Η αλλαγή του *bv1* από τη δεύτερη στην τέταρτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 23 ενώ η αλλαγή του *bv2* από τη τρίτη στην πέμπτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 25.
- Η *swap()* της γραμμής 27 ανταλλάσσει τις τιμές στοιχείων διαφορετικών αντικειμένων.
- Τέλος, στη γραμμή 31 βλέπεις την «άλλη *swap()*», αυτήν που έχουν όλα τα περιέχοντα της STL. Το αποτέλεσμα φαίνεται στις δύο τελευταίες του απότελέσματος: τα *bv1* και *bv2* έχουν ανταλλάξει τιμές. Το ίδιο αποτέλεσμα θα είχαμε με την "`bv2.swap(bv1);`".

26.2.2 Το Περίγραμμα "*deque*"

Το περίγραμμα *deque* (προφέρεται "deck") υλοποιεί αυτό που λέει το όνομά της: μια ουρά δύο άκρων (*d*ouble *e*nded *que*ue).

Για να μπορεί να λειτουργήσει ως ουρά δύο άκρων χρειάζεται –εκτός από τις *front()*, *back()*, *push_back()*, *pop_back()*– και τις *push_front()*, *pop_front()* για εισαγωγή και διαγραφή στην αρχή του περιέχοντος που τις έχουμε δει στο *SListT* (§25.7.1):

- Εισαγωγή στοιχείου στην αρχή του περιέχοντος:
  `void deque<K>::push_front( const K& val );`
  Μετά την "`a.push_front( c );`" η "`a.front()`" μας δίνει το *c*· το ίδιο και η "`*a.begin()`".
- Διαγραφή του πρώτου στοιχείου της ακολουθίας του περιέχοντος (αν δεν είναι κενό):
  `void deque<K>::pop_front();`

Οι δύο αυτές μέθοδοι έχουν σταθερό χρόνο εκτέλεσης, όπως άλλωστε και οι *push_back()*, *pop_back()*.

Κατά τα άλλα, παρ' όλο που ο τρόπος αποθήκευσης των περιεχομένων είναι πιο πολύπλοκος από τον απλό μονοδιάστατο πίνακα, η λειτουργικότητα του *deque* είναι ίδια με αυτήν του *vector* με μια διαφορά: *Το deque δεν έχει τις reserve() και capacity()*.

26.2.3 Το Περίγραμμα "*list*"

Το περίγραμμα *list*:

```
template < class T, class Allocator = allocator<T> >
class list;
```

υλοποιεί λίστα με διπλή σύνδεση με περιεχόμενο στοιχείων τύπου *T*.

Για το *list*, πέρα από αυτά που έχει το *deque*, υπάρχουν επιπλέον:

- Μέθοδοι για μεταφορά στοιχείων (προσοχή: *μεταφορά όχι αντιγραφή*) σε μια λίστα από άλλη λίστα με στοιχεία και παραχωρητή μνήμης ίδιου τύπου:

  ```
  void splice( iterator pos, list<T,Allocator>& x );
  void splice( iterator pos, list<T,Allocator>& x, iterator i);
  void splice( iterator pos, list<T,Allocator>& x,
               iterator first, iterator last );
  ```

  Αν οι *l1*, *l2* είναι λίστες `list<K>` τότε:

  - Η "`l1.splice(it, l2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1*, μεταφέρει στην *l1* όλα τα στοιχεία της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it*. Η *l2* παραμένει χωρίς στοιχεία.
  - Η "`l1.splice(it, l2, it2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και ο *it2* προσεγγιστής προς στοιχείο της *l2*, μεταφέρει στην *l1* το στοιχείο που δείχνει ο *it2* και το συνδέει πριν από αυτό που δείχνει ο *it* ενώ το αφαιρεί από την *l2*. Αν δώσεις "`l1.splice(it, l1, it2);`" κάνεις εσωτερική μετακίνηση στην *l1*.
  - Η "`l1.splice(it, l2, f, l);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και οι *f*, *l* προσεγγιστές προς στοιχεία της *l2*, μεταφέρει στην *l1* το στοιχεία της περιοχής `[f,l)` της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it* ενώ τα αφαιρεί από την *l2*. Αν δώσεις "`l1. splice(it, l1, f, l);`" έχεις εσωτερική μετακίνηση στοιχείων στην *l1*. Στην περίπτωση αυτήν ο *it* πρέπει να είναι έξω από την `[f,l)`.
- Μέθοδος για διαγραφή στοιχείων με συγκεκριμένη τιμή και μέλος–περίγραμμα για διαγραφή στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void remove( const T& v );
  template < class Predicate >
  void remove_if( Predicate pred );
  ```

  Η "`l3.remove( v );`" έχει ως αποτέλεσμα την αφαίρεση από την *l3* όλων των στοιχείων που έχουν τιμή *v*.

  Μέσω του *pred()*, που είναι ένα κατηγόρημα με μια παράμετρο τύπου *T*, περνάμε στη *remove_if()* τις προδιαγραφές. Δες ένα

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
 6: {
 7:    for ( list<K>::iterator it(lv.begin());
```

```
 8:          it != lv.end(); ++it ) tout << *it << ' ';
 9:    tout << endl;
10: } // display
11:
12: struct Dec0
13: {
14:    bool operator()( const int& v )
15:    {  return ( (0 <= v) && (v < 10) );  }
16: }; // Dec0
17:
18: int main ()
19: {
20:    int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
21:    list<int> l3( ints, ints+11 );
22:    display<int>( cout, l3 );
23:
24:    l3.remove( 23 );  display<int>( cout, l3 );
25:
26:    l3.remove( 7 );  display<int>( cout, l3 );
27:
28:    l3.remove_if( Dec0() );  display<int>( cout, l3 );
29: }
```

δίνει:

```
3 23 3 19 5 19 17 7 7 17 23
3 3 19 5 19 17 7 7 17
3 3 19 5 19 17 17
19 19 17 17
```

Με την εντολή της γραμμής 24 ζητούμε να αφαιρεθούν από τη λίστα όλα τα "`23`" και παίρνουμε τη δεύτερη γραμμή του αποτελέσματος. Παρομοίως, με την εντολή της γραμμής 26 αφαιρούμε τα "`7`" και παίρνουμε την τρίτη γραμμή του αποτελέσματος.

Στις γραμμές 12-16 ορίζουμε ένα συναρτησοειδές (*Dec0*) που επιλέγει μονοψήφιους φυσικούς. Με την εντολή της γραμμής 28 αφαιρούμε από τη λίστα τους μονοψήφιους φυσικούς "`3`", "`3`" και "`5`".

Αντί για το συναρτησοειδές θα μπορούσαμε να ορίσουμε:

```
bool dec0( int v )  {  return ( (0 <= v) && (v < 10) );  }
```

και να πάρουμε το ίδιο αποτέλεσμα γράφοντας:

```
   l3.remove_if( dec0() );
```

- Μέθοδος για διαγραφή *διαδοχικών αντιγράφων ενός στοιχείου* και μέλος–περίγραμμα για διαγραφή διαδοχικών στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void unique();
  template < class BinaryPredicate >
  void unique( BinaryPredicate binaryPred );
  ```

  Για την πρώτη περίπτωση διασχίζεται η λίστα με έναν προσεγγιστή *it* και συγκρίνονται για ισότητα ζεύγη περιεχόμενων τιμών (`*it`, `*(it-1)`). Αν βρεθούν ίσα διαγράφεται αυτό που δείχνει ο *it*.

  Για τη δεύτερη μορφή, το κατηγόρημα «τροφοδοτείται» με τα `*it`, `*(it-1)`. Αν το κατηγόρημα επιστρέψει τιμή `true` διαγράφεται το στοιχείο που δείχνει ο *it*. Όπως καταλαβαίνεις, το κατηγόρημα πρέπει να είναι δυαδικό.

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
```

```
 6: // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
 7:
 8: struct NotInOrder
 9: {
10:    bool operator()( const int& v1, const int& v2 )
11:    {  return ( v1 > v2 );  }
12: }; // NotInOrder
13:
14: int main ()
15: {
16:    int ints[] = { 2, 2, 2, 3, 3, 5, 5, 5, 5, 2, 2, 3 };
17:    list<int> l4( ints, ints+12 );
18:    display<int>( cout, l4 );
19:
20:    l4.unique();  display<int>( cout, l4 );
21:
22:    l4.unique( NotInOrder() );  display<int>( cout, l4 );
23: }
```

δίνει:

```
2 2 2 3 3 5 5 5 5 2 2 3
2 3 5 2 3
2 3 5
```

Τι γίνεται με την εντολή της γραμμής 20; Μετά το πρώτο "`2`" διαγράφονται τα δύο αντίγραφά του ("`2`") που το ακολουθούν· τα ίδια ισχύουν και για τα "`3`" και τα "`5`". Το "`2`", που είναι προτελευταίο στοιχείο της ακολουθίας, αφαιρείται διότι υπάρχει "`2`" ακριβώς πριν από αυτό.

Το συναρτησοειδές *NotInOrder* ορίζει ένα κατηγόρημα που επιστρέφει `true` αν το πρώτο όρισμα είναι μεγαλύτερο από το δεύτερο. Η εντολή της γραμμής 24 έχει ως αποτέλεσμα να αφαιρεθεί το "`2`" που ακολουθεί το "`5`" και στη συνέχεια, να αφαιρεθεί και το "`3`" όταν βρίσκεται και αυτό να ακολουθεί το "`5`".

- Μέθοδος και μέλος–περίγραμμα για ταξινόμηση των περιεχομένων της λίστας:

  ```
  void sort();
  template < class Compare >
  void sort( Compare comp );
  ```

  Τα περιεχόμενα μιας λίστας τύπου `list<K>` ταξινομούνται με βάση τον τελεστή "`<`" της *K*.

  Αν θέλουμε ταξινόμηση χωρίς να έχουμε ορισμένο τον "`<`" της *K* ή με άλλη διάταξη από αυτήν που μας δίνει ο γράφουμε ένα συναρτησοειδές-κατηγόρημα που να ορίζει τη διάταξη ("`<`" της *K*) που μας ενδιαφέρει. Για την ταξινόμηση η "*a* `<` *b*" έχει το νόημα «το *a* προηγείται του *b*».

Παράδειγμα

Σε μια λίστα έχουμε τις ημερομηνίες ιστορικών γεγονότων:

```
Date dates[] = {  Date(1821,3,25), Date(1940,10,28),
                  Date(1904,10,13), Date(1453,5,29),
                  Date(1912,10,26)  };
list<Date> dateList( dates, dates+5 );
```

Αν δώσουμε:

```
dateList.sort();
display<Date>( cout, dateList );
```

θα πάρουμε:

```
29.5.1453 25.3.1821 13.10.1904 26.10.1912 28.10.1940
```

Όπως βλέπεις, εδώ έχουμε ιστορικές ημερομηνίες τις επετείους των οποίων γιορτάζουμε πανελληνίως ή τοπικώς. Θα ήταν λοιπόν ενδιαφέρον να τις έχουμε ταξινομημένες κατά μήνα και ημέρα. Για να το κάνουμε αυτό γράφουμε το συναρτησοειδές:

```
struct MonthFirst
{
   bool operator()( const Date& d1, const Date& d2 )
   {
      bool fv( false );
      if ( d1.getMonth() < d2.getMonth() )  fv = true;
      else if ( d1.getMonth() == d2.getMonth() )
      {
         if ( d1.getDay() < d2.getDay() )  fv = true;
         else if ( d1.getDay() == d2.getDay() )
            fv = ( d1.getYear() < d2.getYear() );
      }
      return fv;
   }
}; // MonthFirst
```

και γράφοντας:

```
   dateList.sort( MonthFirst() );
   display<Date>( cout, dateList );
```

θα πάρουμε:

```
25.3.1821 29.5.1453 13.10.1904 26.10.1912 28.10.1940
```

Αν, τώρα, θέλουμε τις ημερομηνίες από τις πιο πρόσφατες προς τις παλαιότερες (κατά «φθίνουσα» τάξη) γράφουμε το συναρτησοειδές:

```
struct Decr
{
   bool operator()( const Date& d1, const Date& d2 )
   {  return ( d2 < d1 );  }
}; // Decr
```

και με τις:

```
   dateList.sort( Decr() );
   display<Date>( cout, dateList );
```

να πάρουμε:

```
28.10.1940 26.10.1912 13.10.1904 25.3.1821 29.5.1453
```

Γιατί να προτιμήσουμε να δώσουμε "`l.sort()`" αντί για "`sort(l.begin(), l.end())`"; Η μέθοδος υπερέχει ως προς το ότι είναι **ευσταθής** (stable): η σχετική διάταξη δύο αντικειμένων με ίσα κλειδιά δεν αλλάζει κατά την ταξινόμηση. Προφανώς αυτό έχει νόημα όταν η λίστα φιλοξενεί σύνθετα αντικείμενα.

- Μέθοδος και μέλος–περίγραμμα για να συγχωνεύσουμε (τα περιεχόμενα από) δύο λίστες που είναι ήδη ταξινομημένες:

  ```
  void merge( list& x );
  template < class Compare >
  void merge( list& x, Compare comp );
  ```

  Αν έχουμε "`list<K> l1, l2`" και οι *l1*, *l2* είναι ταξινομημένες σύμφωνα με τον "`<`" της *K* τότε η "`l1.merge(l2);`" μεταφέρει (δεν αντιγράφει) και συγχωνεύει στην *l1* τα περιεχόμενα της *l2*. Τελικώς:

  - Η *l1* έχει όλα τα στοιχεία που είχαν αρχικώς οι *l1*, *l2* και είναι ταξινομημενα σύμφωνα με τον "`<`" της *K*.
  - Η *l2* είναι *κενή*.

  Αν οι *l1*, *l2* είναι ταξινομημένες με άλλη λογική χρησιμοποιούμε το περίγραμμα βάζοντας με τη δεύτερη παράμετρο τη διάταξη όπως κάναμε στην ταξινόμηση.

  Σε σχέση με τον αλγόριθμο (*std*::)*merge()* οι μέθοδοι έχουν ευστάθεια με την έννοια: μετά την "`l1.merge(l2);`" κάθε στοιχείο της *l2* εισάγεται μετά από το(τα) στοχείο(-α) της *l1* με το ίδιο κλειδί.

- Μέθοδος για την αντιστροφή μιας λίστας:

```
void reverse();
```

Για παράδειγμα, οι εντολές:

```
int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
list<int> l3( ints, ints+11 );
display<int>( cout, l3 );

l3.reverse();  display<int>( cout, l3 );
```

θα δώσουν:

```
3 23 3 19 5 19 17 7 7 17 23
23 17 7 7 17 19 5 19 3 23 3
```

26.2.4 Ποια Ακολουθία να Διαλέξω;

Η πιο συνηθισμένη απάντηση είναι: *vector*! Πράγματι, αυτό το περιέχον φαίνεται να υπερέχει σε όλες τις περιπτώσεις αλλά και τα άλλα έχουν λόγο ύπαρξης.

- Διάλεξε το *deque* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στα δύο άκρα.
- Διάλεξε το *list* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στη «μέση» (μακριά από τα άκρα).

26.3 Συνειρμικά Περιέχοντα

Συνειρμική μνήμη (associative memory) ή **μνήμη προσπελάσιμη με το περιεχόμενο** (content-addressable memory) είναι τρόπος οργάνωσης της μνήμης ώστε η πρόσβαση στην κάθε μονάδα του περιεχομένου της να γίνεται όχι με τη διεύθυνσή της αλλά με τμήμα της μονάδας (κλειδί). Έτσι οργανώνουν τα περιεχόμενά τους τα **συνειρμικά περιέχοντα** (associative containers) της STL.

Δες τις επικεφαλίδες των τεσσάρων περιγραμμάτων:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Η πρώτη παράμετρος-κλάση (*Key*) είναι ο τύπος του αντικειμένου-κλειδιού με βάση το οποίο γίνονται οι αναζητήσεις στη συλλογή. Η παράμετρος-κλάση *Compare* είναι ένα συναρτησοειδές που καθορίζει διάταξη "`<`" για αντικείμενα κλάσης *Key*. Αν για την *Key* είναι ορισμένος ο "`<`" μπορείς να μην ορίσεις το συναρτησοειδές και τον ρόλο του να παίξει το *less*`<`*Key*`>`.[9]

Για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "`<`" που είδαμε πιο πάνω– σε ένα **δένδρο δυαδικής αναζήτησης** (binary search tree).[10] Όπως

[9] Είδαμε το περίγραμμα *"less"* στην §25.6.3.

[10] Συνήθως ένα δένδρο red-black.

κατάλαβαίνεις, παρ' όλο που πραγματευόμαστε –γενικώς– σύνολα και πολυσύνολα, συζητουμε για **διαταγμένα** (ordered) σύνολα και πολυσύνολα.[11]

Κάθε στιγμιότυπο οποιουδήποτε από τα παραπάνω περιγράμματα έχει τη δική του κλάση (αμφίδρομων) προσεγγιστών.

Πέρα από τις μεθόδους που έχουν όλα τα περιέχοντα της STL, στα συνειρμικά περιέχοντα υπάρχουν:

- Μέθοδος που επιστρέφει το πλήθος των αντικειμένων που έχουν συγκεκριμενο κλειδί.
  ```
  size_type A<K>::count( const K& x ) const;
  ```
  Προφανώς, αν το *A* είναι *set* ή *map* η *count()* μπορεί να επιστρέφει "`0`" ή "`1`".[12]
- Μέθοδοι για εντοπισμό κλειδιού στη συλλογή:
  ```
  A<K>::iterator A<K>::find( const K& x );
  A<K>::const_iterator A<K>::find( const K& x ) const;
  ```
  Αν το κλειδί *x* βρεθεί στη συλλογή επιστρέφεται προσεγγιστής προς αυτό το αντικείμενο. Αλλιώς ο προσεγγιστής δείχνει το "`A<K>::end()`".
- Μέθοδοι για εισαγωγές στοιχείων που θα τις δούμε ξεχωριστά για κάθε περιέχον.
- Μέθοδοι για διαγραφές στοιχείων:
  ```
  void A<K>::erase( A<K>::iterator pos );
  size_type A<K>::erase( const K& x );
  void A<K>::erase( A<K>::iterator first, A<K>::iterator last );
  void A<K>::clear();
  ```
  Η πρώτη διαγράφει το αντικείμενο που δείχνει ο *pos*.

  Η δεύτερη διαγράφει όλα τα αντικείμενα που έχουν κλειδί *x*. Επιστρέφει το πλήθος των αντικειμένων που διαγράφηκαν· το ίδιο που θα επέστρεφε το "`count(x)`" πριν από τη διαγραφή.

  Η τρίτη διαγράφει όλα τα αντικείμενα που τα κλειδιά τους ανήκουν στην περιοχή [*first*, *last*).

  Η τελευταία είναι ισοδύναμη με "`erase( begin(), end() )`" και έχει ως αποτέλεσμα κενό περιέχον.
- Μέθοδοι για εντοπισμό περιοχής κλειδιών με συγκεκριμένη τιμή:
  ```
  A<K>::iterator A<K>::lower_bound( const K& x );
  A<K>::const_iterator A<K>::lower_bound( const K& x ) const;
  ```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο που δεν είναι μικρότερο από *x*. Δες και αυτά που λέγαμε για τη συνάρτηση *lower_bound()* στην §26.1.3. Αν στο περιέχον υπάρχει (-ουν) στοιχείο(-α) με κλειδί *x* τότε ο προσεγγιστής που επιστρέφεται δείχνει το πρώτο (ή το μοναδικό) στοιχείο με τέτοιο κλειδί.
  ```
  A<K>::iterator A<K>::upper_bound( const K& x );
  A<K>::const_iterator A<K>::upper_bound( const K& x ) const;
  ```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο του περιέχοντος που είναι μεγαλύτερο από *x*.
  ```
  pair< A<K>::iterator, A<K>::iterator > A<K>::equal_range( const K& x );
  pair< A<K>::const_iterator, A<K>::const_iterator >
                                        A<K>::equal_range( const K& x ) const;
  ```
  Επιστρέφεται ζεύγος[13] προσεγγιστών *f*, *l* ώστε όλα τα κλειδιά στην [*f*, *l*) να έχουν τιμή *x*. Αν δεν υπάρχει το κλειδί οι δυο προσεγγιστές είναι ίσοι (περιοχή κενή). Αν για ένα

---

[11] Το νέο πρότυπο C++11 προδιαγράφει και *μη-διατεταγμένα* σύνολα/πολυσύνολα.

[12] Γράφουμε "`A<K>`" αλλά μπορεί να είναι και "`A<K,C>`" ή "`A<K,C,T>`". Απλώς όλες οι μέθοδοι έχουν να κάνουν μόνο με το κλειδί.

συγκεκριμένο περιέχον πάρουμε το *lb* (*lower_bound()*), το *ub* (*upper_bound()*) και το ζεύγος *eqr* (*equal_range()*) τότε θα έχουμε *eqr*.*first* == *lb* και *eqr*.*second* == *ub*. Το πρόγραμμα:

```
#include <iostream>
#include <functional>
#include <set>
using namespace std;
int main()
{
   int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
   multiset<unsigned int> ms( ints, ints+11 );
   typedef multiset<unsigned int>::iterator msIterator;

   msIterator lb( ms.lower_bound(17) );
   msIterator ub( ms.upper_bound(17) );
   pair<msIterator, msIterator> eqr( ms.equal_range(17) );
   if ( lb == eqr.first )  cout << "lb == eqr.first" << endl;
   if ( ub == eqr.second )  cout << "ub == eqr.second" << endl;
}
```

δίνει

```
lb == eqr.first
ub == eqr.second
```

- Μέθοδος που δίνει αντίγραφο του αντικείμενου-συναρτησοειδούς που χρησιμοποιεί η κλάση:

  ```
  C A<K,C>::key_comp() const;
  ```

  Για τη μέθοδο αυτή θα συζητήσουμε στη συνέχεια.

Κάθε ένα από τα περιγράμματα έχει τη δική του κλάση *αμφίδρομων* προσεγγιστών για τους οποίους ισχύουν τα εξής:

♦ ***Οι πράξεις διαγραφής δεν ακυρώνουν προσεγγιστές εκτός από αυτούς που δείχνουν στοιχεία που διαγράφονται.***

Στη συνέχεια και για κάθε περίγραμμα θα δούμε τις σχετικές πράξεις εισαγωγής (*insert()*). Αλλά από τώρα θα τονίσουμε ότι:

♦ ***Οι πράξεις εισαγωγής δεν ακυρώνουν προσεγγιστές.***

26.3.1 Το Περίγραμμα "*set*"

Το περίγραμμα "*set*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

«φιλοξενεί» συλλογές αντικειμένων που είναι **σύνολα** (sets). Αυτό σημαίνει ότι σε κάθε συλλογή το κάθε αντικείμενο είναι *μοναδικό*.

Δεν υπάρχουν μέθοδοι για το *set* πέρα από αυτές που είδαμε για τα συνειρμικά περιέχοντα γενικώς.

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις στο πρόγραμμά σου:

```
#include <set>
```

Με τη:

```
set< T >  s;
```

δηλώνουμε το *s* ως σύνολο με στοιχεία τύπου *T*. Η τιμή εκκίνησης του *s* είναι το ∅. Π.χ. οι

```
   set< int > intSet;
// intSet == { }
   cout << "#intSet == " << intSet.size() << endl;
```

[13] Το περίγραμμα (*std*::)*pair* είναι ίδιο με το *PairT* που είδαμε στην §25.3.1.

```
   if ( intSet.empty() )  cout<< "intSet == {}" << endl;
                     else  cout<< "intSet != {}" << endl;
```

θα δώσουν:

```
#intSet == 0
intSet == {}
```

- Το "`0`" είναι ο **πληθάριθμος** (cardinality) του *intSet* και μας το δίνει η *intSet*`.size()`.
- Η *intSet*`.empty()` δίνει `true` που σημαίνει «Ναι, το *intSet* είναι κενό.»

  Με την *intSet*`.clear()` δίνεις τιμή ∅ στο *intSet*.

Μπορείς να δώσεις ως αρχική τιμή μια υποπεριοχή είτε από συνήθη πίνακα είτε από κάποιο άλλο περιέχον της STL:

```
   int            intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   vector< int >  intVec( intArr+3, intArr+7 );
// . . .
   set< int > intSet1( intVec.begin(), intVec.end() );
   copy( intSet1.begin(), intSet1.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
   set< int > intSet2( intArr, intArr+5 );
   copy( intSet2.begin(), intSet2.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
-11 8 31 41
-23 -11 8 34 72
```

Πρόσεξε ότι τα στοιχεία των συνόλων μας δίνονται ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Στο θέμα αυτό θα επανέλθουμε στη συνέχεια.

Ο έλεγχος για να δούμε αν μια τιμή x ανήκει σε ένα σύνολο s ($x \in s$) μπορεί να γίνει με τρεις τρόπους:

- Με τη μέθοδο *lower_bound()*. Η "`*(s.lower_bound(x)) == x`" θα επιστρέψει `true` αν η τιμή x υπάρχει στο s. Οι

```
   if ( *(intSet1.lower_bound(31)) == 31 )
      cout << "31 in intSet1" << endl;
   else
      cout << "31 not in intSet1" << endl;
   if ( *(intSet1.lower_bound(61)) == 61 )
      cout << "61 in intSet1" << endl;
   else
      cout << "61 not in intSet1" << endl;
```

  θα δώσουν:

```
31 in intSet1
61 not in intSet1
```

- Με τη μέθοδο *find()*. Η "`s.find(x)`" θα επιστρέψει "`s.end()`" αν η τιμή x δεν υπάρχει στο s. Αλλιώς θα επιστρέψει προσεγγιστή *it*, τύπου `set<int> ::iterator` (το "`int`" για το παράδειγμά μας), τέτοιον ώστε `*`*it* `==` x. Τα ίδια αποτελέσματα με αυτά που πήραμε παραπάνω θα δώσουν και οι

```
   if ( intSet1.find(31)==intSet1.end() )
      cout << "31 not in intSet1" << endl;
   else
      cout << "31 in intSet1" << endl;
   if ( intSet1.find(61)==intSet1.end() )
      cout << "61 not in intSet1" << endl;
   else
      cout << "61 in intSet1" << endl;
```

- Με τη μέθοδο *count()*. Η "`s.count(x)`" θα επιστρέψει "`0`" αν η τιμή x δεν υπάρχει στο s· αλλιώς θα επιστρέψει "`1`". Οι

```
   if ( intSet1.count(31)==0 )  cout << "31 not in intSet1" << endl;
```

```
                          else  cout << "31 in intSet1" << endl;
   if ( intSet1.count(61)==0 )  cout << "61 not in intSet1" << endl;
                          else  cout << "61 in intSet1" << endl;
```

θα δώσουν τα ίδια αποτελέσματα.

Αν θέλεις μπορείς να δοκιμάσεις τα παραπάνω και με το στιγμιότυπο "`set<Date>`"· χρησιμοποίησε τον πίνακα "`dates`" που είδαμε στο παράδειγμα για την ταξινόμηση περιεχομένων λίστας.

Για να χειρίζεσαι ένα σύνολο έχεις ακόμη:

- Τις μεθόδους *erase()* –που είδαμε για τα συνειρμικά περιέχοντα– και επιτρέπουν τη διαγραφή στοιχείων από σύνολο
- Μεθόδους *insert()* που επιτρέπουν εισαγωγή στοιχείων σε σύνολο κλάσης `set<K>`:

```
pair< set<K>::iterator, bool > insert( const K& x );
set<K>::iterator insert( set<K>::iterator position, const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Η πρώτη μορφή επιστρέφει ζεύγος (*pair*) του οποίου

- Το μέλος *first* είναι τύπου `set<K>::iterator` και δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.
- Το μέλος *second* είναι τύπου `bool` και έχει τιμή `true` αν έγινε η εισαγωγή ή `false` αν δεν έγινε εισαγωγή διότι η τιμή υπήρχε.

Η δεύτερη μορφή είναι λίγο «περίεργη»: στην "`insert(it,v)`" με την πρώτη παράμετρο *it* κάνουμε μια υπόδειξη για το *πού* πρέπει να γίνει η εισαγωγή της *v*! Βεβαίως η εισαγωγή θα γίνει εκεί που πρέπει να γίνει αλλά αν ο *it* δείχνει *ακριβώς πριν*[14] από τη θέση που θα γίνει η εισαγωγή κερδίζουμε χρόνο. Επιστρέφει τιμή τύπου `set<K>::iterator` που δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.

Με την τρίτη μορφή εισάγουμε στο σύνολο όλες τις τιμές μιας περιοχής [*first, last*).

Από τις εντολές:

```
   set<int> intSet;
   pair< set<int>::iterator, bool > retVal( intSet.insert(17) );
   cout << boolalpha
        << *(retVal.first) << "  " << retVal.second << endl;
```

παίρνουμε:

```
17  true
```

Ο προσεγγιστής `retVal.first` δείχνει το στοιχείο του *intSet* που έχει τιμή "`17`". Η τιμή `true` του `retVal.second` μας λέει ότι η εισαγωγή αυτής της τιμής έγινε τώρα.

Αν

```
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
```

από τις εντολές:

```
   intSet.insert( intArr, intArr+4 );
   copy( intSet.begin(), intSet.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
```

παίρνουμε:

```
-23 8 17 34 72
```

Μετά από αυτές δίνουμε:

```
   retVal = intSet.insert( 8 );
   cout << *(retVal.first) << "  " << retVal.second << endl;
```

και παίρνουμε:

---

[14] Με το νέο πρότυπο C++11 έχουμε βέλτιστη ταχύτητα αν ο *it* δείχνει *ακριβώς μετά* τη θέση που θα γίνει η εισαγωγή.

```
8  false
```

Και εδώ ο `retVal.first` δείχνει στοιχείο με την τιμή που εισάγουμε αλλά η `retVal.second` έχει τιμή `false`: το "`8`" υπάρχει ήδη στο *intSet*.

Συνεχίζουμε δίνοντας:

```
set<int>::iterator it;
it = intSet.insert( retVal.first, 11 );
cout << *it << endl;
```

και παίρνουμε:

```
11
```

Δηλαδή: χρησιμοποιώντας τη δεύτερη μορφή της *insert()* εισάγουμε την τιμή "`11`". Ως πρώτο όρισμα βάζουμε τον προσεγγιστή που πήραμε από την εισαγωγή του "`8`". Η αποπαραπομπή του προσεγγιστή που μας επιστρέφεται μας δίνει ακριβώς το "`11`".

Ε, τώρα ας κάνουμε και μερικές διαγραφές:

```
int n( intSet.erase(72) );
cout << n << endl;
copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;
intSet.erase( intSet.begin(), intSet.find(17) );
copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;
```

που δίνουν:

```
1
-23 8 11 17 34
17 34
```

Η "`intSet.erase(72)`" επιστρέφει (και κάνει τιμή της *n*) το πλήθος των στοιχείων που διαγράφηκαν. Το "`1`" της πρώτης γραμμής μας λέει ότι διαγράφηκε ένα στοιχείο· δηλαδή το "`72`" υπήρχε και διαγράφηκε.

Η επόμενη «μαζική» διαγραφή διαγράφει τα στοιχεία του συνόλου από την αρχή μέχρι να βρει το "`17`".

Τέλος, να επισημάνουμε κάτι για τους προσεγγιστές του *set*: Είτε γράψεις "`set<K>::iterator it`" είτε γράψεις "`set<K>::const_iterator it`" *δεν επιτρέπεται να τροποποιήσεις το* `*it`. Γιατί; Διότι *τα περιεχόμενα ενός set είναι κλειδιά και* –όπως έχουμε τονίσει και σε προηγούμενα κεφάλαια– *τα κλειδιά δεν τροποποιούνται*. Αν επιτρεπόταν η τροποποίηση του `*it` θα ήταν πολύ απλό να καταστρέψεις και τη μοναδικότητα των στοιχείων του συνόλου και τη διάταξή τους.[15] Ο «νόμιμος» τρόπος να τροποποιήσεις το `*it` είναι ο εξής:

```
K temp( *it );
oneSet.erase( it );
// άλλαξε την τιμή της temp
oneSet( temp );
```

Αν ο *K* είναι τύπος με μεγάλα αντικείμενα αυτός ο τρόπος γίνεται χρονοβόρος. Για την περίπτωση αυτήν υπάρχει άλλο περιέχον για παράσταση συνόλου: το "*map*" που θα δούμε στη συνέχεια.

26.3.1.1 Σχέσεις και Πράξεις Συνόλων

Η σχέση "$\subseteq$" («είναι υποσύνολο», «περιλαμβάνεται») καθώς και οι πράξεις "$\cup$" (ένωση), "$\cap$" (τομή), "$\setminus$" (διαφορά) και "Δ" (συμμετρική διαφορά) υλοποιούνται με περιγράμματα συναρτήσεων. Για να τα χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <algorithm>`".

---

[15] Ο μεταγλωττιστής Borland C++ v.5.5 *επιτρέπει* τροποποίηση του `*it`! Αν τον έχεις δοκίμασε να κάνεις τις «παρανομίες» που λέμε.

Η σχέση $A \subseteq B$ (διαβάζεται ως «B includes A») υλοποιείται με το περίγραμμα *includes()* που ορίζεται στο **algorithm**:

```
template< class InputIter1, class InputIter2 >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2 );
template< class InputIter1, class InputIter2,
          class Compare >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2,
               Compare comp);
```

Με τις:

```
#include <set>
#include <algorithm>
// . . .
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
// . . .
   set<int> intSet1( intArr, intArr+3 );
   copy( intSet1.begin(), intSet1.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   set<int> intSet2( intArr, intArr+7 );
   copy( intSet2.begin(), intSet2.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intSet1.begin(), intSet1.end()) )
      cout << "intSet2 includes intSet1" << endl;
   else
      cout << "intSet2 does not include intSet1" << endl;
```

θα πάρουμε:

```
-23 34 72
-23 -11 8 31 34 41 72
intSet2 includes intSet1
```

Η σχέση $A \subset B$ –**γνήσιο υποσύνολο** (proper ή strict subset)– δεν υλοποιείται απ' ευθείας· θα πρέπει να τη δεις ως $(A \subseteq B)$ **&&** $(A$ **!=** $B)$.

Εδώ όμως θα πρέπει να παρατηρήσουμε μερικά πράγματα:

- Οι τύποι των παραμέτρων βάζουν πολύ λίγους περιορισμούς: είναι απλώς *προσεγγιστές εισόδου*, δηλαδή ό,τι πιο γενικό. Πουθενά δεν φαίνεται ότι θα πρέπει να είναι προσεγγιστές συνόλου. Στην πραγματικότητα γίνεται σύγκριση των περιεχομένων δύο περιοχών που μπορεί να βρίσκονται σε οποιοδήποτε περιέχον STL, σε πίνακα ή ακόμη και στα δεδομένα εισόδου! Πράγματι, αν γράψουμε:

```
// . . .
   istream_iterator< int > cinIt( cin ), cinEoStream;
   if ( includes(intSet2.begin(), intSet2.end(),
                 cinIt, cinEoStream) )
      cout << "intSet2 includes input" << endl;
// . . .
```

και κατά την εκτέλεση πληκτρολογίσουμε:

```
-11 8 31 e
```

θα πάρουμε:

```
intSet2 includes input
```

Αν γράψουμε:

```
// . . .
   copy( intArr+4, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intArr+4, intArr+7) )
      cout << "intSet2 includes [intArr+4, intArr+7)" << endl;
```

```
// . . .
```

θα πάρουμε:

```
-11 31 41
intSet2 includes [intArr+4, intArr+7)
```

Τέλος, αν γράψουμε:

```
deque<int> intDeque( intArr+1, intArr+6 );
sort( intDeque.begin(), intDeque.end() );
copy( intDeque.begin(), intDeque.end(),
      ostream_iterator<int>(cout, " ") );
cout << endl;
if ( includes(intSet2.begin(), intSet2.end(),
              intDeque.begin(), intDeque.end()) )
   cout << "intSet2 includes intDeque" << endl;
```

παίρνουμε:

```
-23 -11 8 31 72
intSet2 includes intDeque
```

- Πρόσεξε την τελευταία περίπτωση: πριν καλέσουμε την *includes()* καλούμε τη *sort()* για να ταξινομήσει τα περιεχόμενα της *intDeque*. Αυτό είναι ουσιώδες: Αν δεν το κάνουμε θα πάρουμε λάθος αποτελέσματα:

```
72 -23 8 -11 31
intSet2 does not include intDeque
```

Για να καταλάβεις τι γίνεται αντιγράφουμε –«ξεκαθαρίζοντας» κάπως– το περίγραμμα *includes()* από τη gcc (αρχείο `stl_algo.h`):[16]

```
 0: template< typename InputIter1, typename InputIter2 >
 1: bool includes( InputIter1 first1, InputIter1 last1,
 2:                InputIter2 first2, InputIter2 last2 )
 3: {
 4:    while( first1 != last1 && first2 != last2 )
 5:       if( *first2 < *first1 )
 6:          return false;
 7:       else if( *first1 < *first2 )
 8:          ++first1;
 9:       else
10:       {  ++first1; ++first2;  }
11:
12:    return first2 == last2;
13: }
```

Όπως καταλαβαίνεις, το πρόβλημα είναι η σύγκριση της γραμμής 5: "`34 < 72`" και δεν υπάρχει αν κάνουμε την ταξινόμηση.

Στη δεύτερη μορφή του περιγράμματος η γραμμή 5 γίνεται:

```
if( comp(*first2,*first1) )
```

και η γραμμή 7:

```
else if( comp(*first1,*first2) )
```

Προϋποθέσεις για τον αλγόριθμο *includes()*: Οι δύο περιοχές

- Περιέχουν στοιχεία του ίδιου τύπου.[17]
- Τα στοιχεία είναι ταξινομημένα.
- Η ταξινόμηση έγινε με το ίδιο κατηγόρημα *comp()*.

Τα παραπάνω ισχύουν και για τις πράξεις μεταξύ συνόλων που βλέπουμε στη συνέχεια. Αν βάλεις στο πρόγραμμά σου "`#include <algorithm>`" μπορείς να χρησιμοποιείς και τις:

---

[16] Κάτι σου θυμίζει; Ξαναδές τον αλγόριθμο της συγχώνευσης (§9.5.3).

[17] Και τα στοιχεία είναι μοναδικά για την καθεμιά; Η αλήθεια είναι ότι αλγόριθμος δουλεύει και χωρίς τη μοναδικότητα, π.χ. και για πολυσύνολα.

- (*std*::)*set_union* (ένωση συνόλων),
- (*std*::)*set_intersection* (τομή συνόλων),
- (*std*::)*set_difference* (διαφορά συνόλων) και
- (*std*::)*set_symmetric_difference* (συμμετρική διαφορά συνόλων).[18]

Με αυτές όμως έχουμε και ένα άλλο πρόβλημα. Ας πάρουμε, για παράδειγμα, το:

```
template< class InputIter1, class InputIter2, class OutputIterator >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result );
template< class InputIter1, class InputIter2,
          class OutputIterator, class Compare >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result, Compare comp );
```

Το νέο στοιχείο εδώ είναι ότι έχουμε *αποτέλεσμα* που είναι (κατ' αρχήν) *σύνολο*. Γιατί «κατ' αρχήν»; Διότι ναι μεν είναι σύνολο αλλά μπορείς να το αποθηκεύσεις όπου σε βολεύει.

Με την παράμετρο *result* περνούμε την *αρχή* της περιοχής όπου θα αποθηκευτεί το απότέλεσμα. Η συνάρτηση επιστρέφει τιμή-προσεγγιστή προς το *τέλος* της περιοχής όπου έγινε η αποθήκευση. *Δεν πρέπει να υπάρχει επικάλυψη της περιοχής που θα πάρει το αποτέλεσμα με οποιαδήποτε από τις περιοχές που ενώνονται.*

Ας πούμε ότι έχουμε δύο σύνολα:

```
set< int > intSet1, intSet2;
```

και τους δίνουμε τιμές { 4, 5, 6 } και { 2, 4, 6, 8, 10 } αντιστοίχως. Μπορούμε να πάρουμε την ένωσή τους κατ' ευθείαν στο *cout*:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           ostream_iterator<int>(cout, " ") );
cout << endl;
```

και να δούμε στην οθόνη μας:

```
2 4 5 6 8 10
```

Μπορούμε να πάρουμε την ένωσή τους στο:

```
vector<int> intVec1(11);
```

με την εντολή:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           intVec1.begin() );
copy( intVec1.begin(), intVec1.end(),
      ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
2 4 5 6 8 10 0 0 0 0 0
```

Τι είναι αυτά τα μηδενικά; Για τον πίνακα έχουμε κρατήσει 11 θέσεις ενώ η ένωση έχει μόνον 6. Δηλαδή αν δηλώσουμε:

```
vector<int> intVec1;
```

δεν θα βγούνε; Δεν θα βγούνε αλλά θα πρέπει να χρησιμοποιήσουμε τον προσαρμογέα που μάθαμε στην §26.1.3:

```
back_insert_iterator< vector<int> > destV( intVec1 );
```

και να δώσουμε:

---

[18] Η **συμμετρική διαφορά** δύο συνόλων A, B ορίζεται ως $A \Delta B = (A \setminus B) \cup (B \setminus A)$· δηλαδή τα στοιχεία του A που δεν ανήκουν στο B και τα στοιχεία του B που δεν ανήκουν στο A. Άρα: $A \Delta B = (A \cup B) \setminus (A \cap B)$.

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           destV );
```

Και αν θέλουμε να αποθηκεύσουμε το αποτέλεσμα σε ένα

```
set<int> intSet3;
```

θα κάνουμε το ίδιο; Όχι, αλλά κάτι παρόμοιο: Το περίγραμμα *back_insert_iterator* μπορεί να δώσει στιγμιότυπα για κλάσεις που έχουν την *push_back()*. Ένα άλλο περίγραμμα-προσαρμογέας που μπορεί να δώσει στιγμιότυπα για κάθε κλάση που έχει την *insert()* είναι ο *inserter*. Για να τον χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <iterator>`".

Αν δώσουμε:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           inserter(intSet3, intSet3.begin()) );
```

η ένωση θα φυλαχθεί στο *intSet3*.

Με τον ίδιο τρόπο δηλώνονται και οι άλλες συναρτήσεις (περιγράμματα) για σύνολα. Για παράδειγμα, για την τομή έχουμε:

```
template< class InputIter1, class InputIter2, class OutputIter >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result );
template< class InputIter1, class InputIter2,
          class OutputIter, class Compare >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result, Compare comp );
```

Ο χειρισμός τους γίνεται με τον ίδιο τρόπο.

26.3.2 Το Περίγραμμα "*map*"

Το περίγραμμα "*set*" δεν είναι το μοναδικό εργαλείο της STL για παράσταση συνόλου. Υπάρχει και η **απεικόνιση** (map) που έχει την επικεφαλίδα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου

```
#include <map>
```

Σε σύγκριση με το "*set*" υπάρχει μια επί πλέον παράμετρος: "`class T`". Τι είναι αυτό; Τα περιεχόμενα του περιγράμματος είναι οργανωμένα σε ζεύγη τύπου:

```
pair< Key, T >
```

Ο πρώτος τύπος (*Key*) είναι ο γνωστός μας (από τα σύνολα) *τύπος κλειδιών* (key type)· ο δεύτερος (*T*) είναι ο **τύπος τιμών** (value type). Έτσι, οι δυο συνιστώσες του ζεύγους είναι:

```
Key first;
T   second;
```

Μήπως το "`map<Key, T>`" είναι ίδιο με το "`set< pair<Key, T> >`"; Όχι! Όταν το ζεύγος μπαίνει στο περιέχον (`map`) η θέση του στη διάταξη καθορίζεται από την τιμή του *first* που μπαίνει στο δένδρο αναζήτησης και *δεν* (πρέπει να) *αλλάζει*.[19] Στο *second* βάζουμε όλα τα *άλλα στοιχεία που έχουμε τη δυνατότητα να τροποποιούμε*. Στο "`set< pair<Key, T> >`" δεν έχουμε δυνατότητα για οποιαδήποτε τροποποίηση.

[19] Το «πρέπει να» για τους μεταγλωττιστές –σαν τον Borland C++ v.5.5– που αφήνουν την περιφρούρηση της δομής του συνόλου αποκλειστικώς στον προγραμματιστή.

Παράδειγμα

Γυρνάμε πίσω, στο Project 4, στην κλάση *CourseCollection* και αναγνωρίζουμε ότι έχουμε ένα σύνολο αντικειμένων *Course*. Θα πρέπει να αντικαταστήσουμε τη δήλωση "`Course* ccArr`" με μια δήλωση *map* αφού κάνουμε την κατάλληλη προεργασία.

- Ο τύπος κλειδιών είναι έτοιμος:

```
struct CourseKey
{
   enum { cCodeSz = 8 };
   char s[cCodeSz];
   explicit CourseKey( string aKey="" )
   {  strncpy( s, aKey.c_str(), cCodeSz-1 );
      s[cCodeSz-1] = '\0';  }
}; // CourseKey
```

- Όλα τα υπόλοιπα μπαίνουν στον τύπο τιμών:

```
class CourseVal
{
public:
   enum { cTitleSz = 80, cCategSz = 4 };
   explicit CourseVal( string aTitle="" );
// . . .
   void display( ostream& tout ) const;
private:
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
// . . .
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
}; // Course
```

- Ο τύπος των ζευγών τιμών θα είναι:

```
   pair< CourseKey, CourseVal >
```

- Αφού για τον *CourseKey* δεν έχουμε επιφορτώσει τον "`<`" θα πρέπει να γράψουμε το κατάλληλο συναρτησοειδές:

```
struct CodeLT
{
   bool operator()( const CourseKey& ck1, const CourseKey& ck2 )
   {  return ( strcmp(ck1.s, ck2.s) < 0 );  }
}; // CodeLT
```

Έτσι, θα δηλώσουμε το σύνολό μας ως:

```
   map< CourseKey, CourseVal, CodeLT >  ccCourses;
```

Αν έχουμε:

```
   CourseKey aCk( "EY02010" );
   CourseVal aCv( "Αντικειμενοστρεφής Προγραμματισμός (θ)" );
```

μπορούμε να εισαγάγουμε στο *ccCourses*:

```
   ccCourses.insert( make_pair(aCk, aCv) );
```

Αν

```
   map<CourseKey, CourseVal, CodeLT>::iterator it;
```

και ο *it* δείχνει το αντικείμενο που βάλαμε μπορούμε να κάνουμε τροποποίηση του *ccCateg*:

```
   (it->second).setCateg( "MEY" );
```

Ο τελεστής "`[]`" –που δεν επιφορτώσαμε για την *CourseCollection* διότι μας φάνηκε «παράξενο» το "`allCourses["EY02010"]`"– επιφορτώνεται για το *map*. Δες το παρακάτω

Παράδειγμα

Στο παρακάτω πρόγραμμα έχουμε μια απεικόνιση από τα ονόματα των μηνών στα πλήθη ημερών τους.

```
  0: #include <iostream>
  1: #include <string>
```

```
 2: #include <map>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:    string monthN[] = { "Jan", "Feb", "Mar", "Apr",
 9:                        "May", "Jun", "Jul", "Aug",
10:                        "Sep", "Oct", "Nov", "Dec" };
11:    unsigned int noOfDays[] = { 31, 28, 31, 30, 31, 30,
12:                                31, 31, 30, 31, 30, 31 };
13:    map< string, unsigned int > months;
14:
15:    for ( int k(0); k < 6; ++k )
16:    {
17:       pair<string, unsigned int> amp;
18:       amp.first = monthN[k];
19:       amp.second = noOfDays[k];
20:       months.insert( amp );
21:    }
22:    for ( int k(6); k < 12; ++k )
23:       months[monthN[k]] = noOfDays[k];
24:
25:    for ( map<string, unsigned int>::const_iterator
26:                                          cit( months.begin() );
27:          cit != months.end(); ++cit )
28:       cout << cit->first << ", " << cit->second << endl;
29: }
```

Στις γραμμές 15-21 εισάγουμε τους πρώτους 6 μήνες δημιουργώντας τα κατάλληλα ζεύγη `pair<string, unsigned int>` και χρησιμοποιώντας τη μέθοδο *insert()*.

Στις γραμμές 22-23 εισάγουμε τους τελευταίους 6 μήνες αλλιώς:

```
   months["Jul"] = 31;
   months["Aug"] = 31;
// . . .
   months["Dec"] = 31;
```

Το "`months["Jul"] = 31`" είναι ισοδύναμο με "`months.insert(make_pair ("Jul",31))`".

Γράφουμε στο *cout* το περιεχόμενο στηριγμένοι στο ότι ο προσεγγιστής *cit* δείχνει ζεύγος που έχει ως *first* το όνομα του μήνα και ως *second* το πλήθος των ημερών. Το πρόγραμμα θα δώσει:

```
Apr, 30
Aug, 31
Dec, 31
Feb, 28
Jan, 31
Jul, 31
Jun, 30
Mar, 31
May, 31
Nov, 30
Oct, 31
Sep, 30
```

Οι μήνες βγαίνουν κατ' αλφαβητική σειρά. Αν αντικαταστήσουμε τις γραμμές 25-28 με τις:

```
   for ( int k(0); k < 12; ++k )
      cout << monthN[k] << ", " << months[monthN[k]] << endl;
```

όπου χρησιμοποιούμε τον "`[]`", θα πάρουμε:

```
Jan, 31
Feb, 28
Mar, 31
Apr, 30
```

```
May, 31
Jun, 30
Jul, 31
Aug, 31
Sep, 30
Oct, 31
Nov, 30
Dec, 31
```

Εκτός από την *insert()* που είδαμε στα δύο παραδείγματα υπάρχουμε και για το *map* οι άλλες δύο μορφές που είδαμε για το *set*.

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν για τα στιγμιότυπα του *map* όπως για τα στιγμιότυπα του *set*.

26.3.3 Τα Περιγράμματα "*multiset*" και "*multimap*"

Βάζοντας στο πρόγραμμά σου την "`#include <set>`" έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα "*multiset*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
```

Κάθε στιγμιότυπο του περιγράμματος έχει αντικείμενα που είναι **πολυσύνολα** (multisets, bags). Σε αντιδιαστολή με το σύνολο, ένα πολυσύνολο μπορεί να έχει πολλαπλά αντίγραφα των μελών του. Έτσι, ενώ η σχέση "*ανήκει*" ορίζεται και για πολυσύνολα ($x \in m$)[20], πιο πολύ νόημα έχει η συνάρτηση **πολλαπλότητα** (multiplicity, multitude) που στο *multiset* υλοποιείται με την *count()*.

Όπως στα σύνολα έτσι και εδώ με την:

```
   multiset< int >  mulSet1;
```

δημιουργούμε ένα κενό πολυσύνολο ($[\![\ ]\!]$). Οι εντολές

```
   cout << "#mulSet1 == " << mulSet1.size() << endl;
   if ( mulSet1.empty() )  cout<< "mulSet1 == [[]]" << endl;
                     else  cout<< "mulSet1 != [[]]" << endl;
```

δίνουν:

```
#mulSet1 == 0
mulSet1 == [[]]
```

Θέλοντας να παραστήσουμε το $[\![\ 1, 2, 4, 3, 2, 4, 4\ ]\!]$ στο πρόγραμμά μας δίνουμε:

```
   int intArr[] = { 1, 2, 4, 3, 2, 4, 4 };

   multiset< int >  mulSet2( intArr, intArr+7 );
   copy( mulSet2.begin(), mulSet2.end(),
                               ostream_iterator<int>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
1 2 2 3 4 4 4
```

Για να δούμε τις πολλαπλότητες των "`2`" και "`7`" στο *mulSet2* δίνουμε:

```
   cout << "mulSet2 # 2 = " << mulSet2.count(2) << endl;
   cout << "mulSet2 # 7 = " << mulSet2.count(7) << endl;
```

και παίρνουμε:

```
mulSet2 # 2 = 2
mulSet2 # 7 = 0
```

Για το *multiset* υπάρχουν τρεις μέθοδοι *insert()*:

---

[20] Συμβολισμός της γλώσσας προδιαγραφών Z.

```
multiset<K>::iterator multiset<K>::insert( const K& x );
multiset<K>::iterator multiset<K>::insert( multiset<K>::iterator pos,
                                           const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Με την πρώτη μορφή εισάγουμε στο πολυσύνολο την τιμή *x*. Αν η τιμή υπάρχει ήδη θα αυξηθεί η πολλαπλότητά της κατά 1. Επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο.

Και με τη δεύτερη μορφή εισάγουμε στο πολυσύνολο την τιμή *x* και μας επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο. Η πρώτη παράμετρος (*pos*) μπορεί να επιταχύνει την εισαγωγή, όπως λέγαμε και στα σύνολα.

Με την τρίτη μορφή εισάγουμε όλα τα στοιχεία της περιοχής [*first*, *last*).

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν και για το *multiset*.

Παρατηρήσεις:▶

1. Ως *ένωση* (*U*) και *τομή* (*I*) δύο πολυσυνόλων M_1 και M_2 ορίζονται τα πολυσύνολα που έχουν $U \# v = max(M_1 \# v, M_2 \# v)$ και $I \# v = min(M_1 \# v, M_2 \# v)$.

Στη βιβλιογραφία (π.χ. στη γλώσσα Z) θα βρεις να αναφέρεται ως «ένωση» και αυτό που άλλοι ονομάζουν «άθροισμα πολυσυνόλων» (bag sum) και έχει $S \# v = M_1 \# v + M_2 \# v$.

2. Ως *διαφορά* (*D*) των πολυσυνόλων $M_1 \setminus M_2$ ορίζεται το πολυσύνολο που έχει $D \# v = (M_1 \# v > M_2 \# v)$? $M_1 \# v - M_2 \# v$: 0.◀

Με την "`#include <map>`" στο πρόγραμμά σου έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Όπως στο "*map*" έτσι και στο "*multimap*" τα περιεχόμενα είναι οργανωμένα σε ζεύγη τύπου:

```
pair< Key, T >
```

26.3.4 Διάταξη Στοιχείων

Τα στοιχεία ενός συνόλου (ή ενός πολυσυνόλου) δεν είναι κατ' ανάγκη διαταγμένα. Αλλά, όπως θα παρατήρησες σε όλα τα παραδείγματα που δώσαμε τα στοιχεία συνόλων και πολυσυνόλων βγαίνουν ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Γιατί; Διότι «*για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "<" που είδαμε πιο πάνω– σε ένα δένδρο δυαδικής αναζήτησης.*» Έτσι, όταν δηλώνεις σύνολο με στοιχεία τύπου *K* θα πρέπει στον τύπο αυτόν να είναι ορισμένη η διάταξη "`<`".

Πάντως είναι δυνατόν να ανατρέψεις αυτήν τη διάταξη ορίζοντας δική σου όπως θα δεις στο παρακάτω παράδειγμα:

```
int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };

set< int >  intSet( intArr, intArr+7 );
copy( intSet.begin(), intSet.end(),
                            ostream_iterator<int>(cout, " ") );
cout << endl;

set< int, greater<int> > intSetG( intArr, intArr+7 );
copy( intSetG.begin(), intSetG.end(),
                            ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
-23 -11 8 31 34 41 72
72 41 34 31 8 -11 -23
```

Το δεύτερο σύνολο μας δίνει τα μέλη του κατά φθίνουσα τάξη. Αυτό είναι αποτέλεσμα του συναρτησοειδούς που έχουμε ως δεύτερο όρισμα (`greater <int>`) στο στιγμιότυπο του περιγράμματος.

Για παράδειγμα δες πώς ορίζεται (στο `functional`) το περίγραμμα

```
template < class T >
struct greater : public binary_function<T, T, bool>
{
   bool operator() (const T& x, const T& y) const { return x > y; }
};
```

Μέσα στο πρόγραμμά σου μπορείς να γράψεις: `greater(5,2)` ή `greater(0.5, 1.2)` και να σου δώσουν τιμές `true` και `false` αντιστοίχως. Φυσικά, για να γίνει η εξειδίκευση θα πρέπει να ορίζεται ο τελεστής "`>`" για τον τύπο *T*.

Αν δεν ορίζεται ο τελεστής διάταξης που σε ενδιαφέρει τι κάνεις; Έχεις δύο επιλογές:

- Να τον επιφορτώσεις (αν φυσικά επιφορτώνεται).
- Να γράψεις το δικό σου συναρτησοειδές.

Για τη δεύτερη περίπτωση δες το παρακάτω παράδειγμα:

```
struct cstrGT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) > 0;
   }
};

int main()
{
   set<const char*, cstrGT> css;
   css.insert( "μια" );  css.insert( "κλάση" );
   css.insert( "στην" );  css.insert( "οποία" );
   css.insert( "επιφορτώνουμε" );  css.insert( "τον" );
   css.insert( "τελεστή" );  css.insert( "κλήσης" );
   css.insert( "συνάρτησης" );
   copy( css.begin(), css.end(),
                        ostream_iterator<const char*>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
τον τελεστή συνάρτησης στην οποία μια κλήσης κλάση επιφορτώνουμε
```

Αν όμως ορίσουμε:

```
struct cstrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) < 0;
   }
};
```

δηλώσουμε

```
   set<const char*, cstrLT> css;
```

και εισαγάγουμε τις ίδιες λέξεις παίρνουμε:

```
επιφορτώνουμε κλάση κλήσης μια οποία στην συνάρτησης τελεστή τον
```

Δηλώνοντας:

```
set< T > s;
```

είναι σαν να δηλώνεις

```
set< T, less<T> > s;
```

με την προϋπόθεση ότι στον *T* είναι ορισμένος ο "`<`".

Αυτά που λέμε εδώ για τα στιγμιότυπα του "*set*" εφαρμόζονται και στα άλλα συνειρμικά περιέχοντα και όχι μόνον· τα είδαμε και στα περιγράμματα *min()*, *max()*, *sort()* κλπ. Δηλαδή, εκείνο το *comp()* είναι σαν τον "`<`"; Πρόσεξε:

- Όταν γράφουμε "`a < b`" εννοούμε ότι "το *a* είναι μικρότερο του *b*". Εννοούμε όμως και κάτι άλλο: αν έχουμε αύξουσα διάταξη το *a προηγείται του b*.
- Η παράσταση "`comp(a, b)`" έχει τιμή `true` αν και μόνον αν το *a προηγείται του b*. Δηλαδή το *comp()* καθορίζει τη διάταξη· το «μικρότερο» το ξεχνούμε.

Από την επιφόρτωση του "`<`" ή τον ορισμό του *comp()* έχουμε και υποκατάστατο του "`==`": "`!(a<b) && !(b<a)`" ή "`!comp(a,b) && !comp(b,a)`". Στην περίπτωση αυτή λέμε ότι τα κλειδιά *a* και *b* είναι **ισοδύναμα** (equivalent) κρατώντας τον όρο «*ίσα*» μόνο για την περίπτωση σύγκρισης με τον "`==`" (αν είναι ορισμένος).

Στις παρατηρήσεις για τη *lower_bound()* στην §26.1.3 γράφαμε «*Η comp πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.*» Γενικώς:

- ***Αν μια συνάρτηση χρησιμοποιεί την ταξινόμηση των στοιχείων ενός περιέχοντος το συναρτησοειδές (comp ή less<T>) διάταξης θα πρέπει να είναι το ίδιο με αυτό που έγινε η ταξινόμηση.***

26.4 Ποιο Περιέχον να Διαλέξω;

Μετά την ερώτηση «*Ποια Ακολουθία να Διαλέξω;*» έρχεται η γενικότερη ερώτηση: «*Ποιο Περιέχον να Διαλέξω;*»

- Για πολλούς η απάντηση είναι η ίδια με αυτήν που θα έδιναν και στην πρώτη ερώτηση: «*vector και πάλι vector! Δεν νομίζω ότι υπάρχει κάτι άλλο!*»
- Άλλοι θα πουν: όπως λέγαμε στην §9.5.1 «*Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου.*» Θα πρέπει λοιπόν να χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα των περιεχομένων είναι κλειδιά ή
  - το "*map*".

Η δεύτερη άποψη φαίνεται πιο σωστή αφού έχουμε πιο εύκολη εισαγωγή και διαγραφή στοιχείων· θυμίσου ότι στο σημείο αυτό το "*vector*" υστερεί και ως προς το "*list*". Ακόμη, οι αναζητήσεις στα συνειρμικά περιέχοντα έχουν λογαριθμική πολυπλοκότητα.

Βέβαια και στο "*vector*" μπορείς να έχεις γρήγορες αναζητήσεις αν έχεις ταξινομήσει τα περιεχόμενά του. Ακόμη, το "*vector*" έχει το πλεονέκτημα της αποθήκευσης των στοιχείων του σε συναπτές θέσεις της μνήμης πράγμα που επιτρέπει –εκτός από τη δυαδική αναζήτηση– τη χρήση συναρτήσεων που έχουν γραφεί για απλούς πίνακες. Το μειονέκτημά του είναι η χρονοβόρα εισαγωγή/διαγραφή στοιχείων έτσι ώστε να μην καταστρέφεται η ταξινόμηση.

Ας προσπαθήσουμε να βγάλουμε μερικούς κανόνες για τις επιλογές μας.

Ένας κάπως επιφανειακός αλλά όχι λανθασμένος κανόνας είναι αυτός που είπαμε πιο πάνω:

- Αν έχουμε μια συλλογή αντικειμένων που έχει χαρακτηριστικά συνόλου (μοναδικότητα στοιχείων) χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα είναι κλειδιά ή
  - το "*map*".
- Αν η συλλογή μας είναι πολυσύνολο τότε
  - αν έχουμε συχνές εισαγωγές/διαγραφές χρησιμοποιούμε "*multiset*" ή "*multimap*",
  - αλλιώς χρησιμοποιούμε (ταξινομημένο) "*vector*".

Αν σε ενδιαφέρει πολύ η ταχύτητα θα πρέπει να εξετάσεις την περίπτωση του ταξινομημένου "*vector*" και σε σχέση με τα "*set*"/"*map*". Φυσικά, σε μια τέτοια περίπτωση έχεις να πληρώσεις και κάτι ακόμη: είναι δική σου υποχρέωση να διασφαλίζεις τη μοναδικότητα των μελών του συνόλου.

26.5 Άλλα Περιγράμματα

Στη συνέχεια θα δούμε εν συντομία δύο περιγράμματα κλάσεων που δεν είναι περιέχοντα: *bitset, complex*.[21]

26.5.1 Το Περίγραμμα "*bitset*"

Βάζοντας στο πρόγραμμά σου την οδηγία "`#include <bitset>`" μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template < size_t N >
class bitset;
```

που είναι σαν τη δική μας "*Bitmap*" αλλά χωρίς περιορισμούς από το μέγεθος (σε δυαδικά ψηφία) του ακέραιου: το μέγεθος καθορίζεται από την παράμετρο *N* και μπορεί να είναι πολύ μεγάλο.

Έχει κάποια σχέση με το "`vector<bool>`"; Μοιάζει πολύ αλλά και διαφέρει· η βασική διαφορά είναι ότι το "*bitset*"

- δεν είναι δυναμικό –η τιμή της *N* καθορίζεται όταν γίνεται η μεταγλώττιση και μετά δεν αλλάζει–
- δεν έχει προσεγγιστές και φυσικά δεν έχει "*push_back()*".

Με μεθόδους επιφορτώνονται οι τελεστές "`&=`", "`|=`", "`^=`", "`~`", "`[]`", "`<<=`", "`>>=`","`==`", "`!=`" ενώ με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι "`&`", "`|`", "`^`".

Οι "`<<`", "`>>`" επιφορτώνονται δύο φορές

- με μεθόδους ως τελεστές ολίσθησης:

```
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
```

- με περιγράμματα καθολικών συναρτήσεων ως τελεστές εισόδου/εξόδου:

```
template < size_t N >
istream& operator>>( istream& is, bitset<N>& x );
template < size_t N >
ostream& operator<<( ostream& os, const bitset<N>& x );
```

Το περίγραμμα έχει αρκετούς δημιουργούς:

- Ερήμην δημιουργό:

```
bitset();
```

που ξεκινάει όλα τα ψηφία στο "`0`":

```
bitset<12> bsd;
cout << bsd << endl;
```

Αποτέλεσμα:

```
000000000000
```

- Δημιουργό με αρχική τιμή `unsigned long int`:

```
bitset( unsigned long val );
```

Ας τον δούμε με ένα παράδειγμα: οι

```
bitset<12> bsu0( 21UL ), bsu1( 1000000UL );
cout << bsu0 << endl << bsu1 << endl;
```

[21] Δεν θα δούμε το περίγραμμα *valarray* και τα σχετικά με αυτό.

```
bitset<32> bsu2( 1000000UL );
cout << bsu2 << endl;
```

δίνουν:

```
000000010101
001001000000
00000000000011110100001001000000
```

Στην πρώτη γραμμή βλέπουμε την παράσταση του "21" στο δυαδικό σύστημα (16+4+1) με 12 ψηφία (το λιγότερο σημαντικό ψηφίο στα δεξιά).

Στην τρίτη γραμμή βλέπεις το 1 εκατομμύριο σο δυαδικό σύστημα (32 ψηφία). Και στη δεύτερη τι έχουμε; Τα 12 λιγότερο σημαντικά ψηφία αυτού που βλέπουμε στην τρίτη γραμμή· τόσα χωράνε στη *bsu1*!

Καταλαβαίνεις λοιπόν πώς δουλεύει αυτός ο δημιουργός:

- Αν η δυαδική παράσταση του ορίσματος χωράει στα δυαδικά ψηφία που έχουμε (*N*) όλα πάνε καλά (με πιθανή συμπλήρωση με μηδενικά από αριστερά).
- Αν δεν χωράει αποθηκεύονται τα λιγότερο σημαντικά ψηφία που χωράνε.

- Δημιουργό με αρχική τιμή `string`:

```
explicit bitset( const string& str, size_type pos=0, size_type n=npos );
```

Το πρέπει να έχει μόνον τους χαρακτήρες "`0`" και "`1`". Αν βρεθεί κάτι άλλο ρίχνεται εξαίρεση (*std*::)*invalid_argument*. Μπορείς να ορίσεις και υποορμαθό (*n* χαρακτήρες ξεκινώντας από *pos*). Οι εντολές

```
bitset<12> bss0( string("100100101010") ),
           bss1( string("100100101") ),
           bss2( string("111111001001010100101010") );
cout << bss0 << endl << bss1 << endl << bss2 << endl;
```

δίνουν:

```
100100101010
000100100101
010100101010
```

Στο *bss0* βάζουμε ως αρχική τιμή ορμαθό (με "`0`" και "`1`") με μήκος 12. Τα ψηφία γίνονται τιμές των δυαδικών ψηφίων της *bss0*.

Στη δεύτερη περίπτωση έχουμε ορμαθό με 9 ψηφία και συμπληρώνεται με τρία "`0`" στην αρχή.

Στην τελευταία περίπτωση ο ορμαθός έχει μήκος μεγαλύτερο από 12. Η παίρνει την τιμή της από τους *τελευταίους* 12 χαρακτήρες.

Ας δούμε τώρα τις αντιστοιχίες μεταξύ του (δικού μας) *Bitmap* και του *bitset*:

- Αντί για τη *bitValue()* έχουμε τον τελεστή "`[]`" που επιφορτώνεται με μεθόδους:

```
bitset<N>::reference operator[]( size_t pos );
bool operator[]( size_t pos ) const;
```

Για τον τύπο "`bitset<N>::reference`" ισχύουν αυτά που είπαμε στην §26.2.1.2 για τον "`vector<bool>::reference`".

Μπορείς ακόμη να πάρεις την τιμή του ψηφίου σε τύπο `bool` με τη μέθοδο

```
bool test( size_t pos ) const;
```

- Αντί για τη *setBit()* έχουμε τη

```
bitset<N>& set( size_t pos, int val=1 );
```

Πάντως αυτή μπορεί να χρησιμοποιηθεί και για να μηδενίσουμε ένα ψηφίο αν κληθεί με δεύτερο όρισμα "`0`".

Υπάρχει όμως και μια άλλη μορφή της *set()*

```
bitset<N>& set();
```

που βάζει τιμή "`1`" σε όλα τα ψηφία του `*this`.

- Αντί για την *clearBit()* υπάρχει η
  `bitset<N>& reset( size_t pos );`
  Είπαμε όμως ότι μπορείς να χρησιμοποιήσεις και τη *set()* με δεύτερο όρισμα "`0`".
  Και για τη *reset()* υπάρχει η μορφή
  `bitset<N>& reset();`
  που μηδενίζει όλα τα ψηφία του `*this`.
- Αντί για την *count1()* υπάρχει η
  `size_t count() const;`
- Αντί για τη *display()* χρησιμοποίησε τον τελεστή "`<<`" όπως κάναμε στα παραδείγματα που δώσαμε παραπάνω.

Για τα τις *part(), firstBitSet()* δεν υπάρχουν αντίστοιχες.
Άλλες μέθοδοι που υπάρχουν στο *bitset*:

- Μέθοδοι αντιστροφής δυαδικών ψηφίων: Η
  `bitset<N>& flip( size_t pos );`
  αντιστρέφει ("`0`"→"`1`" και "`1`"→"`0`") το ψηφίο στη θέση *pos* και επιστρέφει το `*this` και η
  `bitset<N>& flip();`
  επιστρέφει το `*this` αφού αντιστρέψει *όλα* τα ψηφία του. Οι "`bs1 = ~bs0`" και "`bs1 = bs0.flip()`" έχουν το ίδιο αποτέλεσμα.
- Μέθοδος μετατροπής σε `unsigned long int`:
  `unsigned long to_ulong() const;`
  Επιστρέφει την τιμή `unsigned long int` που αντιστοιχεί στα δυαδικά ψηφία που είναι αποθηκευμένα στο `*this`. Αν τα ψηφία είναι περισσότερα από αυτά που χρησιμοποιούνται για τον `long` ρίχνει εξαίρεση (*std*`::`)*overflow_error*.
- Μέθοδος μετατροπής σε `string`:
  `string to_string() const;`
  Επιστρέφει την τιμή `string` που προκύπτει από την αντιστοίχιση κάθε ψηφίου μηδέν στον χαρακτήρα `'0'` και κάθε ψηφίου ένα στον χαρακτήρα `'1'`.
- Μέθοδος:
  `bool any() const;`
  που επιστρέφει `true` αν έστω και ένα ψηφίο του `*this` έχει τιμή ένα.
- Μέθοδος:
  `bool none() const;`
  που επιστρέφει `true` αν δεν υπάρχει ψηφίο του `*this` με τιμή ένα.

Παρατήρηση:▶

Θα πεις τώρα: «Καλα, τα "*bits*" τα είδαμε· το "*set*" που είναι;» Ας δούμε ένα (όχι πολύ καλό) παράδειγμα: Γυρνάμε στο Project 4 και θέλουμε σε κάθε *Course* να βάλουμε και το σύνολο των φοιτητών που έχουν εγγραφεί σε αυτό. Τα στοιχεία των φοιτητών φυλάγονται σε *scNOfStudents* θέσεις του πίνακα scArr. Ένας τρόπος να λύσουμε το πρόβλημά μας είναι ο εξής: Σε κάθε αντικείμενο κλάσης *Course* προσθέτουμε ένα ακομή μέλος:

```
Bitset<allStudents.getNOfStudents()> cEnrolled;
```

Αν ο φοιτητής με τα στοιχεία του στη `scArr[k]` είναι γραμμένος στο μάθημα *crs* τότε το `crs.cEnrolled[k]` έχει τιμή "`1`" αλλιώς "`0`". Σε μια τέτοια περίπτωση αχρηστεύεται και το *cNoOfStudents* αφού θα μπορούμε να έχουμε:

```
unsigned int Course::getNoOfStudents() const
{  return cEnrolled,count();  }
```

Γιατί δεν είναι πολύ καλό το παράδειγμα; Διότι ο πίνακας φοιτητών δεν είναι και τοσο σταθερός: φοιτητές εισάγονται και διαγράφονται και οι θέσεις των στοιχείων στον πίνακα μπορεί να μεταβάλλονται.◀

26.5.2 Το Περίγραμμα "*complex*"

Βάζοντας στο πρόγραμμά σου "`#include <complex>`" μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template< class T >
class complex
{
public:
   typedef T value_type;
   complex( const T& re=T(), const T& im=T() );
   complex( const complex& );
   template<class X>  complex( const complex<X>& );
   T real() const;
   T imag() const;
   complex<T>& operator=( const T& );
   complex<T>& operator+=( const T& );
   complex<T>& operator-=( const T& );
   complex<T>& operator*=( const T& );
   complex<T>& operator/=( const T& );
   complex& operator=( const complex& );
   template<class X> complex<T>& operator=( const complex<X>& );
   template<class X> complex<T>& operator+=( const complex<X>& );
   template<class X> complex<T>& operator-=( const complex<X>& );
   template<class X> complex<T>& operator*=( const complex<X>& );
   template<class X> complex<T>& operator/=( const complex<X>& );
}; // complex
```

για μιγαδικούς αριθμούς.

Στο `complex` υπάρχουν και τρεις εξειδικεύσεις του περιγράμματος για `float`, `double` και `long double`.

Ακόμη, με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται:

- Οι τελεστές "`+`", "`-`", "`*`", "`/`".
- Οι συναρτήσεις *abs()* και *arg()* με τα

```
template< class T > T abs( const complex<T>& x )
{  return sqrt( x.real()*x.real()+x.imag()*x.imag() );  }
template< class T > T arg( const complex<T>& x )
{  return atan2( imag(x), real(x) )  }
```

που μας δίνουν τα συστατικά της πολικής μορφής.

- Η συνάρτηση (περίγραμμα)

```
template< class T >
complex<T> polar( const T& rho, const T& theta = 0 );
```

που μας δίνει έναν μιγαδικό από τα πολικά συστατικά του. Θα πρέπει να έχουμε:

$$x == polar(\ abs(x), arg(x)\ )$$

αλλά, αφού έχουμε να κάνουμε με τιμές κινητής υποδιαστολής, αντί για το "`==`" καλύτερα να σκέφεται "`≈`".

- Η συνάρτηση

```
template< class T > complex<T> conj( const complex<T>& x );
```

που μας δίνει τον συζυγή του *x*.

- Περιγράμματα συναρτήσεων για *sqrt()*, *pow()*, *cos()*, *sin()*, *tan()*, *exp()*, *log()*, *log10()*, *cosh()*, *sinh()*.

26.6 Τι (Πρέπει να) Έμαθες στο Κεφάλαιο Αυτό

Όταν λέμε STL εννοούμε *περιέχοντα*, *προσεγγιστές*, *συναρτησιακά αντικείμενα* και *αλγόριθμους* (και όχι απλώς το *vector* όπως πιστεύουν μερικοί).

Τα *περιέχοντα* της STL είναι *vector*, *list*, *deque* (ακολουθίες) και *set*, *map*, *multiset* και *multimap* (συνειρμικά). Το ότι μέχρι τώρα παριστάναμε όλες τις συλλογές αντικειμένων με πίνακες (απλούς ή δυναμικούς) δεν σημαίνει ότι από εδώ και πέρα θα τις παριστάνουμε με αντικείμενα στιγμιοτύπων του *vector*. Το περιέχον που επιλέγουμε εξαρτάται από το πρόβλημα που έχουμε να λύσουμε.

Οι *προσεγγιστές* είναι πολύτιμα εργαλεία για την αξιοποίηση των περιεχόντων αλλά και των αλγορίθμων. Με βάση τις δυνατότητές τους κατηγοριοποιούνται σε *προσεγγιστές εισόδου*, *εξόδου*, *πρόσθιους*, *αμφίδρομους* και *τυχαίας πρόσβασης*.

Τα *συναρτησιακά αντικείμενα* είναι απαραίτητα για την παραμετροποίηση των περιγραμμάτων κλάσεων και συναρτήσεων. Δεν χρειάζεται να γράφουμε τα πάντα: η STL μας δίνει μια μικρή αλλά πολύ χρήσιμη συλλογή.

Για να χρησιμοποιήσουμε τα περιέχοντα της STL, εκτός από τις μεθόδους τους, μας δίνονται και αρκετοί *αλγόριθμοι* (περιγράμματα συναρτήσεων). Πολλοί από αυτούς δρουν σε συλλογές αντικειμένων. Σε τέτοιες περιπτώσεις στις παραμέτρους της συνάρτησης περιλαμβάνονται προσεγγιστές *first*, *last* και υπονοείται επεξεργασία των αντικειμένων **it* όταν ο *it* διατρέχει την περιοχή [*first*, *last*).

Στις βιβλιοθήκες της C++ υπάρχουν και άλλα περιγράμματα κλάσεων εκτός από τα περιέχοντα. Εν συντομία είδαμε δύο τέτοια παραδείγματα: *bitset* και *complex*.

Ασκήσεις

26-1 Αν

```
vector < T > v1;
```

τι αποτέλεσμα έχουν οι εντολές:

```
set< T > s1( v1.begin(), v1.end() );
vector< T > v2( s1.begin(), s1.end() );
```

Πώς αλλιώς (χωρίς να χρησιμοποιήσεις σύνολο) θα μπορούσες να βάλεις στο *v1* το τελικό περιεχόμενο του *v2*;

26-2 Δύο συναρτήσεις χρήσιμες για πολυσύνολα είναι οι εξής:

- *eqmult()*: η *eqmult(v, M_1, M_2)* μας δίνει τιμή `true` αν δύο πολυσύνολα M_1, M_2 έχουν ίσα πλήθη του ατόμου *v*.
- **κλιμάκωση** (scaling): η $n \otimes M_1$ (*n*: φυσικός) μας δίνει πολυσύνολο που περιέχει τα στοιχεία του M_1 και μόνον αυτά αλλά σε *n*-πλάσιο πλήθος από ότι τα έχει το M_1.

Να τις υλοποιήσεις με τα κατάλληλα περιγράμματα συναρτήσεων που θα είναι δυνατόν να χρησιμοποιηθούν για στιγμιότυπα των *multiset*, *multimap*.

project

7

Φοιτητές και Μαθήματα με STL

Περιεχόμενα:

Εισαγωγικό Σημείωμα:

Στόχος αυτού του Project είναι να ξαναδούμε το γνωστό μας πρόβλημα χρησιμοποιώντας αυτά που μάθαμε για την STL. Θα αλλάξουμε τη λύση που είδαμε στο Project 4 με δύο τρόπους:

- Στην πρώτη περίπτωση θα αλλάξουμε όλους τους δυναμικούς πίνακες σε *vector*.
- Στη δεύτερη περίπτωση θα επιλέγουμε το περιέχον που μας φαίνεται πιο κατάλληλο.

Και στις δύο περιπτώσεις, θα κάνουμε τις μετατροπές υπακούοντας στον εξής περιορισμό: *Δεν θα πρέπει να αλλάζει ο τρόπος που καλούνται οι μέθοδοι που είναι "*`public`*" ώστε* να μην χρειάζεται να αλλάξουμε τα προγράμματα που χρησιμοποιούν τις κλάσεις μας. Αυτό σημαίνει ότι οι μέθοδοι θα πρέπει:

- Να έχουν τις ίδιες προδιαγραφές (προϋπόθεση-απαίτηση).
- Να έχουν την ίδια επικεφαλίδα.

Για τις μεθόδους που είναι "`private`" είμαστε πιο ελαστικοί.

Όπως θα δεις στη συνέχεια, θα παραβιάσουμε τον παραπάνω περιορισμό μόνον στην περίπτωση μιας συγκεκριμένης λειτουργίας: στην εξαγωγή ολόκληρου πίνακα (όπως, για παράδειγμα, ο ccArr της *CourseCollection*).

Prj07.1 Το Πρόγραμμα με STL I: *vector*

Θα αλλάξουμε τη λύση του Project 4 αντικαθιστώντας με το κατάλληλο στιγμιότυπο του "`vector`" κάθε πίνακα που έχουμε στη λύση. Πέρα από αυτό μπορεί να αλλάζουμε και μερικά από αυτά που έχουμε γράψει με κάτι «καλό» από την STL.

Prj07.1.1 Κλάση *Course*

Η κλάση *Course* δεν έχει πίνακες και έτσι δεν αλλάζει.

Prj07.1.2 Κλάση *CourseCollection*

Εδώ η βασική αλλαγή είναι στη δήλωση του ccArr:

```
   vector<Course> ccArr;
```

Τα *ccIncr*, *ccNOfCourses*, *ccReserved* μας είναι άχρηστα. Η *getNOfCourses()* γίνεται:

```
   size_t getNOfCourses() const {  return ccArr.size();  }
```

Με τη *getArr()* τι θα κάνουμε; Μια λύση είναι:

```
   const Course* getArr() const {  return &ccArr[0];  }
```

Θα το ξανασυζητήσουμε…

Πέρα από τα παραπάνω, η πρώτη απλούστευση της κλάσης φαίνεται στον δημιουργό:

```
   CourseCollection() { };
```

και στον καταστροφέα που δεν τον χρειάζόμαστε! Εμείς όμως, κατά τη συνήθειά μας θα γράψουμε:

```
   ~CourseCollection() { };
```

Θα ξεκινήσουμε τις μετατροπές από την (εσωτερική) συνάρτηση-μέλος *findNdx()* που χρησιμοποιείται παντού. Αντί για τη *linSearch()* (για να γίνουμε «πιο STL») θα χρησιμοποιήσουμε την (*std*::)*find()*:

```
vector<Course>::iterator CourseCollection::findNdx( string code )
{
   vector<Course>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = ccArr.end();
   else
      ndx = find( ccArr.begin(), ccArr.end(), Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Η *findNdx()* επέστρεφε τιμή τύπου `int` που ήταν δείκτης στοιχείου του ccArr (δηλαδή `ccArr[ndx]`). Τώρα επιστρέφει έναν προσεγγιστή προς στοιχείο του πίνακα· έτσι:

όπου βάζαμε "`ccArr[ndx]`" θα βάζουμε "`*ndx`" και

όπου βάζαμε "`ccArr[ndx].`*X*" θα βάζουμε "`ndx->`*X*"

Ακόμη, οι πράξεις "`++ccNOfCourses`" και "`--ccNOfCourses`" δεν έχουν νόημα και θα πρέπει να αφαιρεθούν.

Παρατήρηση:▶

Για σύνδεση με τα προηγούμενα κρατούμε το όνομα "*findNdx*" που όμως είναι συντομογραφία του «*find index*» (βρες δείκτη). Τώρα –που η μέθοδος δεν επιστρέφει δείκτη αλλά προσεγγιστή– θα έπρεπε να αλλάξουμε το όνομα σε κάτι σαν "*findIt*" για το «*find iterator*» (βρες προσεγγιστή).◀

Ξεκινούμε με τη:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) != ccArr.end() );  };
```

Συνεχίζουμε με τη *delete1Course()*:

```
void CourseCollection::delete1Course( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx != ccArr.end() )  // υπάρχει
   {
      vector<Course>::iterator k;
      k = find_if( ccArr.begin(), ccArr.end(),
                   bind2nd(EqPrereq(),code) );
      if ( k != ccArr.end() )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::prereqRef,
                                      code.c_str() );
      int enrStdnt( ndx->getNoOfStudents() );
      if ( enrStdnt > 0 ) // υπάρχουν εγγραφές φοιτητών
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::enrollRef,
                                      code.c_str(), enrStdnt );
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Εδώ, πέρα από τις αλλαγές που έχουν σχέση με την αλλαγή τύπου του *ndx*, έχουμε και κάτι άλλο: ενσωματώσαμε αυτό που είδαμε στο παράδειγμα της §26.1.4. Εδώ βέβαια το *k* είναι τύπου `vector<Course>::iterator` και αντί για `ccArr`, `ccArr+ccNOfCourses` έχουμε `ccArr.begin()`, `ccArr.end()` αντιστοίχως. Η *EqPrereq* είναι ίδια:

```
struct EqPrereq :  binary_function< Course, string, bool >
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Πρόσεξε ότι πρέπει να αλλάξει η

```
void CourseCollection::erase1Course( vector<Course>::iterator ndx )
{
  *ndx = ccArr.back();
  ccArr.pop_back();
} // CourseCollection::erase1Course
```

Και προχωρούμε στην *add1Course()*:

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   vector<Course>::iterator ndx( findNdx(aCourse.getCode()) );
   if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )  // υπάρχει
      {                                            // προαπαιτούμενο
         vector<Course>::iterator ndx(
                                    findNdx(aCourse.getPrereq()) );
         if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί του
                                   // προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                    CourseCollectionXptn::prereqRef,
                                        aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      insert1Course( aCourse );
      ndx = findNdx( aCourse.getCode() );
      ndx->clearStudents();
```

```
   }
} // CourseCollection::add1Course
```

όπου:

```
void CourseCollection::insert1Course( const Course& aCourse )
{
   try {  ccArr.push_back( aCourse );  }
   catch ( bad_alloc& )
   {  throw CourseCollectionXptn( "insert1Course",
                              CourseCollectionXptn::allocFailed );  }
} // CourseCollection::insert1Course
```

Εδώ, όπως περιμέναμε, η απλούστευση είναι εντυπωσιακή. Βέβαια, η πιθανότητα να «ξεμείνουμε» από μνήμη υπάρχει και παίρνουμε τα μέτρα μας.

Η τελευταία μέθοδος «ενός στοιχείου» είναι η:

```
const Course& CourseCollection::get1Course( string code ) const
{
   vector<Course>::const_iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return *ndx;
} // CourseCollection::get1Course
```

Πρόσεξε το "`const_iterator ndx`": αυτό είναι απαραίτητο συμπλήρωμα των "`const`" της επικεφαλίδας.

Πριν δούμε τις μεθόδους φύλαξης-φόρτωσης ας δούμε τη

```
void CourseCollection::swap( CourseCollection& rhs )
{
   ccArr.swap( rhs.ccArr );
} // CourseCollection::swap
```

που όπως βλέπεις απλουστεύθηκε.

Η φύλαξη γίνεται με τη:

```
void CourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::fileNotOpen );
   size_t  ccNOfCourses( ccArr.size() );
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( vector<Course>::const_iterator it(ccArr.begin());
         it != ccArr.end(); ++it )
      it->save( bout );
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::cannotWrite );
} // CourseCollection::save
```

και η φόρτωση με τη:

```
void CourseCollection::load( ifstream& bin )
{
   CourseCollection tmp;
   size_t n;  //ccNOfCourses

   bin.read( reinterpret_cast<char*>(&n), sizeof(size_type) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
```

```
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Η *load()* μένει όπως ήταν. Αυτό οφείλεται στο ότι δηλώσαμε την *tmp* τύπου *CourseCollection*. Αν επιλέξουμε "`vector<Course> tmp`" τότε αλλάζουμε:

τη "`tmp.insert1Course(oneCourse)`" σε "`tmp.push_back(oneCourse)`"

και τη "`swap(tmp)`" σε "`ccArr.swap(tmp)`"

και η φόρτωση γίνεται κάπως πιο γρήγορα.

Η *display()* δεν αλλάζει παρά μόνο στη συνθήκη της **for**. Θα πρέπει να γίνει "`k < ccArr.size()`".

Οι *add1Student()* και *delete1Student()* –ενδιάμεσες προς τις μεθόδους της *Course* με το ίδιο όνομα– αλλάζουν όπως είπαμε στην αρχή της παραγράφου:

```
void CourseCollection::add1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->add1Student();
} // CourseCollection::add1Student
```

και

```
void CourseCollection::delete1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->delete1Student();
} // CourseCollection::delete1Student
```

Prj07.1.2.1 Σχετικώς με τη *getArr()*

Υποσχεθήκαμε να ξανασυζητήσουμε για τη μέθοδο *CourseCollection*::*getArr()* και αυτό θα κάνουμε αλλά πριν προχωρήσεις καλό είναι να γυρίσεις στην §20.7.5 και να διαβάσεις το σημείο "**5. Μέθοδος** *getAllStops()*".

Εκεί είχαμε καταλήξει ότι το καλύτερο που είχαμε να κάνουμε για να «βγάλουμε» έναν πίνακα από το αντικείμενο ήταν να επιστρέψουμε δυναμικό πίνακα, αντίγραφο του εσωτερικού. Βέβαια επισημάναμε τον κίνδυνο «διαρροής» μνήμης και στηρίζαμε τις ελπίδες μας για την αποφυγή της στις καλές συνήθειες των πεπειραμένων προγραμματιστών.[1]

Τώρα θα δούμε μια πιο σίγουρη λύση, με παράδειγμα εφαρμογής στον ccArr:

```
void CourseCollection::getArr( vector<Course>& crsArr ) const
{
   try  {  crsArr = ccArr;  }
   catch( bad_alloc& )
   {  throw CourseCollectionXptn( "getArr",
                                  CourseCollectionXptn::allocFailed );  }
} // CourseCollection::getArr
```

Αυτή η λύση είναι στην πραγματικότητα μια μορφή RAII. Όταν δώσεις:

```
   CourseCollection allCourses;
// . . .
   vector<Course> courseTbl;
```

[1] Που, όμως, όταν ρίχνονται εξαιρέσεις, μπορεί να μην είναι αρκετές.

```
// . . .
   allCourses.getArr( courseTbl );
```

η δυναμική μνήμη που παίρνουμε κρύβεται μέσα στην *courseTbl*. Όταν τελειώσει η ζωή της ο καταστροφέας της θα ανακυκλώσεί αυτομάτως και τη μνήμη.

Φυσικά, θα υπενθυμίσουμε ότι και αυτή η «καλύτερη λύση» κοστίζει σε υπολογιστικό χρόνο αφού έχουμε αντιγραφή πίνακα.

Με αυτόν τον τρόπο θα βγάζουμε όλους τους πίνακες στη συνέχεια. Στο σημείο αυτό θα αλλάξουν οι διεπαφές όλων των κλάσεων.

Prj07.1.3 Κλάση *Student*

Η *Student* αλλάζει αφού θα αντικαταστήσουμε τα «εργαλεία υλοποίησης» του δυναμικού πίνακα μαθημάτων:

```
   enum { sIncr = 3 };
   size_t             sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
   size_t             sReserved;
```

με τη

```
   vector<Course::CourseKey> sCourses;
```

Ύστερα από αυτό θα έχουμε:

```
   unsigned int getNoOfCourses() const {  return sCourses.size();  }
   void clearCourses() {  sCourses.clear();  sWH = 0;  }
```

Φυσικά, οι πράξεις "`++sNoOfCourses`" και "`--sNoOfCourses`" και εδώ δεν έχουν νόημα και θα πρέπει να αφαιρεθούν.

Γράφουμε τη *getCourses()* όπως κάναμε στην *CourseCollection* με τη *getArr()*:

```
void Student::getCourses( vector<Course::CourseKey>& crsTbl ) const
{
   try {  crsTbl = sCourses;  }
   catch( bad_alloc& )
   {  throw StudentXptn( sIdNum, "getCourses",
                         StudentXptn::allocFailed );  }
} // Student::getCourses
```

Αρχίζουμε με τους δημιουργούς: Ο ερήμην δημιουργός γίνεται:

```
Student::Student( int aIdNum )
   :  sWH( 0 )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum,
                         aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
}; // Student()
```

και ο δημιουργός αντιγραφής:

```
Student::Student( const Student& rhs )
{
   sIdNum = rhs.sIdNum;
   strcpy( sSurname, rhs.sSurname );
   strcpy( sFirstname, rhs.sFirstname );
   sWH = rhs.sWH;
   try  {  sCourses = rhs.sCourses;  }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "Student",
                         StudentXptn::allocFailed );
   }
} // Student( const Student& rhs )
```

Τώρα πρόσεξε: *δημουργός αντιγραφής δεν χρειάζεται κατ' αρχήν* αφού το περίγραμμα *vector* έχει (σωστό) δημιουργό αντιγραφής. Και γιατί το βάλαμε; Για να πιάνουμε και να αλλάζουμε τη *bad_alloc*.

Παρατήρηση:▶

Και για τον ερήμην δημιουργό δεν χρειάζεται να κάνουμε το ίδιο; Κατ' αρχήν: ναι!◀

Θα συνεχίσουμε με τη *findNdx()*· την παίρνουμε από την *CourseCollection*::*findNdx()* αν βάλουμε:

- Τύπο προσεγγιστή

  "`vector<Course::CourseKey>::iterator`"

  (αντί για "`vector<Course>::iterator`").
- Όνομα πίνακα "`sCourses`" (αντί για "`ccArr`").
- Αντικείμενο αναζήτησης "`Course::CourseKey(code)`" (αντί για "`Course(code)`").

```
vector<Course::CourseKey>::iterator Student::findNdx( const string& code )
{
   vector<Course::CourseKey>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = sCourses.end();
   else
      ndx = find( sCourses.begin(), sCourses.end(),
                  Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Και μετά από αυτήν οι τρεις μέθοδοι «ενός στοιχείου»:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) != sCourses.end() );  };

void Student::add1Course( const Course& oneCourse )
{
   if ( findNdx(oneCourse.getCode()) == sCourses.end() )
   {
      insert1Course( Course::CourseKey(oneCourse.getCode()) );
      sWH += oneCourse.getWH();
   }
} // Student::add1Course

void Student::delete1Course( const Course& oneCourse )
{
   vector<Course::CourseKey>::iterator
                                 ndx( findNdx(oneCourse.getCode()) );
   if ( ndx != sCourses.end() )  // υπάρχει
   {
      erase1Course( ndx );
      sWH -= oneCourse.getWH();
   }
} // Student::delete1Course
```

όπου:

```
void Student::insert1Course( const Course::CourseKey& aCode )
{
   try {  sCourses.push_back( aCode );  }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "insert1Course",
                         StudentXptn::allocFailed );
   }
} // Student::insert1Course

void Student::erase1Course( vector<Course::CourseKey>::iterator ndx)
{
   *ndx = sCourses.back();
   sCourses.pop_back();
```

```
} // Student::erase1Course
```

Από τις υπόλοιπες μεθόδους αλλάζουν και οι:

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );

   std::swap( sWH, rhs.sWH );

   sCourses.swap( rhs.sCourses );
} // Student::swap

void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
   bout.write( sFirstname, sizeof(sFirstname) );
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
   size_t noOfCourses( sCourses.size() );
   bout.write( reinterpret_cast<const char*>(&noOfCourses),
               sizeof(noOfCourses) );// αριθμός μαθημάτων που δήλωσε
   for ( int k(0); k < noOfCourses; ++k )
      bout.write( sCourses[k].s, Course::cCodeSz );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save

void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      bin.read( reinterpret_cast<char*>(&tmp.sWH), sizeof(sWH) );
      size_t noOfCourses;              // αριθ. μαθημ. που δήλωσε
      bin.read( reinterpret_cast<char*>(&noOfCourses), sizeof(noOfCourses) );
      for ( int k(0); k < noOfCourses; ++k )
      {
         char aKey[Course::cCodeSz];
         bin.read( aKey, Course::cCodeSz );
         tmp.sCourses.push_back( Course::CourseKey(aKey) );
      }
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load",
                            StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load
```

Prj07.1.4 Κλάση *StudentCollection*

Εδώ, όπως κάναμε στις προηγούμενες κλάσεις, θα αντικαταστήσουμε τις

```
   enum { scIncr = 30 };
```

```
   Student*                scArr;
   size_t                  scNOfStudents;
   size_t                  scReserved;
```

με τη

```
   vector<Student>            scArr;
```

Και όπως μάθαμε, θα πρέπει να αλλάξουμε τις:

```
   size_t getNOfStudents() const {  return scArr.size();  }

void StudentCollection::getArr( vector<Student>& stdntTbl ) const
{
   try {  stdntTbl = scArr;  }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                            StudentCollectionXptn::allocFailed );  }
} // StudentCollection::getArr
```

Ο δημιουργός και ο καταστροφέας απλουστεύονται:

```
StudentCollection::StudentCollection()
{  scPAllEnrollments = 0;  }
   ~StudentCollection() { };
```

Η συνέχεια είναι γνωστή:

```
vector<Student>::iterator StudentCollection::findNdx( int aIdNum )
{
   vector<Student>::iterator ndx;
   if ( aIdNum <= 0 )
      ndx = scArr.end();
   else
      ndx = find( scArr.begin(), scArr.end(), Student(aIdNum) );
   return ndx;
} // StudentCollection::findNdx

   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) != scArr.end() );  };

const Student& StudentCollection::get1Student( int aIdNum ) const
{
   vector<Student>::const_iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "get1Student",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   return *ndx;
} // StudentCollection::get1Student

void StudentCollection::add1Student( const Student& aStudent )
{
   vector<Student>::iterator ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx == scArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      insert1Student( aStudent );
      vector<Course::CourseKey> aStCourses;
      aStudent.getCourses( aStCourses );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                     StudentInCourse(aStudent.getIdNum(), aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Στην *add1Student()* πρόσεξε την αλλαγή στον χειρισμό του *aStCourses* λόγω της αλλαγής της *Student*::*getCourses()*.

```
void StudentCollection::delete1Student( int aIdNum )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx != scArr.end() )  // υπάρχει
   {
      if ( ndx->getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student

void StudentCollection::insert1Student( const Student& aStudent )
{
   try {  scArr.push_back( aStudent );  }
   catch( MyTmpltLibXptn& )
   {
      throw StudentCollectionXptn( "insert1Student",
                                   StudentCollectionXptn::allocFailed );
   }
} // StudentCollection::insert1Student

void StudentCollection::erase1Student(vector<Student>::iterator ndx)
{
   *ndx = scArr.back();
   scArr.pop_back();
} // StudentCollection::erase1Student
```

Εδώ έχουμε επιπλέον:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   ndx->add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   ndx->delete1Course( aCourse );
} // StudentCollection::delete1Course
```

Τέλος, έχουμε τις:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   scArr.swap( rhs.scArr);
} // StudentCollection::swap

void StudentCollection::save( ofstream& bout, SIndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   size_t nOfStudents( scArr.size() );
   bout.write( reinterpret_cast<const char*>(&nOfStudents),
               sizeof(nOfStudents) );
```

```
   for ( int k(0); k < nOfStudents; ++k )
   {
      index[k].sIdNum = scArr[k].getIdNum();
      index[k].loc = bout.tellp();
      scArr[k].save( bout );
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save

void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.insert1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Ένα ερώτημα για τη δεύτερη παράμετρο της *save()*: Δεν θα αλλάξουμε τον πίνακα του ευρετηρίου; Θα τον αλλάξουμε· είπαμε όμως να μην αλλάξουμε τις διεπαφές των κλάσεων. Θα τα πούμε στη συνέχεια…

Prj07.1.5 Κλάση *StudentInCourse*

Η κλάση *StudentInCourse* δεν χρειάζεται τροποποίηση.

Prj07.1.6 Κλάση *StudentInCourseCollection*

Εδώ, όπως καταλαβαίνεις, θα δηλώσουμε τον δυναμικό πίνακα ως:

```
   vector<StudentInCourse> siccArr;
```

Αυτό έχει τις εξής άμεσες επιπτώσεις:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection

   ~StudentInCourseCollection()  {  };

   size_t getNOfStudentInCourses() const  {  return siccArr.size();  }

void StudentInCourseCollection::getArr( vector<StudentInCourse>& sicTbl) const
{
   try {  sicTbl = siccArr;  }
   catch( bad_alloc& )
   {  throw StudentInCourseCollectionXptn( "getArr",
                    StudentInCourseCollectionXptn::allocFailed );  }
} // StudentInCourseCollection::getArr
```

Και –κατά τη συνήθειά μας– συνεχίζουμε με τη:

```
vector<StudentInCourse>::iterator
                StudentInCourseCollection::findNdx( int aIdNum, string code )
{
   vector<StudentInCourse>::iterator ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = siccArr.end();
   else
      ndx = find( siccArr.begin(), siccArr.end(),
                  StudentInCourse(aIdNum,code) );
   return ndx;
} // StudentInCourseCollection::findNdx
```

Μετά από αυτήν:

```
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) != siccArr.end() );  }

const StudentInCourse&
          StudentInCourseCollection::get1StudentInCourse( int aIdNum,
                                                          string code ) const
{
   vector<StudentInCourse>::const_iterator
                                        ndx( findNdx(aIdNum, code) );
   if ( ndx == siccArr.end() ) // δεν υπάρχει στοιχείο
      throw StudentInCourseCollectionXptn( "get1StudentInCourse",
                           StudentInCourseCollectionXptn::notFound,
                           StudentInCourse::SICKey(aIdNum, code) );
   return *ndx;
} // StudentInCourseCollection::get1StudentInCourse
```

Στην *add1StudentInCourse()* το μόνο που αλλάζει είναι η συνθήκη της πρώτης **if**:

```
void StudentInCourseCollection::add1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs )
{
   if ( findNdx( aStdnInCrs.getSIdNum(),     // δεν υπάρχει φοιτητής
                 aStdnInCrs.getCCode()) == siccArr.end() )
   {
   // . . .
      insert1StudentInCourse( aStdnInCrs );
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Η *insert1StudentInCourse()* απλουστεύεται:

```
void StudentInCourseCollection::insert1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs )
{
   try {  siccArr.push_back( aStdnInCrs );  }
   catch( MyTmpltLibXptn& )
   {
      throw StudentInCourseCollectionXptn( "insert1StudentInCourse",
                                  StudentInCourseCollectionXptn::allocFailed );
   }
} // StudentInCourseCollection::insert1StudentInCourse
```

Για τις διαγραφές έχουμε:

```
void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                        string code )
{
   vector<StudentInCourse>::iterator ndx( findNdx(aIdNum, code) );
   if ( ndx != siccArr.end() ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                     StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                   StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course( code) );
```

```
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      erase1StudentInCourse( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse
```

όπου:

```
void StudentInCourseCollection::erase1StudentInCourse(
                                      vector<StudentInCourse>::iterator ndx )
{
   *ndx = siccArr.back();
   siccArr.pop_back();
} // StudentInCourseCollection::erase1StudentInCourse
```

Και η *swapArr()* απλουστεύεται:

```
void StudentInCourseCollection::swapArr( StudentInCourseCollection& rhs )
{
   siccArr.swap( rhs.siccArr );
} // StudentInCourseCollection::swapArr
```

ενώ οι *save()* και *load()* γίνονται:

```
void StudentInCourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                 StudentInCourseCollectionXptn::fileNotOpen );
   size_t nOfStudentInCourse( siccArr.size() );
   bout.write( reinterpret_cast<const char*>(&nOfStudentInCourse),
               sizeof(nOfStudentInCourse) );
   for ( int k(0); k < nOfStudentInCourse; ++k )
      siccArr[k].save( bout );
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                        StudentInCourseCollectionXptn::cannotWrite );
} // StudentInCourseCollection::save

void StudentInCourseCollection::load( ifstream& bin )
{
   StudentInCourseCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         StudentInCourse oneStudentInCourse;
         oneStudentInCourse.load( bin );
         tmp.insert1StudentInCourse( oneStudentInCourse );
      }
      if ( bin.fail() )
         throw  StudentInCourseCollectionXptn( "load",
                                    StudentInCourseCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentInCourseCollection::load
```

Prj07.1.7 Ο Πίνακας-Ευρετήριο

Είπαμε πιο πριν ότι δεν θα αλλάξουμε την επικεφαλίδα της *StudentCollection*::*save()* αλλά είπαμε στην αρχή ότι θα αλλάξουμε όλους τους πίνακες σε αντικείμενα στιγμιοτύπων του *vector*. Αυτά τα δύο είναι συμβατά; Ναι! Στη `main()` δίνουμε:

```
      vector<SIndexEntry> sIndex;
      try
      {  sIndex.reserve( allStudents.getNOfStudents() );  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
```

και στη συνέχεια:

```
saveCollections( allCourses, allStudents, &sIndex[0], allEnrollments );
```

Η *saveCollections()* θα καλέσει:

```
allStudents.save( bout, sIndex );
```

Πρόσεξε το εξής: Προϋπόθεση της *StudentCollection*::*save()* είναι η ύπαρξη των στοιχείων του πίνακα sIndex. Αυτό μας το εξασφαλίζει η (επιτυχής) εκτέλεση της *sIndex*.*reserve()*. Αν δεν τη βάλουμε θα πρέπει να αλλάξουμε τη *save()* και να κάνουμε τις καταχωρίσεις με *push_back()*.

Prj07.2 Το Πρόγραμμα με STL II

Τώρα θα (ξανα)αλλάξουμε τη λύση του Project 4 αντικαθιστώντας κάθε πίνακα που έχουμε στη λύση με το καταλληλότερο –κατά περίπτωση– περιέχον.

Prj07.2.1 Επιλογές

Ξεκινούμε εξετάζοντας τους πίνακες έναν προς έναν για να δούμε με ποιο περιέχον θα αντικαταστήσουμε τον καθένα. Τι εξετάζουμε; Τη χρήση τους στα δύο προγράμματα του Project. Πάντως το δεύτερο πρόγραμμα το βλέπουμε γενικότερα, όπως θα έπρεπε να είναι, για τις ανάγκες της γραμματείας ενός τμήματος σπουδών.

Prj07.2.1.1 ccArr της *CourseCollection*

Σε αντικείμενο κλάσης *CourseCollection* έχουμε τον πίνακα μαθημάτων που υπάρχουν στο πρόγραμμα του τμήματος.

Ο πίνακας μαθημάτων είναι ένα *σύνολο* αντικειμένων τύπου *Course*. Ένα τέτοιο αντικείμενο είναι αρκετά μεγάλο (περί τα 100 B) ενώ το κλειδί είναι μικρό (8 B). Κατ' αρχήν θα πρέπει να παρασταθεί με στιγμιότυπο του *map*.

Ας δούμε όμως και τη χρήση του πίνακα:

- Στο 1ο Πρόγραμμα: ο πίνακας μαθημάτων, αφού φορτωθεί, χρησιμοποιείται μόνο για αναζητήσεις.
- Γενικότερα: Το πρόγραμμα ενός τμήματος αλλάζει μια φορά κάθε 3-4 χρόνια. Επομένως, η συνήθης χρήση του πίνακα μαθημάτων είναι η αναζήτηση πληροφοριών για τα μαθήματα που περιέχει.

Επειδή δεν έχουμε συχνή ενημέρωση του πίνακα θα εξετάσουμε και αυτό που λέμε στο τέλος της §26.4: ταξινομημένο *vector*. Δηλαδή:

- Στο 1ο Πρόγραμμα: ο πίνακας μαθημάτων, φορτώνεται σε ένα `vector <Course>`, ταξινομείται, στη συνέχεια οι αναζητήσεις γίνονται με τη *lower_bound()* και φυλάγεται ταξινομημένος.
- Στα άλλα προγράμματα ο πίνακας φορτώνεται ταξινομημένος και οι αναζητήσεις γίνονται με τη *lower_bound()*.

Στην περίπτωση αυτή θα έχουμε τα εξής προβλήματα:

- Όπως λέγαμε στην §26.4 «*είναι δική σου υποχρέωση να διασφαλίζεις τη μοναδικότητα των μελών του συνόλου.*» Αυτό το κάναμε ήδη στη λύση που έχουμε δώσει (§Prj07.1.2).
- Για να εκμεταλλευτούμε τα πλεονεκτήματα της επιλογής αυτής θα πρέπει να κάνουμε το εξής: Να κρατούμε τον πίνακα ταξινομημένο από την αρχή και να κάνουμε τις εισαγωγές με *insert()* στη σωστή θέση (και όχι με *push_back()*).

Παρατήρηση:▶
Γιατί να μην κάνουμε τις εισαγωγές στον πίνακα όπως την κάναμε στην πρώτη μετατροπή (με *push_back()*), στη συνέχεια να ταξινομήσουμε τον πίνακα και από εκεί και πέρα να ψάχνουμε με *lower_bound()*; Κάτι τέτοιο απαιτεί να έχουμε «διπλές» μεθόδους *findNdx()*, *find1Course()* και *get1Course()*. Αυτό είναι «εγγύηση» για δύσκολα λάθη στη συνέχεια! Ας το αφήσουμε καλύτερα…◀

Πιο πολύ για επίδειξη, θα επιλέξουμε τη λύση με το «ταξινομημένο *vector*». Αν θέλεις να δοκιμάσεις την απεικόνιση από τους κωδικούς μαθημάτων προς αντικείμενα *Course* καλύτερα να μιμηθείς αυτά που θα κάνουμε στη συνέχεια για τη *StudentCollection* και όχι αυτά που κάνουμε στο πρώτο παράδειγμα της §26.3.2.

Prj07.2.1.2 sCourses της *Student*

Εδώ έχουμε ένα πολύ μικρό *σύνολο* κωδικών μαθημάτων (κλειδιών).

Το σύνολο δημιουργείται στην αρχή του εξαμήνου και μπορεί να διορθωθεί μια φορά.[2] Χαρακτηριστική περίπτωση "*set*", αλλά μόνο για λόγους αρχής. Λόγω του μεγέθους του συνόλου δεν έχουμε κέρδος στις αναζητήσεις. Θα δεις ότι έχουμε κάτι μικρά κέρδη στις εισαγωγές και διαγραφές.

Prj07.2.1.3 scArr της *StudentCollection*

Αυτή η συλλογή έχει μερικές εκατοντάδες μέχρι μερικές χιλιάδες μέλη. Ενημερώσεις; Πολύ συχνές!

- Εισαγωγές με τις εγγραφές νέων φοιτητών.
- Διαγραφές με τις αποφοιτήσεις.
- Εισαγωγές με μετεγγραφές φοιτητών *από* άλλα τμήματα.
- Διαγραφές με μετεγγραφές φοιτητών *προς* άλλα τμήματα.
- Διαγραφές λόγω μη ανανέωσης εγγραφής.

Κάθε αντικείμενο της συλλογής είναι μοναδικό (τα στοιχεία ενός φοιτητή)· έχουμε δηλαδή ένα *σύνολο*.

Το μέγεθος του κλειδιού (αριθμός μητρώου του φοιτητή) είναι λιγότερο από 10% του μεγέθους του αντικειμενου. Εξ άλλου υπάρχουν και ενημερώσεις (π.χ. αλλαγές στο sCourses κάποιου φοιτητή) που δεν είναι εισαγωγές/διαγραφές.

Θα κάνουμε την υλοποίηση με "*map*", απεικόνιση των αριθμών μητρώου (κλειδιά) σε αντικείμενα *Student*.

Prj07.2.1.4 siccArr της *StudentInCourseCollection*

Εδώ έχουμε συλλογή με περίπου πενταπλάσιο αριθμό μελών από το προηγούμενο, αφού κάθε φοιτητής δηλώνει κατά μέσον όρο πέντε μαθήματα.

Κάθε αντικείμενο της συλλογής –δήλωση ενός φοιτητή για συμμετοχή σε ένα μάθημα για το τρέχον εξάμηνο– είναι μοναδικό· δηλαδή έχουμε και εδώ ένα *σύνολο*.

Ενημερώσεις; Αρκετές:

- Δηλώσεις φοιτητών για συμμετοχή στα μαθήματα στην αρχή του εξαμήνου.
- Διορθώσεις (διαγραφές-εγγραφές) των δηλώσεων.
- Ακυρώσεις των δηλώσεων όσων φοιτητών μετεγγράφονται *προς* άλλα τμήματα.

Στο τέλος του εξαμήνου έχουμε και άλλη ενημέρωση: εισαγωγή βαθμού.

Τι περιέχον θα επιλέξουμε; Πολλά τα βάλε-βγάλε άρα ούτε σκέψη για "*vector*".

---

[2] Δηλαδή νομίμως μια φορά. Διότι με τις «κλάψες» στη γραμματεία μπορεί να έχουμε και άλλες διορθώσεις.

Το κλειδί (αριθμός μητρώου, κωδικός μαθήματος) είναι –σε μέγεθος– το 75% του αντικειμένου. Μήπως το απλούστερο που έχουμε να επιλέξουμε είναι το "*set*"; Αν κάνουμε αυτήν την επιλογή, πώς θα βάζουμε τους βαθμούς; Κατ' ανάγκη με τις πράξεις: «βγάλε-διάγραψε-διόρθωσε-ξαναβάλε». Αν επιλέξουμε "*map*" θα έχουμε τη δυνατότητα για ενημέρωση επί τόπου.

Θα επιλέξουμε "*map*".

Prj07.2.1.5 Πίνακας–Ευρετήριο

Πόσο μας απλούστευσε το πρόγραμμα η αλλαγή που κάναμε στην §Prj07.1.7; Καθόλου! Ο απλός δυναμικός πίνακας είναι εξ ίσου απλός (αν όχι απλούστερος) με το στιγμιότυπο του *vector*. Φυσικά αυτό δεν είναι περίεργο:

- παίρνουμε όλη τη μνήμη που χρειαζόμαστε με μια `new[]` και την ανακυκλώνουμε με μια `delete[]`,
- συμπληρώνουμε τα στοιχεία με πολύ απλό τρόπο,
- έχουμε τη δυνατότητα να κάνουμε τη φύλαξη του με μια *write()*.

Κατ' αρχή λοιπόν δεν θα αλλάξουμε τη διαχείριση αυτού του πίνακα. Αλλά ας δούμε τι κάνουμε (ξανά και ξανά) στο δεύτερο πρόγραμμα:

- παίρνουμε έναν αριθμό μητρώου φοιτητή,
- βρίσκουμε από το ευρετήριο τη θέση του αντίστοιχου αντικειμένου στο αρχείο με τα στοιχεία των φοιτητών,
- το φορτώνουμε και
- το δείχνουμε.

Είπαμε όμως ότι η συλλογή στοιχείων φοιτητών «*έχει μερικές εκατοντάδες μέχρι μερικές χιλιάδες μέλη.*» Εδώ η ταχύτητα αναζήτησης έχει νόημα.[3] Τι να κάνουμε; Δεν θα αλλάξουμε περιέχον αλλά θα ταξινομήσουμε τον πίνακα (κατ' αύξοντα αριθμό μητρώου) ώστε να υπάρχει δυνατότητα δυαδικής αναζήτησης από τα προγράμματα εφαρμογής που τον χρησιμοποιούν.

Prj07.2.2 Κλάση *Course*

Όπως θα δεις παρακάτω, η *CourseCollection* θέτει απαίτηση για επιφόρτωση του τελεστή "`<`" για την *Course*: [4]

```
bool operator<( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) < 0 );  }

bool operator<( const Course& lhs, const Course& rhs )
{  return ( Course::CourseKey(lhs.getCode()) <
            Course::CourseKey(rhs.getCode()) );  }
```

Prj07.2.3 Κλάση *CourseCollection*

Όπως στην §Prj07.1.2 έτσι και εδώ δηλώνουμε:

```
   vector<Course> ccArr;
```

[3] Θα πεις: και αυτό «κατ' αρχήν» διότι το βήμα «*παίρνουμε έναν αριθμό μητρώου φοιτητή*» εκτελείται με ανθρώπινες ταχύτητες· και θα έχεις δίκιο.

[4] Για λόγους συνέπειας με τους κανόνες μας, βάζουμε ακόμη:

```
bool operator>=( const Course::CourseKey& lhs, const Course::CourseKey& rhs)
{  return ( !(lhs < rhs) );  }
bool operator>=( const Course& lhs, const Course& rhs )
{  return ( !(lhs < rhs) );  }
```

και έχουμε:

```
   CourseCollection() { };
   ~CourseCollection() { };
   size_t getNOfCourses() const {  return ccArr.size();  }
   const Course* getArr() const {  return &ccArr[0];  }
```

Τη *getArr()* θα την αλλάξουμε όπως στην §Prj07.1.2.1.

```
void CourseCollection::getArr( vector<Course>& crsArr ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Να δούμε όμως τι γίνεται με τη *findNdx()*. Τώρα δεν θα καλέσουμε τη *find()* αλλά τη *lower_bound()*:

```
vector<Course>::iterator CourseCollection::findNdx( const string& code )
{
   vector<Course>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = ccArr.end();
   else
      ndx = lower_bound( ccArr.begin(), ccArr.end(), Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Πρόσεξε τώρα: η *lower_bound()* θα μας επιστρέψει προσεγγιστή προς το αντικείμενο που ψάχνουμε (αν υπάρχει) ή προς τη θέση που πρέπει να εισαχθεί (αν δεν υπάρχει). Επομένως η *find1Course()* θα γίνει έτσι:

```
bool CourseCollection::find1Course( const string& code ) const
{
   vector<Course>::iterator ndx( findNdx(code) );
   return ( (ndx != ccArr.end()) && (*ndx == Course(code)) );
} // CourseCollection::find1Course
```

Η συνθήκη στη `return` είναι η διατύπωση της απόφανσης «*το βρήκα*». Όπως λέγαμε στην §5.6, αν η "`ndx != ccArr.end()`" υπολογιστεί `false` η συνάρτηση επιστρέφει `false` χωρίς να κάνει την (απαγορευμένη) αποπαραπομπή του *ndx*. Η διατύπωση της «*δεν το βρήκα*» είναι η άρνηση της παράστασης της `return` που ισοδυναμεί με:

"`ndx == ccArr.end() || (*ndx != Course(code))`"

Και εδώ δεν κινδυνεύουμε από απαγορευμένη αποπαραπομπή.

Για να χρησιμοποιήσουμε αυτή τη μορφή της *lower_bound()* χρειάζεται να έχουμε ορίσει τον "`<`" για την *Course*.

Η *CourseCollection*::*delete1Course()* είναι ίδια με αυτήν της §Prj07.1.2 με μια μόνο διαφορά: στη συνθήκη της **if** που ελέγχει αν υπάρχει το προς διαγραφή αντικείμενο που τώρα γίνεται:

```
   if ( ndx != ccArr.end() && (*ndx == Course(code)) )  // υπάρχει
```

Ίδια είναι και η *EqPrereq*. Πιο σοβαρή είναι η διαφορά στην *erase1Course()* που τώρα γίνεται:

```
void CourseCollection::erase1Course( vector<Course>::iterator ndx )
{
   ccArr.erase( ndx );
} // CourseCollection::erase1Course
```

Γράφουμε ολόκληρη την *add1Course()* στη νέα της μορφή:

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   vector<Course>::iterator ndx( findNdx(aCourse.getCode()) );
   if ( ndx==ccArr.end() || (*ndx!=aCourse) )
   {                                          // δεν βρέθηκε το κλειδί
      if ( strcmp(aCourse.getPrereq(), "") != 0 )  // υπάρχει
      {                                           // προαπαιτούμενο
```

```
         vector<Course>::iterator ndx(findNdx(aCourse.getPrereq()));
         if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί του
                                   // προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::prereqRef,
                                            aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      ndx = insert1Course( ndx, aCourse );
      ndx->clearStudents();
   }
} // CourseCollection::add1Course
```

Ποιες είναι οι διαφορές;

- Η συνθήκη που ελέγχει την τιμή του *ndx*.
- Αλλάξαμε την *insert1Course()* ώστε να επιστρέφει προσεγγιστή προς το αντικείμενο που εισάχθηκε στον πίνακα όπως τον επιστρέφει η *ccArr.insert()*. Έτσι, δεν είναι απαραίτητο να αναζητήσουμε το αντικείμενο που μόλις βάλαμε στον πίνακα.

```
vector<Course>::iterator
               CourseCollection::insert1Course( vector<Course>::iterator ndx,
                                                const Course& aCourse )
{
   try {  return ccArr.insert( ndx, aCourse );  }
   catch ( bad_alloc& )
   {  throw CourseCollectionXptn( "insert1Course",
                             CourseCollectionXptn::allocFailed );  }
} // CourseCollection::insert1Course
```

Η *get1Course()* γίνεται:

```
const Course& CourseCollection::get1Course( string code ) const
{
   vector<Course>::const_iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() || (*ndx != Course(code)) )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return *ndx;
} // CourseCollection::get1Course
```

Δηλαδή, το μόνο που άλλαξε είναι ο έλεγχος του *ndx* στην **if**.

Το ίδιο ισχύει και για τις επόμενες:

```
void CourseCollection::add1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx==ccArr.end() || (*ndx!=Course(code)) )
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->add1Student();
} // CourseCollection::add1Student

void CourseCollection::delete1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx==ccArr.end() || (*ndx!=Course(code)) )
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->delete1Student();
} // CourseCollection::delete1Student
```

Η *swap()* γίνεται όπως στην §Prj07.1.2· το ίδιο και η *save()*.

Η *load()* πρέπει να αλλάξει λίγο: αφού αλλάξαμε την *insert1Course()* αντί "`tmp.insert1Course(oneCourse)`" θα πρέπει να δώσουμε "`tmp.add1Course( oneCourse )`".

Prj07.2.4 Κλάση *Student*

Για το σύνολο κωδικών μαθημάτων δηλώνουμε:

```
   set<Course::CourseKey> sCourses;
```

και ύστερα από αυτό έχουμε (όπως στην §Prj07.1.3):

```
   unsigned int getNoOfCourses() const {  return sCourses.size();  }
   void clearCourses() {  sCourses.clear();  sWH = 0;  }
```

Η *getCourses()* αλλάζει λίγο:

```
void Student::getCourses( vector<Course::CourseKey>& crsTbl ) const
{
   try
   {
      crsTbl.clear();
      for ( set<Course::CourseKey>::const_iterator it(sCourses.begin());
            it != sCourses.end(); ++it )
         crsTbl.push_back( *it );
   }
   catch( bad_alloc& )
   {  throw StudentXptn( sIdNum, "getCourses", StudentXptn::allocFailed );  }
} // Student::getCourses
```

Τώρα δεν μπορούμε να περάσουμε τους κωδικούς με μια εκχώρηση· πρέπει να διασχίσουμε όλο το σύνολο και να τους περνούμε έναν προς έναν.

Οι δύο δημιουργοί είναι σαν αυτούς που γράψαμε στην §Prj07.1.3, τουλάχιστον οπτικώς. Αλλά

- Στον ερήμην δημιουργό δεν φαίνεται ότι δημιουργείται το (κενό σύνολο) *sCourses*.
- Στον δημιουργο αντιγραφής η "`sCourses = rhs.sCourses`" είναι εκχώρηση με σύνολα.

Να επισημάνουμε ότι ούτε εδώ χρειάζεται δημιουργός αντιγραφής αλλά τον αφήνουμε για να πιάνει και να αλλάζει την (πιθανή) *bad_alloc*. Για τον ίδιο λόγο (και μόνον) αφήνουμε και τον `Student::operator=()`. Όσο για τον καταστροφέα, αυτός δεν έχει τι να κάνει:

```
   ~Student() { };
```

Και προχωρούμε στη *findNdx()*. Μπορούμε να την πάρουμε από αυτήν της §Prj07.1.3 αλλάζοντας το "`vector<Course::CourseKey>::iterator`" σε "`set<Course::CourseKey>::iterator`"; Κατ' αρχήν ναι. Αλλά είναι προτιμότερο να κάνουμε άλλη μια αλλαγή:

```
set<Course::CourseKey>::iterator Student::findNdx( const string& code )
{
   set<Course::CourseKey>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = sCourses.end();
   else
      ndx = sCourses.find( Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Η *sCourses.find()* ψάχνει σε χρόνο $O(log(sCourses.size()))$ αντί για το $O(sCourses.size())$ της αναζήτησης της *std::find()*.

Η *find1Course()* γίνεται, όπως και στην §Prj07.1.3:

```
   bool find1Course( const string& code )
   {  return ( findNdx(code) != sCourses.end() );  };
```

Θα αλλάξουμε όμως τις *add1Course()* και *delete1Course()* αχρηστεύοντας τις (εσωτερικές) *insert1Course()* και *erase1Course()*:

```
typedef pair< set<Course::CourseKey>::iterator, bool > InsRetType;

void Student::add1Course( const Course& oneCourse )
{
   try
   {
      InsRetType rv(sCourses.insert(Course::CourseKey(oneCourse.getCode())) );
```

```
      if ( rv.second )  sWH += oneCourse.getWH();
   }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "insert1Course", StudentXptn::allocFailed );
   }
} // Student::add1Course
```

Να δούμε τι κάναμε εδώ: Στην §26.3.1 είπαμε ότι η μέθοδος *insert()* του *set* (1η μορφή) παίρνει ως όρισμα το κλειδί που θέλουμε να εισαγάγουμε –στην περίπτωσή μας: `Course::CourseKey(oneCourse.getCode())`– και επιστρέφει μια τιμή *pair* (στην περίπτωσή μας: *rv*):

- Το πρώτο μέλος της (*rv*.*first*) είναι προσεγγιστής του τύπου συνόλου που δουλεύουμε (`set<Course::CourseKey>::iterator`) προς το κλειδί όπως βρίσκεται μέσα στο σύνολο.
- Το δεύτερο μέλος της (*rv*.*second*) είναι τιμή `bool` που έχει τιμή `true` αν έγινε η εισαγωγή ή`false` αν δεν έγινε εισαγωγή διότι η τιμή υπήρχε.

Αν το *rv*.*second* είναι `true` προσθέτουμε στις εβδομαδιαίες ώρες του φοιτητή τις εβδομαδιαίες ώρες του μαθήματος.

Παρομοίως, στη *delete1Course()*, αξιοποιούμε την τιμή που επιστρέφει η *erase()* του *set*:

```
void Student::delete1Course( const Course& oneCourse )
{
   size_t erased( sCourses.erase(Course::CourseKey(oneCourse.getCode())) );
   if ( erased > 0 )  sWH -= oneCourse.getWH();
} // Student::delete1Course
```

Φυλάγουμε την τιμή που επιστρέφει η *sCourses*.*erase()* στη μεταβλητή *erased*.

- Αν το κλειδί δεν υπήρχε στο σύνολο τότε η τιμή της γίνεται "`0`".
- Αλλιώς, αν υπήρχε, γίνεται "`1`" και επειδή ισχύει η "`erased > 0`" αφαιρούμε από τις εβδομαδιαίες ώρες του φοιτητή τις εβδομαδιαίες ώρες του μαθήματος.

Η *swap()* παραμένει όπως έγινε στην §Prj07.1.3 αλλά η εντολή "`sCourses.swap(rhs.sCourses)`" ανταλλάσσει τιμές συνόλων.

Στη *save()* υπάρχει μια υποχρεωτική αλλαγή: Η **for** δεν μπορεί να ελέγχεται πια με τον δείκτη του στοιχείου πίνακα· θα ελέγχεται από προσεγγιστή που διατρέχει ολόκληρο σύνολο:

```
   for( set<Course::CourseKey>::const_iterator it(sCourses.begin());
        it != sCourses.end(); ++it )
      bout.write( it->s, Course::cCodeSz );
```

Παρόμοια αλλαγή θα πρέπει να γίνει και στη *display()*.

Στη *load()*, βρίσκουμε τη διαφορά όταν φτάνουμε στην ανάγνωση των κωδικών μαθημάτων:

```
      for ( int k(0); k < noOfCourses; ++k )
      {
         char aKey[Course::cCodeSz];
         bin.read( aKey, Course::cCodeSz );
         tmp.sCourses.insert( Course::CourseKey(aKey) );
      }
```

Λόγω της αλλαγής περιέχοντος, η κλήση της *tmp*.*sCourses*.*insert()* αντικαθιστά την κλήση της *tmp*.*sCourses*.*push_back()*.

Prj07.2.5 Κλάση *StudentCollection*

Είπαμε στην §Prj07.1.1.4 ότι «*Θα κάνουμε την υλοποίηση με "map", απεικόνιση των αριθμών μητρώου (κλειδιά) σε αντικείμενα Student.*» Θα δηλώσουμε:

```
   map< unsigned int, Student > scArr;
```

Ο τύπος κλειδιών είναι "`unsigned int`" και ο τύπος τιμών "`Student`". Θα πεις: αφού το κλειδί είναι ο αριθμός μητρώου και αφού αυτό υπάρχει μέσα στο *Student* (μέλος *sIdNum*)

δεν έχουμε πλεονασμό (που, όπως είναι γνωστό, είναι επικίνδυνος); Έχουμε! Αλλά, θα είμαστε προσεκτικοί στους χειρισμούς μας. Θα κάνουμε εισαγωγές στοιχείων στο scArr με την:

"`scArr[aStudent.getIdNum()] = aStudent`"

ή με την:

"`scArr.insert( make_pair(aStudent.getIdNum(),aStudent) )`"

Σε σχέση με την §Prj07.1.4, δεν έχουμε αλλαγή στη

```
size_t getNOfStudents() const {  return scArr.size();  }
```

αλλά έχουμε αλλαγή στη

```
void StudentCollection::getArr( vector<Student>& stdntTbl ) const
{
   try
   {
      for ( map<unsigned int,Student>::const_iterator it(scArr.begin());
            it != scArr.end();  ++it )
         stdntTbl.push_back( it->second );
   }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                                   StudentCollectionXptn::allocFailed );  }
} // StudentCollection::getArr
```

Ο *it* στη **for** δείχνει ένα ζεύγος (κλειδί, αντικείμενο). Το "`it->second`" είναι το αντικείμενο με τα στοιχεία φοιτητή. Σύγκρινε τη *getArr()* με τη *getCourses()* της *Student*.

Δημιουργός και καταστροφέας δεν διαφέρουν οπτικώς από αυτούς που γράψαμε στην §Prj07.1.4.

Πάμε τώρα στη

```
map< unsigned int, Student >::iterator StudentCollection::findNdx( int aIdNum)
{
   map<unsigned int,Student>::iterator ndx;
   if ( aIdNum <= 0 )
      ndx = scArr.end();
   else
      ndx = scArr.find( aIdNum );
   return ndx;
} // StudentCollection::findNdx
```

Τι δείχνει ο *ndx* και ο προσεγγιστής που επιστρέφει η *findNdx()*; Ένα ζεύγος τύπου `pair<unsigned int,Student>`.

Η *find1Student()* γίνεται όπως στην §Prj07.1.4. Πρόσεξε όμως πώς γίνεται η

```
const Student& StudentCollection::get1Student( int aIdNum ) const
{
   map<unsigned int,Student>::const_iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "get1Student",
                                   StudentCollectionXptn::notFound, aIdNum );
   return ndx->second;
} // StudentCollection::get1Student
```

Σε σχέση με αυτήν που γράψαμε στην §Prj07.1.4, δεν άλλαξε μόνον ο τύπος του *ndx* αλλά και η επιστρεφόμενη τιμή που τώρα είναι "`ndx->second`".

Ο τύπος του *ndx* αλλάζει και στην *add1Student()* αλλά εδώ αλλάζουμε και κάτι άλλο: αχρηστεύουμε την *insert1Student()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx == scArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
```

```
      scArr.insert( make_pair(aStudent.getIdNum(),aStudent) );
      vector<Course::CourseKey> aStCourses;
      aStudent.getCourses( aStCourses );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                     StudentInCourse(aStudent.getIdNum(), aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Παρομοίως, στη *delete1Student()* αχρηστεύουμε την *erase1Student()*:

```
void StudentCollection::delete1Student( int aIdNum )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx != scArr.end() )  // υπάρχει
   {
      if ( (ndx->second).getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      scArr.erase( ndx );
   }
} // StudentCollection::delete1Student
```

Οι ενδιάμεσες μέθοδοι εισαγωγής/διαγραφής μαθημάτων σε έναν φοιτητή γίνονται:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   (ndx->second).add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   (ndx->second).delete1Course( aCourse );
} // StudentCollection::delete1Course
```

Τέλος, για φύλαξη/φόρτωση έχουμε:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   scArr.swap( rhs.scArr );
} // StudentCollection::swap

void StudentCollection::save( ofstream& bout, SIndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   size_t scNOfStudents( scArr.size() );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( map<unsigned int,Student>::const_iterator it(scArr.begin()),
         unsigned int k(0);
         it != scArr.end();  ++it, ++k )
   {
      index[k].sIdNum = it->first;
      index[k].loc = bout.tellp();
      (it->second).save( bout );
   }
   if ( bout.fail() )
```

```
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save

void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.add1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                 StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Φυσικά, θα αλλάξει και η *checkWH()*:

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( map<unsigned int,Student>::const_iterator it(scArr.begin());
         it != scArr.end();  ++it )
   {
      if ( (it->second).getWH() > maxWH )
         log << "student with id num " << (it->second).getIdNum()
             << ": " << (it->second).getWH() << " hours/week"
             << endl;
   } // for
} // StudentCollection::checkWH
```

Prj07.2.6 Κλάση *StudentInCourseCollection*

Εδώ η απεικόνισή μας θα είναι:

```
map< StudentInCourse::SICKey, StudentInCourse > siccArr;
```

Αυτό θα πει ότι θα βάλουμε μια φορά το κλειδί και άλλη μια φορά το κλειδί μαζί με μια τιμή `float`; Θα προτιμούσες δηλαδή να βάλουμε "`map<StudentInCourse::SICKey, float>`"; Αν θέλεις, δοκίμασέ το ως μια (εύκολη) άσκηση. Να έχεις υπόψη σου πάντως ότι – σε πραγματικές συνθήκες– μπορεί να υπάρχουν και άλλα στοιχεία εκτός από έναν βαθμό.

Η παραπάνω δήλωση προϋποθέτει την επιφόρτωση του τελεστή "`<`" για την κλάση *StudentInCourse*. Αντί να κάνουμε αυτό, κυρίως για εκπαιδευτικούς λόγους, θα ορίσουμε το συναρτησοειδές:

```
   struct SICKeyLT
   {
      bool operator()( const SICKey& frst, const SICKey& scnd ) const
      {
         bool fv( false );
         if ( frst.CCode < scnd.CCode )  fv = true;
         else if ( frst.CCode == scnd.CCode )
         {
            if ( frst.sIdNum < scnd.sIdNum )  fv = true;
         }
         return fv;
      }
```

```
   }; // SICKeyLT
```

θα ορίσουμε τη συντομογραφία *SICMap* ως:

```
   typedef map< StudentInCourse::SICKey, StudentInCourse,
                StudentInCourseCollection::SICKeyLT > SICMap;
```

και θα δηλώσουμε:

```
   SICMap siccArr;
```

Πώς θα αποθηκεύονται τα αντικείμενα στην απεικόνιση; Όλα τα αντικείμενα με τον ίδιο κωδικό μαθήματος θα αποθηκεύονται μαζί (κατ' αύξοντα αριθμό μητρώου).

Πού μπαίνουν όλα αυτά; Στην αρχή της περιοχής "**private**":

```
class StudentInCourseCollection
{
public:
// . . .
private:
   struct SICKeyLT
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   typedef map< StudentInCourse::SICKey, StudentInCourse,
                StudentInCourseCollection::SICKeyLT > SICMap;
   SICMap siccArr;
// . . .
}; // StudentInCourseCollection
```

Σημείωση:▶

Ως άσκηση (εύκολη) δοκίμασε και τον άλλο δρόμο: επιφόρτωσε τον τελεστή "**<**" για την κλάση *StudentInCourse*:

```
bool operator<( const StudentInCourse::SICKey& frst,
                const StudentInCourse::SICKey& scnd )
{
   bool fv( false );
   if ( frst.CCode < scnd.CCode )  fv = true;
   else if ( frst.CCode == scnd.CCode )
   {
      if ( frst.sIdNum < scnd.sIdNum )  fv = true;
   }
   return fv;
} // operator<
```

Σύγκρινέ την με την επιφόρτωση του "**()**" στο συναρτησοειδές.◀

Μετά τους παραπάνω ορισμούς και δηλώσεις έχουμε δημιουργό:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection
```

και καταστροφέα:

```
   ~StudentInCourseCollection()  { };
```

Ακόμη έχουμε:

```
   size_t getNOfStudentInCourses() const  {  return siccArr.size();  }

void StudentInCourseCollection::getArr( vector<StudentInCourse>& sicTbl) const
{
   try
   {
      for ( StudentInCourseCollection::SICMap::const_iterator
                                                    it(siccArr.begin());
            it != siccArr.end();  ++it )
         sicTbl.push_back( it->second );
   }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                                   StudentCollectionXptn::allocFailed );  }
```

```
} // StudentInCourseCollection::getArr

StudentInCourseCollection::SICMap::iterator
                 StudentInCourseCollection::findNdx( int aIdNum, string code )
{
   StudentInCourseCollection::SICMap::iterator ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = siccArr.end();
   else
      ndx = siccArr.find( StudentInCourse::SICKey(aIdNum, code) );
   return ndx;
} // StudentInCourseCollection::findNdx
```

Οι μέθοδοι για «1 *StudentInCourse*» γίνονται:

```
   bool find1StudentInCourse( int aIdNum, string code )
   {  return ( findNdx(aIdNum, code) != siccArr.end() );  }

const StudentInCourse&
          StudentInCourseCollection::get1StudentInCourse( int aIdNum,
                                                          string code ) const
{
   StudentInCourseCollection::SICMap::const_iterator
                                                 ndx( findNdx(aIdNum, code) );
   if ( ndx == siccArr.end() )
      throw StudentInCourseCollectionXptn( "get1StudentInCourse",
                                     StudentInCourseCollectionXptn::notFound,
                                     StudentInCourse::SICKey(aIdNum, code) );
   return ndx->second;
} // StudentInCourseCollection::get1StudentInCourse

void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                        string code )
{
   StudentInCourseCollection::SICMap::iterator
                                      ndx( findNdx(aIdNum, code) );
   if ( ndx != siccArr.end() ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn(
                                        "delete1StudentInCourse",
                            StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn(
                                        "delete1StudentInCourse",
                          StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course( code) );
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      siccArr.erase( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse

void StudentInCourseCollection::add1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs )
{
   if ( findNdx( aStdnInCrs.getSIdNum(),
                           aStdnInCrs.getCCode()) == siccArr.end() )
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                             StudentInCourseCollectionXptn::noCrs );
      if ( !(siccPAllCourses->find1Course(aStdnInCrs.getCCode())))
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                          StudentInCourseCollectionXptn::unknownCrs,
                                          aStdnInCrs.getCCode() );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
```

```
                                        StudentInCourseCollectionXptn::noStdnt );
      if ( !(siccPAllStudents->find1Student(aStdnInCrs.getSIdNum())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                    StudentInCourseCollectionXptn::unknownStdn,
                                              aStdnInCrs.getSIdNum() );
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
      try
      {
        siccArr.insert( make_pair(aStdnInCrs.getKey(), aStdnInCrs));
      }
      catch( bad_alloc& )
      {  throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                              StudentInCourseCollectionXptn::allocFailed );  }
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Τέλος, οι *save()*, *load()* και *swapArr()* γίνονται:

```
void StudentInCourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                    StudentInCourseCollectionXptn::fileNotOpen );
   size_t siccNOfStudentInCourses( siccArr.size() );
   bout.write( reinterpret_cast<const char*>(&siccNOfStudentInCourses),
               sizeof(siccNOfStudentInCourses) );
   for ( StudentInCourseCollection::SICMap::const_iterator
                                                        it(siccArr.begin());
         it != siccArr.end();  ++it )
      (it->second).save( bout );
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                    StudentInCourseCollectionXptn::cannotWrite );
} // StudentInCourseCollection::save

void StudentInCourseCollection::load( ifstream& bin )
{
   StudentInCourseCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         StudentInCourse oneStudentInCourse;
         oneStudentInCourse.load( bin );
         tmp.add1StudentInCourse( oneStudentInCourse );
      }
      if ( bin.fail() )
         throw  StudentInCourseCollectionXptn( "load",
                           StudentInCourseCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentInCourseCollection::load

void StudentInCourseCollection::swapArr( StudentInCourseCollection& rhs )
{
   siccArr.swap( rhs.siccArr );
} // StudentInCourseCollection::swap
```

Prj07.2.6.1 Δυο Λόγια για τη *SICKeyLT*

Είπαμε παραπάνω ότι με το συναρτησοειδές που ορίσαμε «*Όλα τα αντικείμενα με τον ίδιο κωδικό μαθήματος θα αποθηκεύονται μαζί (κατ' αύξοντα αριθμό μητρώου).*» Ας το δούμε αυτό.

Βάζουμε στο πρώτο πρόγραμμα, πριν από τη φύλαξη των συλλογών, τις εντολές:

```
   log.open( "sic.txt" );
   vector<StudentInCourse> allSic;
   allEnrollments.getArr( allSic );
   for ( int k(0); k < allSic.size(); ++k )
      allSic[k].display( log );
   log.close();
```

Στο `sic.txt` βλέπουμε τα εξής:

```
2019 ΕΥ0100Ε 0
2099 ΕΥ0100Ε 0
2173 ΕΥ0100Ε 0
2069 ΕΥ0100Θ 0
2077 ΕΥ0100Θ 0
2081 ΕΥ0100Θ 0
2117 ΕΥ0100Θ 0
. . .
2067 ΤΠ0305Θ 0
2240 ΤΠ0305Θ 0
2044 ΤΠ0308Θ 0
2067 ΤΠ0314Ε 0
2059 ΤΠ0314Θ 0
2091 ΤΠ0314Θ 0
2127 ΤΠ0314Θ 0
2153 ΤΠ0314Θ 0
2164 ΤΠ0314Θ 0
2203 ΤΠ0314Θ 0
```

ή αλλιώς:

```
ΕΥ0100Ε
2019
2099
2173
ΕΥ0100Θ
2069
2077
2081
2117
. . .
ΤΠ0305Θ
2067
2240
2044
ΤΠ0314Ε
2067
ΤΠ0314Θ
2059
2091
2127
2153
2164
2203
```

Όπως βλέπεις, εδώ έχουμε τις καταστάσεις εγγεγραμμένων στα μαθήματα που δίνει η γραμματεία στους διδάσκοντες.

Καλό θα είναι να αλλάξεις τη σειρά των συγκρίσεων στο συναρτησοειδές κα να ξαναδοκιμάσεις. Μπορείς να προβλέψεις τι θα πάρεις; Τις δηλώσεις μαθημάτων ανά φοιτητή (όπως υπάρχουν στα αντικείμενα –τύπου *Student*– του μητρώου των φοιτητών).

Prj07.2.7 Πίνακας – Ευρετήριο

Στην §Prj07.2.1.5 είπαμε σχετικώς με τον πίνακα-ευρετήριο «*Δεν θα αλλάξουμε περιέχον αλλά θα ταξινομήσουμε τον πίνακα (κατ' αύξοντα αριθμό μητρώου) ώστε να υπάρχει δυνατότητα δυαδικής αναζήτησης από τα προγράμματα εφαρμογής που τον χρησιμοποιούν.*»

Ε, δεν θα χρειαστεί να κουραστούμε: ο πίνακας *είναι* ταξινομημένος όπως τον θέλουμε! Πράγματι, στην §Prj07.2.5, για την κλάση *StudentCollection*, δηλώσαμε περιέχον για τη συλλογή:

```
map< unsigned int, Student > scArr;
```

Το εννοούμενο τρίτο όρισμα του περιέχοντος είναι "`less<unsigned int>`". Στο scArr εισάγουμε στοιχεία με την

"`scArr.insert( make_pair(aStudent.getIdNum(),aStudent) )`"

και τέλος, για τη φύλαξη στο αρχείο, διασχίζουμε το περιέχον με τη:

```
for (map<unsigned int,Student>::const_iterator it(scArr.begin()),
     unsigned int k(0);
     it != scArr.end();  ++it, ++k )
{
   index[k].sIdNum = it->first;
   index[k].loc = bout.tellp();
   (it->second).save( bout );
}
```

Και πού μπαίνει η δυαδική αναζήτηση; Στο δεύτερο πρόγραμμα, αν θέλουμε. Κατ' αρχήν το δεύτερο πρόγραμμα –όπως και το πρώτο– δουλεύει μια χαρά χωρίς να χρειάζεται τροποποίηση.

Αν θέλεις να χρησιμοποιήσεις την (*std*::)*lower_bound()* για να εκμεταλλευθείς την ταξινόμηση του πίνακα θα πρέπει να κάνεις τα εξής:

- Ορίζεις την

```
bool comp( const SIndexEntry& frst, const SIndexEntry& scnd )
{  return ( frst.sIdNum < scnd.sIdNum );  } // comp
```

- Αντικαθιστάς τις:

```
int ndx( linSearch(sIndex, ndxSz, 0, ndxSz-1,
         SIndexEntry(idNum)) );
if ( ndx < 0 )
```

με τις

```
SIndexEntry* ndx( lower_bound(sIndex, sIndex+ndxSz,
                  SIndexEntry(idNum), comp) );
if ( ndx->sIdNum != idNum )
```

και τη

```
bin.seekg( sIndex[ndx].loc );
```

με τη

```
bin.seekg( ndx->loc );
```

Φυσικά, δεν θα ξεχάσεις να βάλεις και την "`#include <algorithm>`".

Prj07.3 Τι Είδαμε στα Δύο Παραδείγματα

Σύγκρινε τους δύο τρόπους που γράψαμε τις κλάσεις μας για να καταλάβεις ότι η άκριτη αποκλειστική χρήση του *vector* μπορεί να σου ευκολύνει κάπως τη δουλειά αλλά δεν σου δίνει το καλύτερο αποτέλεσμα.

Είναι ουσιώδες να κατανοήσεις καλά το πρόβλημα που έχεις να λύσεις και να επιλέξεις το καταλληλότερο περιέχον για την κάθε συλλογή δεδομένων.

Πάντως θα πρέπει να επισημάνουμε το πόσο καλό εργαλείο είναι το *vector* για την τεχνική RAII όταν ο πόρος που χειρίζεσαι είναι δυναμική μνήμη. Δηλαδή, τα άλλα περιέχοντα δεν είναι κατάλληλα; Μια χαρά είναι και τα άλλα αφού έχουν τους καταστροφείς τους αλλά το *vector* είναι το πιο απλό στον χειρισμό.

Τέλος, πρόσεξε ότι τα δικά μας εργαλεία (του `MyTmpltLib.h`) αχρηστεύθηκαν αφού η STL μας δίνει καλύτερες λύσεις.

project

8

Προβλήματα για Λύση

Περιεχόμενα:

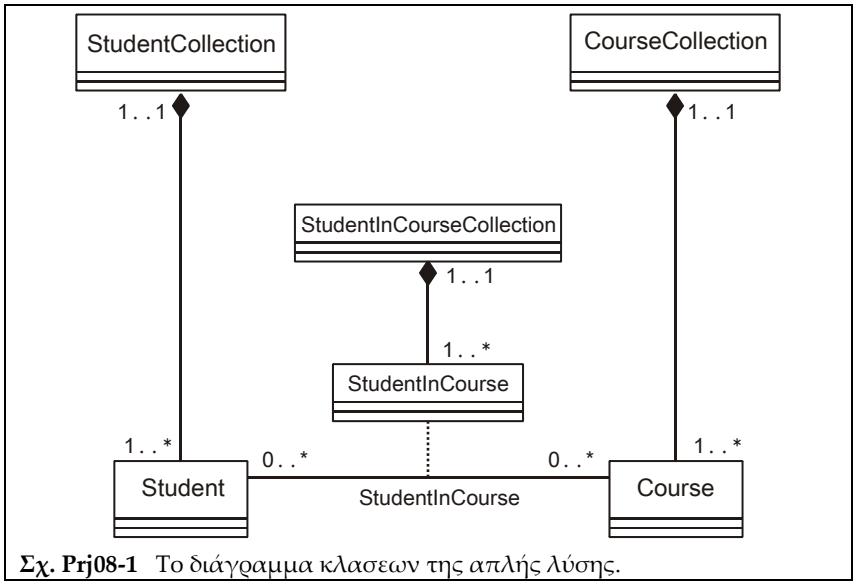
Εισαγωγικό Σημείωμα:
Στο Project αυτό δίνουμε προβλήματα για να τα λύσεις εσύ! Απόλαυσε την εργασία σου!

Prj08.1 Φοιτητές και Μαθήματα: Η Απλή Λύση

Αφού είδαμε αυτά που θέλαμε να δούμε από προγραμματισμό ας δούμε τώρα τη σωστή λύση για το πρόβλημα «Φοιτητές και Μαθήματα» που είναι και απλούστατη. Στην πραγματικότητα πρόκειται για το αρχικό σχέδιο (Project 3) αλλά με τους πίνακες ως αντικείμενα κλάσεων (*CourseCollection* κλπ). Δες το Σχ. Prj08-1.

Στην §Prj04.13 λέγαμε:

- *«Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος*



Σχ. Prj08-1 Το διάγραμμα κλασεων της απλής λύσης.

φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
```

- *Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:*

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)»

Παίρνουμε λοιπόν το διάγραμμα κλάσεων του Σχ. Prj08-1 όπου *Student* και *Course* είναι τα σταθερά μέρη που περιγράψαμε πιο πάνω. Και τα μεταβλητά τα «πετάμε»; Ναι! Δεν μας χρειάζονται. Και όταν θέλουμε να δείξουμε τα μαθήματα που γράφηκε ένας φοιτητής; Τα βρίσκουμε από το *allEnrollments*! Οι *Student*::*getWH()*, *Student*::*getNoOfCourses()*, *Student*::*getCourses()*, *Course*::*getNoOfStudents()* θα υλοποιηθούν με μέτρημα καταχωρίσεων στο *allEnrollments*.

Λύσε λοιπόν το πρόβλημα χρησιμοποιώντας όσο πιο πολύ μπορείς την STL. Χρησιμοποίησε ό,τι θέλεις από τις λύσεις που δώσαμε πιο πριν αλλά να θυμάσαι ότι σε κάποιες περιπτώσεις, για να δείξουμε ορισμένα πράγματα, δεν χρησιμοποιήσαμε τις απλούστρες επιλογές.

Prj08.2 Αεροπλάνα και Πιλοτοι

Το πληροφοριακό σύστημα της Karakaxa Airlines (KA) περιλαμβάνει πληροφορίες για προσωπικό, αεροπλάνα, πτήσεις και δρομολόγια.

Για κάθε πιλότο έχουμε αντικείμενο της μορφής:

```
class Pilot
{
public:
// . . .
private:
   unsigned int   piIdNum;         // αριθμός μητρώου
   char           piSurname[20];   // επώνυμο
   char           piFirstName[16]; // όνομα
   ?????          piExper;         // εμπειρία σε αεροσκάφη
}; // Pilot
```

όπου *piExper* κάποιο περιέχον με στοιχεία τύπου[1]

```
struct ExperAircraft
{
   char   xaAircraftType[20]; // τύπος αεροσκάφους
   double xaFlightTime;       // ώρες πτήσης
}; // ExperAircraft
```

Ιδιαίτερο ενδιαφέρον έχει η εμπειρία του κάθε πιλότου σε διάφορους τύπους αεροσκαφών. Κάθε στοιχείο της *piExper* μας λέει πόσες ώρες πτήσης (`xaFlightTime`) έχει ο πιλότος στον συγκεκριμένο τύπο αεροσκάφους (`xaAircraftType`).

Εφοδίασε την κλάση *Pilot* με μεθόδους *save()* και *load()* που φυλάγουν τα δεδομένα ενός πιλότου σε ένα μη-μορφοποιημένο (binary) αρχείο και το ξαναφορτώνουν από αυτό.

Συμπλήρωσε την κλάση *Pilot* με ό,τι (μέλος, μέθοδο) πιστεύεις ότι χρειάζεται (μέλη μπορείς να προσθέσεις αλλά όχι να αφαιρέσεις).

Το αρχείο text `pilots.txt` έχει γραμμές σαν και τις παρακάτω:

```
2105\tΚΡΗΤΙΚΟΣ\tΕΛΕΥΘΕΡΙΟΣ\n
2088\tΚΥΛΙΔΗΣ\tΔΗΜΟΣΘΕΝΗΣ\n
```

(πιλότος με αριθμό μητρώου 2105, επώνυμο ΚΡΗΤΙΚΟΣ και όνομα ΕΛΕΥΘΕΡΙΟΣ· πιλότος με αριθμό μητρώου 2088, επώνυμο ΚΥΛΙΔΗΣ και όνομα ΔΗΜΟΣΘΕΝΗΣ)

Το πρώτο πρόγραμμα που θα γράψεις θα διαβάζει το αρχείο `pilots.txt` και θα απόθηκεύει τα στοιχεία όλων των πιλότων σε περιέχον

```
????? allPilots;
```

Για κάθε αεροπλάνο έχουμε αντικείμενο της μορφής:

```
class AircraftType
{
public:
// . . .
private:
   char         atType[20];         // τύπος αεροσκάφους
   unsigned int atNoOfSeats;        // αριθμός θέσεων επιβατών
   double       atMaxTakeOffLoad;   // μέγιστο βάρος απογείωσης σε kgr
   ???????      atExperPilots;      // πιλότοι με εμπειρία
}; // AircraftType
```

όπου *atExperPilots* περιέχον με στοιχεία τύπου[2]

```
struct ExperPilot
{
   unsigned int xpIdNum;      // αρθμός μητρώου πιλότου
   double       xpFlightTime; // ώρες πτήσης
}; // ExperPilot
```

Ιδιαίτερο ενδιαφέρον έχουν οι πιλότοι της εταιρείας που μπορούν να πετάξουν τον συγκεκριμένο τύπο αεροσκάφους. Αυτοί συνοψίζονται στο περιέχον *atExperPilots*. Κάθε στοιχείο του μας λέει πόσες ώρες πτήσης (`xpFlightTime`) έχει στον συγκεκριμένο τύπο αεροσκάφους ο πιλότος με αριθμό μητρώου `xpIdNum`.

Εφοδίασε την κλάση *AircraftType* με μεθόδους *save()* και *load()* που φυλάγουν τα δεδομένα ενός αεροσκάφους σε μη-μορφοποιημένο αρχείο και το ξαναφορτώνουν από αυτό.

Το αρχείο text `aircraftTypes.txt` έχει γραμμές σαν και τις παρακάτω:

```
AIRBUS A340-300\t295\t275000\n
ATR - 42 - 320\t50\t16700 \n
```

(αεροπλάνο τύπου AIRBUS A340-300, με 295 θέσεις επιβατών και μέγιστο βάρος απόγείωσης 275000 kgr.)

Μετά τους πιλότους, το πρώτο πρόγραμμα θα διαβάζει το αρχείο `aircraftTypes.txt` και θα αποθηκεύει τα στοιχεία όλων των πιλότων σε περιέχον

---

[1] Καλύτερο θα ήταν να δηλώσουμε (`public`) την *ExperAircraft* μέσα στην κλάση.

[2] Καλύτερο θα ήταν να δηλώσουμε (`public`) την *ExperPilot* μέσα στην κλάση.

```
?????? allAircraftTypes;
```

Στη συνέχεια θα διαβάζει το αρχείο `flightHours.txt` και θα συμπληρώνει όπου χρειάζεται τα στοιχεία των πιλότων (θα γεμίζει τα περιέχοντα *piExper* των αντικειμένων που παριστάνουν τους πιλότους).

Το αρχείο text `flightHours.txt` έχει γραμμές σαν και τις παρακάτω:

```
2040\tAIRBUS A319\t21.7\n
2145\tATR - 42 - 320\t1374.2\n
```

που σημαίνει ότι ο πιλότος με αριθμό μητρώου 2040, έχει 21.7 ώρες πτήσης με αεροπλάνα τύπου AIRBUS A319, ενώ ο πιλότος με αριθμό μητρώου 2145, έχει 1374.2 ώρες πτήσης με αεροπλάνα τύπου ATR - 42 – 320.

Τέλος, θα φυλάγει σε μη-μορφοποιημένα αρχεία

- τα ενημερωμένα στοιχεία των πιλότων στο αρχείο με όνομα `pilotExp.dta` και
- τα ενημερωμένα στοιχεία των τύπων αεροσκαφών στο αρχείο `aircraftTypes.dta`.

Ένα δεύτερο πρόγραμμα θα φορτώνει καταλλήλως τα περιεχόμενα των `pilotExp.dta` και `aircraftTypes.dta` και θα ενημερώνει τα περιεχόμενά τους από τα δεδομένα των πτήσεων (που ολοκληρώνονται) που θα διαβάζει από το πληκτρολόγιο.

Για να απλουστεύσουμε τα πράγματα, για κάθε πτήση θα διαβάζουμε μόνον:

- διάρκεια πτήσης (πρώτα λεπτά),
- τύπος αεροσκάφους (πρέπει να υπάρχει στο *allAircraftTypes*),
- αριθμός μητρώου κυβερνήτη (πρέπει να υπάρχει στο *allPilots*),
- αριθμός μητρώου συγκυβερνήτη (πρέπει να υπάρχει στο *allPilots*).[3]

Προσοχή! Για να γίνουν δεκτά τα στοιχεία πτήσης θα πρέπει: κυβερνήτης και συγκυβερνήτης να έχουν τον τύπο αεροσκάφους της πτήσης σε αυτά «που ξέρουν» (έστω και με 0 ώρες). Δηλαδή, η διαδικασία ενημέρωσης του συστήματος ότι «*εκπαιδεύτηκα και σε άλλο αεροπλάνο*» δεν είναι αυτόματη.

Αν θέλεις δώσε (στο δεύτερο πρόγραμμα) τη δυνατότητα να δείχνει όλα τα στοιχεία ενός πιλότου ή/και ενός τύπου αεροσκάφους.

Παρατηρήσεις:▶

1. Λύσε το πρόβλημα χρησιμοποιώντας όσο πιο πολύ μπορείς την STL.
2. Προσοχή στον πλεονασμό, διότι υπάρχει αρκετός: Τα ίδια δεδομένα υπάρχουν στους πιλότους και στους τύπους αεροσκαφών.
3. Θέλεις να δοκιμάσεις κάτι πιο απλό (σαν την Prj08.1); Δοκίμασέ το, αλλά προσοχή: Δεν μπορείς να αφαιρέσεις από τα στοιχεία του πιλότου τους τύπους αεροσκαφών που μπορεί να κυβερνήσει.◀

Prj08.3 Αρχεία jpeg

Prj08.3.1 Ο Κατάλογος

Μια εταιρεία πουλάει διάφορα προϊόντα και θέλει να κάνει έναν ψηφιακό κατάλογο προϊόντων για τους πελάτες της. Κάθε προϊόν θα αποθηκεύεται ως αντικείμενο κλάσης

```
class Product
{
public:
// . . .
private:
```

[3] Κυβερνήτης και συγκυβερνήτης αυξάνουν κατά «διάρκεια πτήσης» τις ώρες πτήσης που έχουν με τον τύπο αεροσκάφους.

```
   char           prCode[20];     // κωδικός προϊόντος
   char           prDescr[100];   // περιγραφή
   double         prL, prW, prH;  // Διαστάσεις (μήκος, πλάτος, ύψος) σε cm
   unsigned char* prJpeg;         // φωτογραφία σε jpeg
   unsigned int   prJpegSz;       // μεγεθος του jpeg σε bytes
   char           prJpegPath[130];// πλήρης διαδρομή και όνομα αρχείου jpeg
   void displayJpeg( ostream& tout,
                     const unsigned char* jpegImg,
                     int jpegSz ) { };
}; // Product
```

Ειδικώς για τα *prJpeg*, *prJpegSz*, *prJpegPath* είναι δεκτές οι εξής καταστάσεις:

- *prJpeg* `!=` 0 (`NULL`), δηλαδή υπάρχει φωτογραφία, *prJpegSz* `>` 0 (μας δίνει το μέγεθος της φωτογραφίας σε bytes), *prJpegPath* έχει ως τιμή το όνομα αρχείου (με πλήρη διαδρομή) όπου θα αποθηκευτεί η φωτογραφία με το επόμενο μήνυμα *save*.
- *prJpeg* `==` 0 (δεν υπάρχει φωτογραφία), *prJpegSz* `==` 0 και *prJpegPath* `== ""`.

α) Γράψε όλες τις μεθόδους που νομίζεις ότι χρειάζεται η κλάση.

β) Πέρα από όποιες άλλες μεθόδους θεωρήσεις απαραίτητες, η κλάση θα πρέπει να έχει και τις εξής:

```
   void save( ostream& bout ) const;
```

που θα φυλάγει τις τιμές μελών ενός αντικειμένου κλάσης *Product* σε ένα μη-μορφοποιημένο αρχείο συνδεμένο με το ρεύμα *bout*. Προσοχή όμως: η φωτογραφία (αν υπάρχει) δεν φυλάγεται μέσω του ρεύματος *bout* αλλά σε άλλο αρχείο (binary) του οποίου το όνομα (με πλήρη διαδρομή) υπάρχει στο μελος *prJpegPath*.

```
   void load( istream& bin );
```

που θα φορτώνει τις τιμές μελών ενός αντικειμένου κλάσης *Product* από ένα αρχείο binary συνδεμένο με το ρεύμα *bout*. Μετά τη φόρτωση τα *prJpeg*, *prJpegSz* θεωρείται ότι έχουν τιμή 0. Αν *prJpegPath* `!= ""` τότε φορτώνεται και η φωτογραφία από το αρχείο που το όνομά του υπάρχει στο *prJpegPath*.

```
   void loadJpeg( string jPath );
```

που τροφοδοτείται με το όνομα αρχείου (με πλήρη διαδρομή) που περιέχει την εικόνα και τη φορτώνει σε δυναμικό πίνακα που δείχνει η *prJpeg*. Αν η φόρτωση γίνει επιτυχώς ενημερώνονται καταλλήλως και τα *prJpegSz*, *prJpegPath*. Αν η φόρτωση αποτύχει παραμένουν τα πάντα όπως ήταν.

Όπως φαίνεται, χειριζόμαστε μια εικόνα jpeg σαν μια ακολουθία bytes. Όταν δίνουμε *save* το αντικείμενο θα αποθηκευτεί στο αρχείο που είναι συνδεμένο με το ρεύμα *bout* αλλά η εικόνα θα αποθηκευτεί σε ξεχωριστό αρχείο που καθορίζεται στο *prJpegPath*.

γ) Στο αρχείο `products.dta` (binary) υπάρχουν μερικά προϊόντα σε αντικείμενα της παραπάνω κλάσης. Γράψε μέθοδο *display* που θα δείχνει όλα τα στοιχεία ενός προϊόντος σε ένα αρχείο text. Όλα; Και την εικόνα; Καλά... Για την εικόνα θα καλείς τη βοηθητική συνάρτηση *displayJpeg* που είναι «άδεια» (`{ }`)!

Χρησιμοποίησε

- τη *load()* για να διαβάσεις τα στοιχεία από το αρχείο και
- τη *display()* για να βγάλεις τα στοιχεία όλων των προϊόντων στην οθόνη.

δ) Σε κάποιο προϊόν βάλαμε λάθος εικόνα. Βάλαμε ως αρχείο εικόνας το `rondinejr.jpg` αντί για `Product10.jpg`. Βρες το προϊόν και διόρθωσέ το με χρήση της *loadJpeg()*. Στη συνέχεια, χρησιμοποιώντας τη *save()* φύλαξε όλα τα (σωστά) στοιχεία.

Prj08.3.2 Η Σύνθεση

Στην §Prj08.3.1 είδαμε μια κλάση χρήσιμη σε μια εταιρεία για τον ψηφιακό κατάλογο των προϊόντων της. Η εταιρεία θέλει και μια κλάση

```
class ProductConstr
{
// . . .
}; // ProductConstr
```

για την κατασκευή των προϊόντων που παράγει η ίδια.

Η *ProductConstr* έχει τα χαρακτηριστικά και τη συμπεριφορά (μεθόδους) της *Product* και περιέχει επιπλέον όλα τα εξαρτήματα (συνιστώσες) που απαιτούνται για την κατασκευή μιας μονάδας του προϊόντος. Τα εξαρτήματα περιγράφονται σε ζεύγη (κωδικός συνιστώσας, ποσότητα), π.χ. (CRW0756, 6), (CXL0718, 10).

Με βάση την *Product* γράψε την κλάση *ProductConstr*. Η νέα κλάση θα έχει όλες τις ιδιότητες της *Product*. Φυσικά οι *save*, *load* και *display* θα αλλάξουν και επιπλέον: Σε έναν (δυναμικό) πίνακα του κάθε αντικειμένου κρατούμε τα ζεύγη (κωδικός συνιστώσας, ποσότητα) του προϊόντος. Θα χρειαστούμε και μεθόδους για τη διαχείριση του πίνακα.

Στο αρχείο `products.dta` (το ξέρεις από την προηγούμενη άσκηση) υπάρχουν τα στοιχεία των προϊόντων (αντικείμενα κλάσης *Product*). Στο αρχείο text `prodComp.txt` υπάρχουν τα στοιχεία σύνθεσης των προϊόντων. Σε κάθε γραμμή υπάρχουν τριάδες:

Κωδικός προϊόντος `tab` κωδικός εξαρτήματος `tab` ποσότητα

Γράψε πρόγραμμα που θα διαβάζει τα δύο αρχεία και θα δημιουργεί τα αντικείμενα κλάσης *ProductConstr*, ένα για κάθε προϊόν. Προσοχή! Το δεύτερο αρχείο δεν έχει ελεγχθεί. Μπορεί λοιπόν να βρεις, ας πούμε, μια γραμμή

```
LED-R  NQE4868      9
```

και παρακάτω:

```
LED-R  NQE4868      8
```

Στη συνέχεια θα δίνει τη δυνατότητα στον χρήστη να κάνει διορθώσεις. Δηλαδή θα δείχνει τη σύνθεση του προϊόντος που ζητάει ο χρήστης και θα τουδίνει τη δυνατότητα α) να σβήσει κάποιο εξάρτημα β) να εισαγάγει ένα νέο εξάρτημα με την αντίστοιχη ποσότητα γ) να αλλάξει την ποσότητα κάποιου εξαρτήματος που ήδη υπάρχει.

Τέλος, θα φυλάγει όλα τα αντικείμενα σε ένα αρχείο binary δημιουργώντας και το κατάλληλο ευρετήριο για να είναι δυνατή η ανάκτηση.

Prj08.4 Τραπεζικά

Μια τράπεζα για να μηχανοργανώσει τις εργασίες της χρειάζεται κάποιες κλάσεις. Ας δούμε μερικές από αυτές.

Κλάση *Person* που τα αντικείμενά της έχουν τα στοιχεία ενός ανθρώπου που έχει (ή είχε στο παρελθόν) κάποια σχέση με την τράπεζα:

- ΑΔΤ, Αριθμός Δελτίου Ταυτότητας (δύο κεφαλαία ελληνικά γράμματα και έξη δεκαδικά ψηφία από τα οποία το πρώτο δεν είναι 0, π.χ. `KM209084`),
- επώνυμο,
- όνομα,
- (ένας) αρ. τηλεφώνου (10ψήφιος).

Στο αρχείο `persons.txt` υπάρχουν τέτοια στοιχεία, μια γραμμή για κάθε άνθρωπο, πχ.:

```
KM209084\tΓΑΡΥΦΑΛΛΟΣ\tΞΕΝΟΦΩΝ\t6997208 489\n
```

Κλάση *Loan* που τα αντικείμενά της έχουν στοιχεία δανείων:

- κωδ. δανείου (αποτελείται από δύο πενταψήφιους ορμαθούς, π.χ.: `22555 70105`), είδος δανείου (στεγαστικό, επαγγελματικό κλπ) ακέραιος από 1 μέχρι 6,
- ΑΔΤ (του μοναδικού) δανειολήπτη,
- διάρκεια σε μήνες,

- ποσό.

Στο αρχείο `loans.txt` υπάρχουν γραμμές με τέτοια στοιχεία, π.χ.:

```
22555 70105\t5\t467\t190321
```

που σημαίνει: κωδ. δανείου: **22555 70105**, τύπος δανείου **5** (ας πούμε στεγαστικό), διάρκεια δανείου **467** μήνες (κάπου 39 χρόνια), συνολικό ποσό **190321**€.

Κλάση *Account* που τα αντικείμενά της είναι στοιχεία λογαριασμών:

- κωδικός IBAN[4] (για την Ελλάδα αποτελείται 27 χαρακτήρες οι δύο πρώτοι είναι GR, οι δύο επόμενοι είναι ψηφία ελέγχου και οι υπόλοιποι (Basic Bank Account Number) έχουν σχέση με την τράπεζα, το υποκατάστημα, τον λογαριασμό),
- είδος λογαριασμού (ταμιευτηρίου, όψεως κλπ) ακέραιος από 1 μέχρι 5,
- υπόλοιπο,
- ΑΔΤ όλων των δικαιούχων του λογαριασμού.

Στο αρχείο `accounts.txt` υπάρχουν γραμμές με τέτοια στοιχεία[5], π.χ.:

```
GR0133444402010136120121721\t4\t14110
```

που σημαίνει: IBAN λογαριασμού: **GR0133444402010136120121721**, τύπος λογαριασμού **4** (ας πούμε ταμιευτήριο), υπόλοιπο **14110**€.

Μια κλάση *Customer* που θα κληρονομεί την *Person* και θα έχει επί πλέον τους κωδικούς όλων των δανείων από την και τους IBAN όλων των λογαριασμών στην τράπεζα του συγκεκριμένου ανθρώπου (πελάτη).

Το αρχείο `customers.txt` έχει γραμμές όπως:

```
Λ\tGR2333407023003420332160230\tTY212003
Δ\t50281 33128\tOP215070
```

Η πρώτη σημαίνει: ο λογαριασμός (**Λ**): **GR2333407023003420332160230**, έχει δικαιούχο τον πελάτη με ΑΔΤ **TY212003** (μπορεί να έχει και άλλους). Η δεύτερη σημαίνει ότι το δάνειο (**Δ**) **50281 33128** έχει (μοναδικό) δανειολήπτη τον πελάτη με ΑΔΤ **OP215070**.

Α) Υλοποίησε τις κλάσεις.

Β) Γράψε πρόγραμμα που θα διαβάζει τα αρχεία και θα αποθηκεύει καταλλήλως τα περιεχόμενά τους. Προσοχή! Κατά την ανάγνωση να γίνονται έλεγχοι: τα αρχεία είναι text και δεν είναι ελεγμένα.

Γ) Το πρόγραμμα θα φυλάγει καταλλήλως τα δεδομένα σε αρχεία binary, ώστε να είναι δυνατή η (σωστή) επαναφόρτωσή τους.

---

[4] International Bank Account Number. Για περισσότερα διάβασε το «*Ερωτήσεις και Απαντήσεις για τον IBAN*» της Ένωσης Ελληνικών Τραπεζών (http://www.hba.gr/iban/IBAN-gr.pdf).

[5] Τα ψηφία ελέγχου στους IBAN του αρχείου είναι λάθος (βαλμένα στην τύχη). Όποιος/α θέλει ας ψάξει στο διαδίκτυο να βρει πώς υπολογίζονται με τη μέθοδο MOD 97-10 και ας τα διορθώσει.

βιβλιογραφία

Το (Stroustrup 1997) είναι το πρώτο βιβλίο που θα δούμε. Σίγουρα δεν θα ξεκινήσεις να μαθαίνεις προγραμματισμό και C++ από αυτό. Αλλά, εκτός του ότι περιέχει ολόκληρο το πρότυπο της C++ είναι πολύτιμο διότι περιλαμβάνει:

- μια εξαιρετική παρουσίαση της γλώσσας από τον δημιουργό της,
- συμβουλές για τη χρήση (καλύτερα την αξιοποίηση) των χαρακτηριστικών της,
- συμβουλές για καλό προγραμματισμό από έναν καλό προγραμματιστή.

Η προηγούμενη (2η) έκδοση έχει μεταφραστεί στα ελληνικά.

Πριν δούμε άλλα βιβλία για τη C++ θα πρέπει να σταθούμε σε δύο βιβλία για τη C. Το πρώτο, (Kernighan & Ritchie 1988), είναι πιθανότατα το πιο πολυδιαβασμένο βιβλίο προγραμματισμού διεθνώς. Το δεύτερο, (Kernighan 1996), συνεχίζει την επιτυχία του προηγούμενου. Μπορεί τα βιβλία να είναι για C αλλά, διαβάζοντάς τα, θα μάθεις πολλά πολύτιμα πράγματα για προγραμματισμό *και για C++*. Το πρώτο έχει μεταφραστεί και στα ελληνικά.

Το (Lippman & Lajoie 2013) ήταν, για χρόνια, το πιο διαδεδομένο διδακτικό εγχειρίδιο όπου διδάσκεται C++. Η 5η έκδοση καλύπτει και το C++11. Είναι εξαιρετικό αλλά δεν θα ξεκινήσεις από αυτό.

Πιο εύκολο για ξεκίνημα είναι το (Deitel & Deitel 2013)· η 6η έκδοση έχει μεταφραστεί στα ελληνικά (Μ. Γκιούρδας), η 9η έκδοση καλύπτει και το C++11 και η 8η (αγγλικά) υπάρχει δωρεάν στο διαδίκτυο.[1] Η Deitel προσφέρει δωρεάν στο διαδίκτυο πολλά υλικά καθώς και μαθήματα.[2]

Να αναφέρουμε και τα (Jamsa 1994), (Liberty 1995), (Lafore 2001)· οι εκδόσεις τους στα αγγλικά ήταν best-sellers και υπάρχουν ελληνικές μεταφράσεις. Στα ελληνικά είναι και το (Σταμούλης 1992).

Αν ξέρεις αγγλικά οι επιλογές σου είναι πολύ περισσότερες:

Τα (Hanly 1997) και (Mansfield & Antonakos 1997) είναι εύκολα, εισαγωγικά.

Τα best-sellers της εποχής είναι: (Schildt 1997), (Schildt 1998), (Satir & Brown 1995), (Yaroshenko 1995). Αλλά προσοχή: όλα, εκτός από το τελευταίο, απευθύνονται σε ανθρώπους που ξέρουν ήδη προγραμματισμό ή –ακόμη περισσότερο– ξέρουν ήδη C.

Το (Feuer 1982) είναι ένα βιβλίο με ασκήσεις στη C (*όχι C++*), που κάποτε «πούλησε» πολύ.

Για όσους έχουν πιο συγκεκριμένα ενδιαφέροντα:

Για περισσότερα στη *Μαθηματική Λογική* σε παραπέμπουμε στα (Τζουβάρας 1987**), (**Κυριακόπουλος 1977**), (**Manna & Waldinger 1985).

Για γλώσσες προδιαγραφών: (Andrews & Ince 1991), (Diller 1990), (Jones 1990), (Spivey 1988).

Θα πρέπει να επιστήσουμε την προσοχή σου σε τρεις αναφορές σε διαδικτυακούς τόπους:

- (ELLEMTEL, 1992) «*Programming in C++: Rules and Recommendations*»: Πρόκειται για προγραμματιστικούς κανόνες που πρέπει να ακολουθούνται από τους προγραμματιστές

[1] http://aicitel.files.wordpress.com/2011/12/deitel_2012.pdf

[2] http://www.deitel.com/FreeTutorials/FreeTutorialIndex/tabid/1575/Default.aspx#CPLUSPLUS

της ELLEMTEL, μιας εταιρείας ανάπτυξης ηλεκτρονικών συστημάτων για την Ericsson και την Televerket.

- (Lockheed Martin, 2005) «*Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*»: Έχει κανόνες ανάπτυξης λογισμικού σε C++ για ένα σύστημα με κρισιμότητα ασφάλειας (safety critical), το σύστημα ενός νέου μαχητικού αεροπλάνου.
- (CERT 2009) «*CERT C++ Secure Coding Standard*»: Πρόκειται για μια έκδοση του Software Engineering Institute (SEI) με κανόνες για να γράφεις ασφαλή προγράμματα σε C++. Δεν είναι περίεργο ότι έχει αρκετές αναφορές στους δύο προηγούμενους ιστότοπους.

Πέρα από τα παραπάνω, στο διαδίκτυο θα βρεις πολλούς τόπους με ερωταποκρίσεις σχετικώς με τη C++ και τον προγραμματισμό.[3] Να επισημάνουμε

- Τον «κλασικό» (Cline 1999)· μπορείς να βρεις και την έντυπη μορφή του (Cline, Lomow, Girou 2000).
- Τον «Guru of the Week» (Sutter 1998) που εκδόθηκε ως *Exceptional C++* (Sutter 1999) που υπάρχει και ως pdf.

Για τις διαφορές του C++11 (και του C++14) από το C++03 παραπέμπουμε κατ' αρχάς στο (Stroustrup, B. 2011). Δες ακόμη το (Kalev 2011).

Αν σε ενδιαφέρουν τα πρότυπα θα βρεις:

- Το C++03 στο (ISO 2003).
- Το C++11 στο (ISO 2011). Για να το πάρεις θα πρέπει να πληρώσεις.
- Στο (ISO/IEC JTC1 SC22 WG21 N3337 2012) θα βρεις δωρεάν το έγγραφο εργασίας που είναι σχεδόν ίδιο με το C++11.
- Στο (ISO/IEC JTC1 SC22 WG21 N3797 2014) θα βρεις το έγγραφο εργασίας που είναι σχεδόν ίδιο με το C++14.

Θα πρέπει να τονίσουμε πάντως ότι τα πρότυπα δεν θα σε βοηθήσουν και πολύ.

Κάππος, Δ. 1962: *Μαθήματα Αναλύσεως - Απειροστικός Λογισμός* τ. Α', Αθήνα.

Κατσαργύρη, Β., Κ. Μεντή, Γ. Παντελίδη, Κ. Σούρλα, 1992: *Μαθηματικά Γ΄ Λυκείου, Ανάλυση*, Ο.Ε.Δ.Β., Αθήνα.

Κυριακόπουλου, Α. Κ. 1977: *Μαθηματική Λογική μετά Μεθόδων Αποδείξεως εις τα Μαθηματικά*, 2η Έκδ., Παπαδημητρόπουλος, Αθήνα.

Σταμούλη, Α. 1992: *Το Βιβλίο της C++*, Learning Plan, Θεσσαλονίκη.

Τζουβάρα, Α. 1987: *Στοιχεία Μαθηματικής Λογικής*, Θεσσαλονίκη.

Abrahams, D. 2001: *Exception-Safety in Generic Components*, retrieved on 28.12.2012 from: `http://www.boost.org/community/exception_safety.html`

Abrahams, D. 2010: *Error and Exception Handling*, retrieved on 28.12.2012 from: `http://www.boost.org/community/error_handling.html`

Alexandrescu, A. 2001: *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley, Reading Ma, US.

Andrews, D. & D. Ince 1991: *Practical Formal Methods with VDM*, McGraw-Hill, London, UK.

Bohm, C. & G. Jacopini 1966: "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *CACM* v.**9**, pp. 366-71.

CERT, 2009: *CERT C++ Secure Coding Standard*, retrieved on 01.06.2009 from: `http://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637`.

[3] Γενικώς –και όχι μόνον για προγραμματισμό– θα πρέπει να υπενθυμίσουμε ότι υπάρχουν ΔΩΡΕΑΝ τα υλικά του OpenCourseWare (OCW) από τα μεγαλύτερα πανεπιστήμια του κόσμου! Στα αγγλικά; Ναι, στα αγγλικά…

Cline, M. 1999: *C++ FAQ*, retrieved on 05.03.2007 from: http://www.parashift.com/c++-faq-lite/index.html.

Cline M., G. Lomow, M. Girou 2000: *C++ FAQs* (Second Edition), Addison Wesley, 2000, retrieved on 19.08.2014 from: http://mmc.geofisica.unam.mx/femp/Herramientas/CyC++/Documentacion/C++FAQs.pdf.

Deitel, H.M. & P.J. Deitel 2013: *C++ How to Program* 9th ed., Prentice-Hall, Upper Saddle River, NJ, USA.

Dijkstra, E.W. 1988: "Go to Statement Considered Harmful", *CACM* v.**11**, pp. 147-148.

Diller, A. 1990: *Z: An Introduction to Formal Methods*, J. Wiley & Sons, Chichester, UK.

ELLEMTEL, 1992: *Programming in C++: Rules and Recommendations*, Document: M 90 0118 Uen, Rev. C, 27 April 1992, retrieved on 01.06.2009 from: http://www.doc.ic.ac.uk/lab/cplus/c++.rules/

Feuer, A.R. 1982: *The C Puzzle Book*, Prentice Hall, Englewood Cliffs, NJ, US.

Haendel, L. 2001: *www.Function-Pointer.org: The site dedicated to C and C++ Function Pointers*, retrieved on Oct 2009 from: http://www.newty.de/fpt/index.html

Hanly, J.R. 1997: *Essential C++ for Engineers and Scientists*, Addison-Wesley, Reading Ma, US.

Henricson, M. & E. Nyquist 1997: *Industrial Strength C++ - Rules and Recommendations*, Prentice Hall, Upper Saddle River, NJ, US.

Hosey, P. 2007: *Everything you need to know about pointers in C*, retrieved on 10.10.2008 from: http://boredzo.org/pointers/

ISO/IEC 1999 (C99): *Programming Languages — C, Second Edition*, ISO/IEC 9899-1999.

ISO 2003: *International Standard ISO-IEC 14882: Programming Language C++*, Second Edition, 2003-10-15.

ISO 2011: *International Standard ISO-IEC 14882: Information technology — Programming languages — C++*, Third Edition, 2011-09-01.

ISO/IEC JTC1 SC22 WG21 N3337 2012 (C++0X): *Programming Languages — C++* (Working Draft Standard), ISO 2012, retrieved on 16.05.2012 from: http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

ISO/IEC JTC1 SC22 WG21 N3797 2014 (C++14): *Programming Languages — C++* (Working Draft Standard), ISO 2013, retrieved on 29.05.2014 from: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf

Jamsa, K. 1994: *Εισαγωγή στη C++*, Κλειδάριθμος, Αθήνα.

Jones, C.B. 1990: *Systematic Software Development using VDM*, Prentice Hall, London, UK.

Kalev, D. 2011: *The Biggest Changes in C++11 (and Why You Should Care)*, retrieved on 19.08.2014 from: http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/.

Kernighan, B.W. & R. Pike 1996: *The Practice of Programming*, Addison-Wesley, Reading Ma, US.

Kernighan, B.W. & D.M. Ritchie 1988: *The C Programming Language*, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, US.

Knuth, D. 1977: "Structured Programming with **goto** Statements", στο *Current Trends in Programming Methodology*, R.T. Yeh (ed.) vol. 1, Prentice Hall, Englewood Cliffs, NJ, US, αναδημοσίευση από το *Comp. Surveys*, **6**, 4, pp. 261-301 (1974).

Lafore, R. 2001: *Αντικειμενοστρεφής προγραμματισμός με τη C++*, Κλειδάριθμος, Αθήνα.

Liberty, J. 1995: *Εγχειρίδιο C++*, Γκιούρδας, Αθήνα.

Lippman, S.B., J.Lajoie, B.E.Moo 2013: *C++ Primer*, 5th ed., Addison-Wesley, Reading Ma, US. Retrieved on 19.08.2014 from: http://dl.e-book-free.com/2013/07/ c_primer_5th_edition.pdf.

Liskov, B. & J. Wing 1993: *Family Values: A semantic notion of subtyping*, MIT Lab for Computer Science TR-562b, retrieved on 10.10.2001 from: http://citeseer.ist.psu.edu/liskov94family.html.

Lockheed Martin, 2005: *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, doc. num.: 2RDU00001 Rev C., Dec 2005, retrieved on 01.06.2009 from: http://www.research.att.com /%7Ebs/JSF-AV-rules.pdf.

Manna, Z. & R. Waldinger 1985: *The Logical Basis for Computer Programming, Vol.1: Deductive Reasoning*, Addison-Wesley, Reading Ma, US.

Mansfield, K.C.Jr & J.L.Antonakos 1997: *An Introduction to Programming using C++*, Prentice Hall, Englewood Cliffs, NJ.

OMG 2010: *OMG Unified Modeling Language™ (OMG UML), Superstructure,* Ver. 2.3, May 2010, retrieved on 19.12.2010 from: http://www.omg.org/spec/UML/2.3/Superstructure.

Satir, G. & D. Brown 1995: *C++ : The Core Language*, O'Reilly & Associates, New York, NY, US.

Schildt, H. 1997: *Teach Yourself C++*, Osborne McGraw-Hill, New York, NY, US.

Schildt, H. 1998: *C++ from the Ground Up*, Osborne McGraw-Hill, New York, NY, US.

SGI 1999: *Standard Template Library Programmer's Guide*, retrieved on 02.06.2003 from http://www.sgi.com/tech/stl/

Spivey, J.M. 1988 : *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge Univ. Press, Cambridge, UK.

Stepanov, A. A., 2007: *Notes on Programming*, retrieved on 12.08.2008 from http://www.stepanovpapers.com/

Stroustrup, B. 1997: *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading Ma, US.

Stroustrup, B. 2011: *C++11 - the new ISO C++ standard*, retrieved on 19.08.2014 from http://www.stroustrup.com/C++11FAQ.html

Sutter, H. 1998: *GotW #49: Template Specialization and Overloading*, Dec 2008, retrieved on March 2007 from: http://www.gotw.ca/gotw/049.htm.

Sutter, H. 1999: *Exceptional C++: 47 Engineering Puzzles*, Addison Wesley, 1999, retrieved on 20.08.2014 from http://aalmos.kaniserver.net/doc/cpp/Exceptional%20C%2B%2B.pdf

Sutter, H. 2002: *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*, Addison Wesley, 2002

Wegner, P. 1990: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOPS Messenger*, v.1, i.1, 7-87.

Yaroshenko, O. 1995: *The Beginner's Guide to C++*, Wrox Press Inc., NewYork, NY, US.

Παραρτήματα

παράρτημα

A

Στοιχεία Προτασιακού Λογισμού

Ο στόχος μας σε αυτό το παράρτημα:

Τα παρακάτω έχουν στόχο να σου υπενθυμίσουν μερικές βασικές λογικές πράξεις που χρησιμοποιούμε.

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Θα πρέπει να τονίσουμε ότι

- τα περιεχόμενα του παραρτήματος δεν αποτελούν κάλυψη του Προτασιακού Λογισμού,
- ο Προτασιακός Λογισμός δεν είναι αρκετός για πλήρεις τυπικές αποδείξεις ορθότητας προγραμμάτων.

Στην §A.5 υπάρχει και μια παρουσίαση των ποσοδεικτών και της ισότητας: «γεύση» από τον Κατηγορηματικό Λογισμό.

Αν θέλεις να διαβάσεις περισσότερα για τη Μαθηματική Λογική δες τη βιβλιογραφία στο τέλος του βιβλίου.

A.1 Συντακτικά

Το αλφάβητο που θα χρησιμοποιήσουμε είναι:

$$\{ P, Q, R, P_0, Q_0, R_0, P_1, Q_1, R_1, \ldots, \mathsf{F}, \&\&, ||, \Rightarrow, \Leftrightarrow, !, (,) \}$$

Τα P, Q, R με δείκτη ή χωρίς είναι τα **προτασιακά σύμβολα** (propositional symbols). Με αυτά θα παριστάνουμε τις ατομικές προτάσεις.

Η "`F`" είναι μια προτασιακή **σταθερά** (constant)· αλλού θα τη δεις και ως "⊥".

Τα "`&&`", "`||`", "⇒", "⇔", "`!`" είναι τα σύμβολα των **συνδέσμων** (connectives)· οι τέσσερις πρώτοι ονομάζονται **διμελείς** (binary, 2-place) σύνδεσμοι ενώ ο "`!`" είναι **μονομελής** (unary, 1-place) σύνδεσμος. Μπορούμε να θεωρήσουμε τη σταθερά "`F`" και ως **σύνδεσμο χωρίς μέλη** (0-place connective).

Οι προτάσεις της γλώσσας ονομάζονται προτάσεις και σχηματίζονται σύμφωνα με τον εξής κανόνα:

πρόταση = *προτασιακό σύμβολο* | "`F`" | "`!`", *πρόταση* |
"`(`", *πρόταση*, *διμελής σύνδεσμος*, *πρόταση*, "`)`" ;
προτασιακό σύμβολο = "*P*" | "*Q*" | "*R*" |
"P_0" | "Q_0" | "R_0" | "P_1" | "Q_1" | "R_1" | ... ;
διμελής σύνδεσμος = "`&&`" | "`||`" | "⇒" | "⇔" ;

Παράδειγμα

Οι

$$P,\quad Q_2,\quad R_4$$

είναι προτάσεις αφού είναι προτασιακά σύμβολα.

Αφού οι P, Q_2, R_4 είναι προτάσεις και οι

$$!P,\quad !Q_2,\quad !R_4$$

είναι προτάσεις αφού έχουν τη μορφή "`!`", *πρόταση*.

Αφού οι P, Q_2, R_4, $!P$, $!Q_2$, $!R_4$ είναι προτάσεις και οι

$$(!P \;\&\&\; R_4),\quad (Q_2 \;||\; !Q_2),\quad (P \Rightarrow !R_4),\quad (Q_2 \Leftrightarrow R_4)$$

είναι προτάσεις αφού έχουν τη μορφή "`(`", *πρόταση*, *διμελής σύνδεσμος*, *πρόταση*, "`)`".

Αφού οι $(!P \;\&\&\; R_4)$, $(Q_2 \;||\; !Q_2)$, $(P \Rightarrow !R_4)$, $(Q_2 \Leftrightarrow R_4)$ είναι προτάσεις και οι

$$!(!P \;\&\&\; R_4),\quad !(Q_2 \;||\; !Q_2),\quad !(P \Rightarrow !R_4),\quad !(Q_2 \Leftrightarrow R_4)$$

είναι προτάσεις αφού έχουν τη μορφή "`!`", *πρόταση*.

Αφού οι $(!P \;\&\&\; R_4)$, $!R_4$, P, $(Q_2 \Leftrightarrow R_4)$, $!Q_2$, $(P \Rightarrow !R_4)$ είναι προτάσεις και οι

$$((!P \;\&\&\; R_4) \;||\; !R_4),$$
$$(P \;\&\&\; (Q_2 \Leftrightarrow R_4)),$$
$$((P \;\&\&\; (Q_2 \Leftrightarrow R_4)) \;||\; !Q_2),$$
$$((P \Rightarrow !R_4) \;||\; (Q_2 \Leftrightarrow R_4))$$

είναι προτάσεις αφού έχουν τη μορφή "`(`", *πρόταση*, *διμελής σύνδεσμος*, *πρόταση*, "`)`".

Όπως βλέπεις και στο παράδειγμα, οι παρενθέσεις μπορεί να γίνουν ενοχλητικά πολλές. Δίνουμε λοιπόν τον παρακάτω συμπληρωματικό κανόνα:

- ***Από μια πρόταση μπορούμε να αφαιρούμε το εξώτερο ζεύγος παρενθέσεων***

Με βάση αυτόν τον κανόνα, μπορούμε να γράψουμε:

$$!P \;\&\&\; R_4 \quad \text{αντί για} \quad (!P \;\&\&\; R_4),$$
$$Q_2 \;||\; !Q_2 \quad \text{αντί για} \quad (Q_2 \;||\; !Q_2),$$
$$P \Rightarrow !R_4 \quad \text{αντί για} \quad (P \Rightarrow !R_4),$$
$$Q_2 \Leftrightarrow R_4 \quad \text{αντί για} \quad (Q_2 \Leftrightarrow R_4),$$
$$(!P \;\&\&\; R_4) \;||\; !R_4 \quad \text{αντί για} \quad ((!P \;\&\&\; R_4) \;||\; !R_4),$$
$$P \;\&\&\; (Q_2 \Leftrightarrow R_4) \quad \text{αντί για} \quad (P \;\&\&\; (Q_2 \Leftrightarrow R_4)),$$
$$(P \;\&\&\; (Q_2 \Leftrightarrow R_4)) \;||\; !Q_2 \quad \text{αντί για} \quad ((P \;\&\&\; (Q_2 \Leftrightarrow R_4)) \;||\; !Q_2),$$
$$(P \Rightarrow !R_4) \;||\; (Q_2 \Leftrightarrow R_4) \quad \text{αντί για} \quad ((P \Rightarrow !R_4) \;||\; (Q_2 \Leftrightarrow R_4))$$

Α.2 Νοηματικά

Κάθε προτασιακό σύμβολο (P, P_0, P_1,... Q, Q_0, Q_1,... R, R_0, R_1,...) μπορεί να πάρει μια από τις λογικές τιμές `false` (ψέμα, δεν ισχύει) ή `true` (αλήθεια, ισχύει). Η σταθερά `F` παίρνει πάντοτε τιμή `false`.

Από τη στιγμή που καθορίζονται οι τιμές των προτασιακών συμβόλων μπορούμε να υπολογίσουμε τη (λογική) τιμή οποιασδήποτε πρότασης που περιέχει αυτά τα σύμβολα. Για να μπορέσουμε να κάνουμε κάτι τέτοιο, θα πρέπει να ξέρουμε τα αποτελέσματα των λογικών πράξεων που συμβολίζονται με τα "`&&`", "`||`", "`!`". Αυτά συνήθως δίνονται σε πίνακες, που λέγονται **πίνακες αλήθειας** ή **αληθοπίνακες** (truth tables).

A.2.1 Λογική Σύζευξη "&&"

Στον Πίν. Α-1 βλέπεις τον αληθοπίνακα της λογικής **σύζευξης** (conjunction) "`&&`". Η πρώτη γραμμή λέει: Αν η πρόταση $\mathcal{A}$ έχει τιμή `false` και η πρόταση $\mathcal{B}$ έχει τιμή `false` τότε και η $\mathcal{A}$ `&&` $\mathcal{B}$ και $\mathcal{B}$ `&&` $\mathcal{A}$ έχουν τιμή `false`. Με τον ίδιο τρόπο διαβάζεις και τις υπόλοιπες γραμμές.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A}$ `&&` $\mathcal{B}$	$\mathcal{B}$ `&&` $\mathcal{A}$
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`false`
`true`	`false`	`false`	`false`
`true`	`true`	`true`	`true`

Πίν. Α-1: Ο αληθοπίνακας της "`&&`".

Πρόσεξε ότι η πράξη "`&&`" είναι αντιμεταθετική: Οι $\mathcal{A}$ `&&` $\mathcal{B}$ και $\mathcal{B}$ `&&` $\mathcal{A}$ έχουν πάντοτε την ίδια τιμή.

Πώς μπορούμε να διατυπώσουμε στην καθομιλουμένη το αποτέλεσμα της πράξης "`&&`";

- ***Αν η πρόταση $\mathcal{A}$ έχει τιμή `true` (είναι αληθής) και ταυτοχρόνως η πρόταση $\mathcal{B}$ έχει τιμή `true` τότε (και μόνον τότε) είναι αληθείς και οι προτάσεις $\mathcal{A}$ `&&` $\mathcal{B}$ και $\mathcal{B}$ `&&` $\mathcal{A}$.***

Παράδειγμα

Έστω ότι P είναι η πρόταση «η μεταβλητή x έχει τιμή μεγαλύτερη από 5» ($x > 5$) και Q η πρόταση «η μεταβλητή x έχει τιμή μικρότερη από 10» ($x < 10$). Η πρόταση «η μεταβλητή x έχει τιμή μεταξύ 5 και 10» ($5 < x < 10$) σημαίνει στην πραγματικότητα

«η μεταβλητή x έχει τιμή μεγαλύτερη από 5»

και

«η μεταβλητή x έχει τιμή μικρότερη από 10»

δηλαδή P `&&` Q. Αν η x έχει τιμή 6, τότε η P και η Q είναι αληθείς, έχουν τιμή `true`. Από την τέταρτη γραμμή του αληθοπίνακα της "`&&`" βλέπουμε ότι και η P `&&` Q έχει τιμή `true`, δηλαδή είναι αληθής.

Αν το x έχει τιμή 17, τότε η P είναι αληθής (τιμή `true`) και η Q είναι ψευδής (τιμή `false`). Από την τρίτη γραμμή του αληθοπίνακα της `&&` βλέπουμε ότι και η P `&&` Q έχει τιμή `false`, δηλαδή είναι ψευδής.

Συχνά θα δεις αυτόν τον σύνδεσμο να γράφεται και ως: $\wedge$, **και**, `and` (στις γλώσσες προγραμματισμού Pascal, Basic και άλλες).

Παρόμοιος είναι και ένας άλλος σύνδεσμος, ο "`cand`"· αυτός έχει «οικονομικό» τρόπο υπολογισμού, δηλαδή: αν κατά τον υπολογισμό της $\mathcal{A}$ `cand` $\mathcal{B}$ η τιμή της $\mathcal{A}$ βρεθεί `false` (ψευδής), η τιμή ολόκληρης της σύνθετης πρότασης υπολογίζεται ως `false` χωρίς να υπολογισθεί η τιμή της $\mathcal{B}$. Να θυμίσουμε ότι έτσι γίνεται ο υπολογισμός της πράξης "`&&`" στη C++ (§5.6).

A.2.2 Λογική Διάζευξη "||"

Στον Πίν. Α-2 βλέπεις τον αληθοπίνακα της λογικής **διάζευξης** (disjunction) "`||`". Στην καθομιλουμένη το αποτέλεσμα της πράξης "`||`" δίνεται με τον εξής κανόνα:

- ***Αν μια τουλάχιστον από τις προτάσεις $\mathcal{A}$, $\mathcal{B}$ έχει τιμή `true` (είναι αληθής) τότε και οι προτάσεις $\mathcal{A}$ `||` $\mathcal{B}$ και $\mathcal{B}$ `||` $\mathcal{A}$ έχουν τιμή `true` (είναι αληθείς).***

Όπως βλέπεις, η μόνη περίπτωση που η $\mathcal{A}$ `||` $\mathcal{B}$ παίρνει τιμή `false` είναι όταν και η $\mathcal{A}$ και η $\mathcal{B}$ έχουν τιμή `false`.

Α	*Β*	*Α* `||` *Β*	*Β* `||` *Α*
`false`	`false`	`false`	`false`
`false`	`true`	`true`	`true`
`true`	`false`	`true`	`true`
`true`	`true`	`true`	`true`

Πίν. Α-2: Ο αληθοπίνακας της "`||`".

Και η "`||`" είναι αντιμεταθετική: Οι *Α* `||` *Β* και *Β* `||` *Α* έχουν πάντοτε την ίδια τιμή.

Παράδειγμα

Έστω ότι *x* = 15 και θέλουμε να αποφανθούμε για την αλήθεια της πρότασης «το *x* είναι πολλαπλάσιο του 2 ή του 3». Ένας άλλος τρόπος να διατυπώσουμε την πρόταση είναι: «το *x* είναι πολλαπλάσιο του 2» ή «το *x* είναι πολλαπλάσιο του 3». Ας ονομάσουμε *P* την «το *x* είναι πολλαπλάσιο του 2» και *Q* την «το *x* είναι πολλαπλάσιο του 3». Η *P* έχει τιμή `false`, ενώ η *Q* έχει τιμή `true`. Σύμφωνα με την τρίτη γραμμή του πίνακά μας η *P* `||` *Q* έχει τιμή `true`.

Άλλες γραφές του συνδέσμου: "∨", "**ή**", "`or`" (στις γλώσσες προγραμματισμού Pascal, Basic και άλλες).

Παρόμοιος είναι και ένας άλλος σύνδεσμος (αντίστοιχος του "`cand`"), ο "`cor`": αν κατά τον υπολογισμό της *Α* `cor` *Β* η τιμή της *Α* βρεθεί `true` (αληθής), η τιμή ολόκληρης της σύνθετης πρότασης υπολογίζεται ως `true` χωρίς να υπολογισθεί η τιμή της *Β*. Αν γυρίσεις στην §5.6 θα θυμηθείς ότι έτσι δουλεύει ο "`||`" της C++.

Α.2.3 Η Λογική Άρνηση "!"

Α	`!`*Α*
`false`	`true`
`true`	`false`

Πίν. Α-3: Ο αληθοπίνακας της "`!`".

Στον Πίν. Α-3 βλέπεις τον αληθοπίνακα της λογικής **άρνησης** (negation) "`!`", που μας λέει απλά:

- ***Αν η Α έχει τιμή*** `true`***, είναι αληθής, τότε η*** `!`***Α έχει τιμή*** `false`***, είναι ψευδής, και αν η Α έχει τιμή*** `false`***, είναι ψευδής, τότε η*** `!`***Α έχει τιμή*** `true`***, είναι αληθής.***

Θα τον δεις γραμμένο ακόμη ως "¬" ή ως "~" ή ως "**όχι**" ή ως "`not`" (Pascal, Basic).

Α.2.4 Συνεπαγωγή "⇒"

Α	*Β*	*Α* ⇒ *Β*	*Β* ⇒ *Α*
`false`	`false`	`true`	`true`
`false`	`true`	`true`	`false`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Πίν. Α-4: Ο αληθοπίνακας της "⇒". Οι στήλες *Α* ⇒ *Β* και *Β* ⇒ *Α* είναι διαφορετικές.

Στον Πίν. Α-3 βλέπεις τον αληθοπίνακα της "⇒", που τη λέμε **συνεπαγωγή** (implication).

Στην *Α* ⇒ *Β* η *Α* λέγεται **ηγουμένη** (antecedent) και η *Β* **επομένη** (consequent) ή **απόδοση** (apodosis). Αλλού θα δεις την *Α* ⇒ *Β* να γράφεται ως "***if Α then Β***", δηλαδή ***αν*** (είναι αληθής η) *Α* ***τότε*** (είναι αληθής η) *Β*.[1]

Θα γράψουμε τον αληθοπίνακα της συνεπαγωγής προσπαθώντας να καταλάβουμε ένα παράδειγμα με βάση την καθημερινή κοινή λογική. Πάρε ως *P* την πρόταση «ο Νίκος λέει πάντα ψέματα» και ως *Q* την «αυτό που είπε τώρα ο Νίκος είναι ψέμα». *P* ⇒ *Q* είναι η πρόταση: «***αν*** ο Νίκος λέει πάντα ψέματα ***τότε*** αυτό που είπε τώρα ο Νίκος είναι ψέμα». Θα δοκιμάσουμε όλες τις δυνατές τιμές αλήθειας για τις *P*, *Q* και στην κάθε περίπτωση θα βλέπουμε αν αυτό που προκύπτει συμφωνεί με την κοινή λογική.

Ξεκινούμε από τα εύκολα: η *P* έχει τιμή `true`, το ίδιο και η *Q* (γραμμή 4 του Πίν. Α-3)· έχουμε δηλαδή την πρόταση «αν ο Νίκος λέει πάντα ψέματα τότε αυτό που είπε τώρα ο Νίκος είναι ψέμα». Αυτή είναι αληθής, σύμφωνα με την κοινή λογική. Βάζουμε λοιπόν τιμή `true` στην *P* ⇒ *Q*.

[1] Όπως καταλαβαίνεις, αυτό το ***if ... then*** δεν έχει σχέση με την αντίστοιχη προγραμματιστική δομή.

Πάμε τώρα στην περίπτωση που η P έχει τιμή `true`, αλλά η Q έχει τιμή `false` (γραμμή 3)· τώρα η $P \Rightarrow Q$ σημαίνει: «αν ο Νίκος λέει πάντα ψέματα τότε αυτό που είπε τώρα ο Νίκος *δεν* είναι ψέμα». Αυτή η πρόταση όμως είναι ολοφάνερα ψευδής με βάση την κοινή λογική. Βάζουμε λοιπόν τιμή `false` στην $P \Rightarrow Q$.

Στη γραμμή 2 η ηγουμένη P έχει τιμή `false` και η Q τιμή `true`, λέμε δηλαδή «αν ο Νίκος *δεν* λέει πάντα ψέματα τότε αυτό που είπε τώρα ο Νίκος είναι ψέμα». Αυτή μπορεί να φαίνεται λίγο "ασυνάρτητη" ή "παράλογη", αλλά το ερώτημα είναι: είναι αληθής; Πώς να απαντήσουμε μια τέτοια ερώτηση για μια "ασυναρτησία"; Ας κάνουμε την άλλη ερώτηση: είναι ψευδής; Όχι! Αφού λοιπόν η πρόταση δεν είναι ψευδής (και η λογική μας είναι δίτιμη) δεν μπορεί παρά να είναι αληθής, δηλαδή η $P \Rightarrow Q$ έχει τιμή `true`.

Τέλος, στη γραμμή 1, ηγουμένη και επομένη έχουν τιμή `false`. Έχουμε δηλαδή την "ασυναρτησία" «αν ο Νίκος *δεν* λέει πάντα ψέματα τότε αυτό που είπε τώρα ο Νίκος *δεν* είναι ψέμα». Και εδώ, με το σκεπτικό της προηγούμενης περίπτωσης, βάζουμε στην $P \Rightarrow Q$ τιμή `true`.

Λίγο παράξενα τα δύο τελευταία... Υπάρχουν όμως και χειρότερα. Π.χ. οι παρακάτω προτάσεις είναι αληθείς:

«αν ο ουρανός είναι γαλανός τότε οι λευκές αρκούδες ζουν πολύ βόρεια»

«αν ο Ήλιος γυρίζει γύρω από τη Γη τότε εγώ είμαι αστροναύτης»

Η πρώτη πρόταση είναι αληθής διότι οι δύο ατομικές προτάσεις που την αποτελούν είναι αληθείς, η δεύτερη είναι επίσης αληθής διότι η πρώτη ατομική πρόταση είναι ψευδής. Αλλά, και στις δύο περιπτώσεις, ηγούμενη και επόμενη είναι τελείως άσχετες. Εδώ το πρόβλημα είναι σοβαρό[2]...

Όπως είναι φυσικό, αυτή η λογική πράξη *δεν είναι αντιμεταθετική*. Άλλο το $\mathcal{A} \Rightarrow \mathcal{B}$ και άλλο το $\mathcal{B} \Rightarrow \mathcal{A}$.

Αλλού θα δεις την $\mathcal{A} \Rightarrow \mathcal{B}$ να γράφεται ως $\mathcal{A} \rightarrow \mathcal{B}$ ή $\mathcal{A} \supset \mathcal{B}$ ή, όπως είπαμε, ***if*** $\mathcal{A}$ ***then*** $\mathcal{B}$.

Παρατήρηση:▶

Αν δεις τον Πίν. Α-4 και πάρεις υπόψη σου τη διάταξη "`false < true`" που έχουμε στη C++ καταλαβαίνεις γιατί η "$P \Rightarrow Q$" υλοποιείται στη C++ ως "P `<=` Q".◀

Α.2.5 Διπλή Συνεπαγωγή "⇔"

Στον Πίν. Α-4 βλέπεις τον αληθοπίνακα της "⇔", που τη λέμε **διπλή συνεπαγωγή** (double implication). Τι μας λέει;

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \Leftrightarrow \mathcal{B}$	$\mathcal{B} \Leftrightarrow \mathcal{A}$
`false`	`false`	`true`	`true`
`false`	`true`	`false`	`false`
`true`	`false`	`false`	`false`
`true`	`true`	`true`	`true`

Πίν. Α-5: Ο αληθοπίνακας της "⇔".

- ***Αν οι $\mathcal{A}$ και $\mathcal{B}$ έχουν την ίδια τιμή (και οι δύο*** `true` ***ή και οι δύο*** `false`***) τότε η $\mathcal{A} \Leftrightarrow \mathcal{B}$ είναι αληθής (τιμή*** `true`***), αλλιώς η $\mathcal{A} \Leftrightarrow \mathcal{B}$ είναι ψευδής (τιμή*** `false`***).***

Όπως βλέπεις, οι $\mathcal{A} \Leftrightarrow \mathcal{B}$ και $\mathcal{B} \Leftrightarrow \mathcal{A}$ έχουν πάντοτε την ίδια τιμή· η πράξη ⇔ είναι αντιμεταθετική.

Παράδειγμα

Έστω ότι $x = 17$. Αν P είναι η πρόταση "$x > 10$" και Q η "$x < 35$", τότε η $P \Leftrightarrow Q$ είναι αληθής διότι και η P και η Q έχουν τιμή `true`. Αν P_1 είναι η πρόταση "$x < 10$" και Q_1 η "$x > 35$" τότε η $P_1 \Leftrightarrow Q_1$ είναι αληθής διότι και η P_1 και η Q_1 έχουν τιμή `false`. Η $P \Leftrightarrow Q_1$ είναι ψευδής διότι η P έχει τιμή `true`, ενώ η έχει Q_1 `false`.

---

[2] Αυτή η συνεπαγωγή ονομάζεται και **ουσιώδης συνεπαγωγή** (material implication) σε αντιδιαστολή με μια άλλη πιο **ακριβή συνεπαγωγή** (strict implication) που εξετάζουν οι **τροπικές λογικές** (modal logics) βάζοντας τη συνεπαγωγή σε πιο "σωστή" βάση.

Αλλού θα δεις την $\mathcal{A} \Leftrightarrow \mathcal{B}$ να γράφεται ως $\mathcal{A} \leftrightarrow \mathcal{B}$ ή ***A if and only if B***. Θα τη δεις ακόμη ως $\mathcal{A} \equiv \mathcal{B}$ αλλά εδώ προσοχή: όπως θα δεις στη συνέχεια, εμείς θα χρησιμοποιούμε το "≡" με άλλο νόημα.

Παρατήρηση:▶

Στη C++ η "$P \Leftrightarrow Q$" υλοποιείται ως "`P == Q`".◀

Α.2.6 Άλλοι Σύνδεσμοι

Όπως καταλαβαίνεις, μπορούμε να έχουμε 2^4 διμελείς συνδέσμους και 2^2 μονομελείς. Θα πρέπει να τους περιλάβουμε όλους στο σύστημά μας; Όχι! Αυτοί που έχουμε εδώ φτάνουν και περισσεύουν, όπως μπορεί να αποδειχθεί. Πάντως υπάρχουν τρεις που είναι χρήσιμοι και καλό είναι να τους δούμε.

Θα δούμε στη συνέχεια ότι οι δύο πρώτοι από αυτούς, οι "`nand`" και "`nor`" έχουν μια ενδιαφέρουσα ιδιότητα: όλοι οι άλλοι σύνδεσμοι μπορεί να δοθούν με βάση έναν από αυτούς.

Α.2.6.1 Ο Σύνδεσμος του Sheffer "|"

Στον Πίν. Α-6 βλέπεις τον αληθοπίνακα του "`|`" που λέγεται **σύνδεσμος του Sheffer** (Sheffer's stroke). Αν τον συγκρίνεις με τον Πίν. Α-1 βλέπεις ότι το $\mathcal{A}$ `|` $\mathcal{B}$ είναι μια συντομογραφία του `!(`$\mathcal{A}$ `&&` $\mathcal{B}$`)`.

Θα τον δεις γραμμένο και ως "↑"ή ως "`nand`" (αφού είναι η "άρνηση της and": **n**ot **and**).

Α.2.6.2 Η Άρνηση της "||": "↓"

Όπως υπάρχει "άρνηση της and" υπάρχει και "άρνηση της or". Συμβολίζεται με το "↓" και στον Πίν. Α-7 βλέπεις τον αληθοπίνακά του. Αν τον συγκρίνεις με τον Πίν. Α-2 βλέπεις ότι, πράγματι, το $\mathcal{A} \downarrow \mathcal{B}$ είναι μια συντομογραφία του `!(`$\mathcal{A}$ `&&` $\mathcal{B}$`)`.

Θα τον δεις γραμμένο και ως "`nor`" (**n**ot **or**).

Α.2.6.3 Αποκλειστική Διάζευξη "⊻"

Η **αποκλειστική διάζευξη** (exclusive disjunction) –αληθοπίνακας στον Πίν. Α-8– διαφέρει από την απλή διάζευξη (Πίν. Α-2) στην τελευταία γραμμή του αληθοπίνακα. Η $\mathcal{A} \veebar \mathcal{B}$ έχει τιμή `true` όταν *μία και μόνο μία* από τις $\mathcal{A}$ και $\mathcal{B}$ έχει τιμή `true`. Συγκρίνοντας με τον Πίν. Α-4 βλέπεις ότι είναι το ίδιο πράγμα με το `!(`$\mathcal{A} \Leftrightarrow \mathcal{B}$`)`.

Θα τον δεις γραμμένο και ως "⊕" ή ως "`xor`" (Turbo Pascal).

Α.2.7 Τιμή Παράστασης

Στους αληθοπίνακες ορισμού των λογικών πράξεων χρησιμοποιήσαμε προτασιακά σύμβολα (P, Q). Οι πράξεις γίνονται με τον ίδιο τρόπο μεταξύ οποιωνδήποτε προτάσεων. Ας δούμε ένα

Παράδειγμα 🡲

Με βάση τους πίνακες, θα υπολογίσουμε την τιμή της παράστασης:

$$((P_0 \;\&\&\; P_1) \;||\; (P_2 \;\&\&\; P_1)) \;\&\&\; P_3$$

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A}$ | $\mathcal{B}$
false	false	true
false	true	true
true	false	true
true	true	false

Πίν. Α-6: Ο αληθοπίνακας της "`|`".

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \downarrow \mathcal{B}$
false	false	True
false	true	False
true	false	False
true	true	False

Πίν. Α-7: Ο αληθοπίνακας της "↓"

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \veebar \mathcal{B}$
False	false	false
False	true	true
True	false	true
True	true	false

Πίν. Α-8: Ο αληθοπίνακας της "⊻".

αν έχουμε δώσει τιμές:

`true` στην P_0, `false` στην P_1, `true` στην P_2, `false` στην P_3

Η P_0 `&&` P_1 παίρνει τιμή `false` (`&&`, γρ. 3),
η P_2 `&&` P_1 παίρνει τιμή `false` (`&&`, γρ. 3),
η (P_0 `&&` P_1) `||` (P_2 `&&` P_1) παίρνει τιμή `false` (`||`, γρ. 1) και
η ((P_0 `&&` P_1) `||` (P_2 `&&` P_1)) `&&` P_3 παίρνει τιμή `false` (`&&`, γρ. 1)[3].
🖘🖘🖘

Α.3 Ταυτολογίες και Ισοδυναμία

Ένα πολύ ενδιαφέρον υποσύνολο προτάσεων είναι οι ταυτολογίες. **Ταυτολογία** (tautology) είναι μια πρόταση που παίρνει τιμή `true` όποιες κι αν είναι οι τιμές των προτασιακών συμβόλων.

Χαρακτηριστικά παραδείγματα ταυτολογιών:

`!F`
A `||` (`!`A)
(A `&&` (B `||` C)) ⇔ ((A `&&` B) `||` (A `&&` C))
(A `||` (B `&&` C)) ⇔ ((A `||` B) `&&` (A `||` C))
(`!`(A `||` B)) ⇔ ((`!`A) `&&` (`!`B))
(`!`(A `&&` B)) ⇔ ((`!`A) `||` (`!`B))

Αν η $A \Leftrightarrow B$ είναι ταυτολογία τότε οι A και B λέγονται **ισοδύναμες** (equivalent). Στην περίπτωση αυτήν γράφουμε $A \equiv B$. Από τα τελευταία παραδείγματα που δώσαμε πιο πάνω έχουμε:

(A `&&` (B `||` C)) ≡ ((A `&&` B) `||` (A `&&` C))
(A `||` (B `&&` C)) ≡ ((A `||` B) `&&` (A `||` C))
(`!`(A `||` B)) ≡ ((`!`A) `&&` (`!`B))
(`!`(A `&&` B)) ≡ ((`!`A) `||` (`!`B))

Αν δεν σε βολεύει να δουλεύεις με την ⇒, μπορείς να χρησιμοποιείς την ισοδυναμία (Άσκ. 1α):

($A \Rightarrow B$) ≡ ((`!`A) `||` B)

Συχνά χρησιμοποιούμε τις ταυτολογίες (Άσκ. 1β):

$A \Rightarrow$ (A `||` B) και $B \Rightarrow$ (A `||` B)

όπως και τις:

(A `&&` B) ⇒ A και (A `&&` B) ⇒ B

Είναι φανερό ότι ισχύουν οι παρακάτω ισοδυναμίες (Άσκ. 2):

($A \veebar B$) ≡ ((A `||` B) `&&` (`!`(A `&&` B)))
($A \veebar B$) ≡ ((A `&&` (`!`B)) `||` ((`!`A) `&&` B))

Α.4 Αποδείξεις

Ας προσπαθήσουμε να αποδείξουμε ότι η A `||` (`!`A) είναι ταυτολογία. Ένας τρόπος είναι να εξετάσουμε όλες τις δυνατές τιμές της A. Αυτό το βλέπεις στον Πίν. Α-9. Όπως βλέπεις, η τελευταία στήλη του πίνακα έχει σ' όλες τις γραμμές την τιμή `true`.

A	`!`A	A `||` (`!`A)
`false`	`true`	`true`
`true`	`false`	`true`

Πίν. Α-9: Αληθοπίνακας για την απόδειξη της ταυτολογίας A`||` (`!`A).

Να δούμε άλλο ένα

[3] Βέβαια, αν είμαστε λίγο προσεκτικοί, αποφαινόμαστε αμέσως ότι πρότασή μας παίρνει τιμή `false`, αφού είναι της μορφής A `&&` P_3 και η P_3 έχει τιμή `false` (`&&`, γρ. 1, 3).

Παράδειγμα

Θα αποδείξουμε ότι η $(P \;\&\&\; Q) \Rightarrow P$ είναι ταυτολογία. Θα χρειαστούμε έναν αληθοπίνακα με στήλες για τις P, Q, $P \;\&\&\; Q$, $(P \;\&\&\; Q) \Rightarrow P$. Θα πρέπει να εξετάσουμε όλους τους συνδυασμούς τιμών των P, Q· άρα ο πίνακας θα έχει 4 (= 2×2) γραμμές. Τον βλέπεις στον Πίν. Α-10.

P	Q	$P \;\&\&\; Q$	$(P \;\&\&\; Q) \Rightarrow P$
`false`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Πίν. Α-10: Πίνακας αλήθειας για την απόδειξη της $(P \;\&\&\; Q) \Rightarrow P$.

Φυσικά, δεν αποδεικνύουμε μόνον ταυτολογίες. Μπορεί να χρειαστεί να αποδείξουμε και κάτι σαν:

Αν ισχύει η A τότε ισχύει και η B.

Παράδειγμα

Απόδειξε ότι: Αν ισχύει η $A \;\&\&\; B$ τότε ισχύει η $A \;||\; B$.

Και στην περίπτωση αυτήν μπορούμε να κάνουμε τον αληθοπίνακα, αλλά τώρα συγκρίνουμε μόνον τις γραμμές στις οποίες η υπόθεση $(A \;\&\&\; B)$ έχει τιμή `true`.[4] Αυτό συμβαίνει μόνο στην 4η γραμμή. Από αυτήν μπορούμε να πούμε ότι: αφού όποτε η υπόθεση παίρνει τιμή `true` και το συμπέρασμα παίρνει τιμή `true`, αποφαινόμαστε ότι: Αν ισχύει η $A \;\&\&\; B$ τότε ισχύει η $A \;||\; B$.

A	B	$A \;\&\&\; B$	$A \;||\; B$
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Πίν. Α-11: Πίνακας αλήθειας για την απόδειξη του θεωρήματος: αν ισχύει η $A \;\&\&\; B$ τότε ισχύει και η $A \;||\; B$.

Όπως βλέπεις, ο πρώτος πίνακας (Α-9) έχει 2 γραμμές, ενώ οι επόμενοι (Α-10, Α-11) έχουν 4 γραμμές. Γενικά, αν προσπαθήσεις να αποδείξεις μια πρόταση με N προτασιακά σύμβολα θα χρειαστείς πίνακα με 2^N γραμμές. Καταλαβαίνεις λοιπόν ότι αυτός ο τρόπος δεν είναι ικανοποιητικός.

Ένας άλλος τρόπος, προτιμότερος, είναι να βάλεις **αξιώματα** και **συμπερασματικούς κανόνες** και να αποδεικνύεις αυτό που θέλεις χωρίς να εξετάζεις όλες τις δυνατές τιμές των προτασιακών συμβόλων.

Ας δούμε, για παράδειγμα, μερικούς συμπερασματικούς κανόνες του προτασιακού λογισμού:

$$\frac{A, B}{A \;\&\&\; B}(1) \qquad \frac{A, B}{B \;\&\&\; A}(2) \qquad \frac{A \;\&\&\; B}{A}(3) \qquad \frac{A \;\&\&\; B}{B}(4)$$

Ο πρώτος κανόνας λέει: "Αν ισχύουν οι A και B τότε μπορείς να βγάλεις το συμπέρασμα ότι ισχύει και η $A \;\&\&\; B$". Ο τρίτος κανόνας λέει: "Αν ισχύει η $A \;\&\&\; B$ τότε μπορείς να βγάλεις το συμπέρασμα ότι ισχύει η A". Όπως καταλαβαίνεις, οι τέσσερις αυτοί κανόνες εκφράζουν αυτό που γράψαμε για την πράξη "`&&`": Αν ισχύει η πρόταση A και ταυτόχρονα ισχύει η πρόταση B τότε και μόνον τότε ισχύουν και οι προτάσεις $A \;\&\&\; B$ και $B \;\&\&\; A$.

Οι συμπερασματικοί κανόνες της "`||`" είναι οι εξής:

$$\frac{A}{A \;||\; B}(5) \qquad \frac{B}{A \;||\; B}(6)$$

Αν διαβάσεις αυτούς τους δύο κανόνες όπως είπαμε παραπάνω θα δεις ότι δεν λένε τίποτε περισσότερο από αυτό που είπαμε πιο πριν: Αν ισχύει η πρόταση A ή η πρόταση B ή και οι δύο τότε ισχύει και η πρόταση $A \;||\; B$.

[4] Φυσικά, το ότι «αν ισχύει η $A \;\&\&\; B$ τότε ισχύει η $A \;||\; B$» *έχει σχέση* με το ότι η $(A \;\&\&\; B) \Rightarrow (A \;||\; B)$ είναι ταυτολογία· δες αυτά που λέμε στο τέλος της παραγράφου.

Ας δούμε τώρα, με ένα παράδειγμα, πώς γίνεται η απόδειξη με αυτούς τους κανόνες, χωρίς να καταφύγουμε σε αληθοπίνακες. Ας πούμε ότι έχουμε να αποδείξουμε ότι: Αν ισχύει η A **&&** B τότε ισχύει η A **||** B.

1. A **&&** B υπόθεση.
2. A από το βήμα (1) και τον κανόνα (3).
3. A **||** B από το βήμα (2) και τον κανόνα (5).

Όπως βλέπεις, η απόδειξη στηρίζεται στη γραφή της πρότασης και όχι στις τιμές των A και B.

Αυτό που αποδείξαμε ήταν ένα *θεώρημα*. Το «Αν ισχύει η A **&&** B τότε ισχύει η A **||** B» γράφεται συμβολικώς ως εξής:

$$(A \;\&\&\; B) \vdash (A \;||\; B)$$

Αν η απόδειξη γίνει με αληθοπίνακες γράφουμε: $(A \;\&\&\; B) \models (A \;||\; B)$.

Πάντως αποδεικνύεται ότι: $S \vdash T$ αν και μόνο αν $S \models T$.

Και κάτι άλλο, ενδιαφέρον· αποδεικνύεται ότι: $S \vdash T$ αν και μόνον αν η $S \Rightarrow T$ είναι ταυτολογία.

Α.5 Κατηγορηματικός Λογισμός

Ο Προτασιακός Λογισμός έχει πολύ περιορισμένες δυνατότητες. Π.χ. δεν μπορούμε να ασχοληθούμε με προτάσεις όπως: «για κάθε πραγματικό x έχουμε $x^2 \geq 0$».

Ο **Κατηγορηματικός Λογισμός** (Κ.Λ. predicate calculus) επιτρέπει τη διαχείριση τύπων που περιέχουν σταθερές και μεταβλητές (που παίρνουν τιμές) από κάποιο σύνολο τιμών (π.χ. το $\mathbb{Z}$, το $\mathbb{R}$ κλπ). Έτσι, το παραπάνω παράδειγμα μπορεί να διατυπωθεί συμβολικά ως εξής:

$$\forall x \bullet x^2 \geq 0$$

Το σύμβολο "$\forall$" σημαίνει "για κάθε" και ονομάζεται **καθολικός ποσοδείκτης** (universal quantifier). Το x είναι μια μεταβλητή και το $x^2 \geq 0$ είναι ένα **κατηγόρημα** (predicate) μιας θέσης (μεταβλητής).

Μια άλλη κατηγορία προτάσεων είναι οι **υπαρξιακές** (existential), π.χ.: «υπάρχει μια τουλάχιστον τιμή της οποίας το τετράγωνο είναι μηδέν». Αυτή διατυπώνεται συμβολικώς ως εξής:

$$\exists x \bullet x^2 == 0$$

Το σύμβολο "$\exists$" σημαίνει "υπάρχει" και ονομάζεται **υπαρξιακός ποσοδείκτης** (existential quantifier). Το "**==**" σημαίνει "είναι ίσο":[5]

Στους δύο τύπους που γράψαμε παραπάνω μπορούμε να αλλάξουμε χωρίς κανένα πρόβλημα όνομα της μεταβλητής, π.χ.:

$$\forall y \bullet y^2 \geq 0 \quad \text{ή} \quad \exists t \bullet t^2 == 0$$

Λέμε ότι οι μεταβλητές αυτές είναι **δεσμευμένες** (bound) από τον ποσοδείκτη. Κοίταξε όμως τον τύπο:

$$\exists x \bullet x + f(x, y) > 0$$

Εδώ η x είναι δεσμευμένη, δηλαδή μπορούμε να γράψουμε: $\exists u \bullet u + f(u,y) > 0$ αλλά δεν συμβαίνει το ίδιο με τη y: η y είναι μια **ελεύθερη** (free) μεταβλητή.

Μετά το "$\bullet$" μπορεί να ακολουθεί, όχι μόνον κατηγόρημα, αλλά τύπος του Κ.Λ., π.χ.:

$$\forall y \bullet (\exists x \bullet x + y == 0)$$

που σημαίνει: για κάθε y υπάρχει x τέτοιο ώστε $x + y == 0$. Για τον τύπο $\exists x \bullet x + y == 0$ η x είναι δεσμευμένη μεταβλητή –και θα μπορούσαμε να της αλλάξουμε όνομα, π.χ.: $\exists z \bullet z + y == 0$– ενώ η y είναι ελεύθερη. Για τον αρχικό τύπο η y είναι δεσμευμένη και μπορούμε να

[5] Γιατί "**==**" και όχι "**=**"; Διότι το "**=**" στη C++ έχει διαφορετικό νόημα. Το "**==**" είναι ο τελεστής σύγκρισης για ισότητα της C++.

της αλλάξουμε όνομα. Θα μπορούσαμε να την ονομάσουμε x; Βεβαίως, αλλά δεν θα ήταν και πολύ έξυπνο διότι τότε ο τύπος σου θα γίνει: $\forall x \bullet (\exists x \bullet x + x == 0)$ και άντε να βγάλεις άκρη! Παρ' όλο που οι συντακτικοί κανόνες του Κ.Λ. επιτρέπουν τέτοιους τύπους, αυτοί δεν είναι δεκτοί.

Πολύ συχνά έχουμε τύπους όπως: $\forall x \bullet ((x > 0) \Rightarrow (-x < 0))$. Αυτός γράφεται και ως εξής: $\forall x\ (x > 0) \bullet (-x < 0))$ ή ακόμη: $\forall x > 0 \bullet (-x < 0))$ και λέμε ότι στην περίπτωση αυτή έχουμε **φραγμένο ποσοδείκτη**.

Γενικώς, στον Κ.Λ. δεν είναι δυνατόν να κάνουμε αποδείξεις με πίνακες, αφού θα πρέπει να εξετάσουμε όλες τις δυνατές τιμές των μεταβλητών μας (εκτός από την περίπτωση που το σύνολο τιμών των μεταβλητών μας είναι πολύ μικρό). Θα πρέπει λοιπόν οι αποδείξεις να γίνονται με βάση αξιώματα και συμπερασματικούς κανόνες. Για τον καθολικό ποσοδείκτη υπάρχουν οι εξής κανόνες:

$$\frac{\forall x \bullet P(x)}{P(a)} \text{ (για τυχαίο } a\text{)} \quad \text{και} \quad \frac{P(a)}{\forall x \bullet P(x)} \text{ (για τυχαίο } a\text{)}$$

Ο πρώτος μας λέει: αν (έχω αποδείξει ότι) για κάθε x ισχύει η $P(x)$ τότε για οποιοδήποτε (τυχαίο) a ισχύει η $P(a)$. Και ο δεύτερος: αν (έχω αποδείξει ότι) για κάποιο τυχαίο (χωρίς ιδιαίτερα χαρακτηριστικά) a ισχύει η $P(a)$ τότε μπορώ να συμπεράνω ότι για κάθε x ισχύει η $P(x)$.

Για τον υπαρξιακό ποσοδείκτη:

$$\frac{\exists x \bullet P(x)}{P(q)} \quad \text{και} \quad \frac{P(q)}{\exists x \bullet P(x)}$$

Ο πρώτος μας λέει: αν (έχω αποδείξει ότι) υπάρχει ένα τουλάχιστον x ώστε να ισχύει η $P(x)$ τότε ισχύει $P(q)$, όπου q κάποια από τις συγκεκριμένες τιμές. Και ο δεύτερος: αν (έχω αποδείξει ότι) για κάποιο q ισχύει η $P(q)$ τότε μπορώ να συμπεράνω ότι υπάρχει τουλάχιστον ένα x ώστε να έχω $P(x)$.

Α.6 Ισότητα

Έστω ότι σε ένα σύνολο τιμών έχουμε ορίσει την **ισότητα** (==). Στο σύνολο αυτό:

1. ορίζεται και η **ανισότητα** (≠) ως εξής: με την $a \neq b$ εννοούμε $!(a == b)$
2. θα ισχύει το αξίωμα της **ανακλαστικότητας**: $\forall x \bullet x == x$
3. και ακόμη οι συμπερασματικοί κανόνες της **αντικατάστασης**: Αν $S(n)$ είναι ένας τύπος που περιέχει το n –αλλά όχι ως δεσμευμένη μεταβλητή– και $m == n$, τότε, μπορείς να αντικαταστήσεις το n με το m και να πάρεις έναν τύπο $S[m/n]$:

$$\frac{m = n, S(n)}{S[m/n]} \text{ (A1)} \quad \text{και} \quad \frac{n = m, S(n)}{S[m/n]} \text{ (A2)}$$

Για παράδειγμα, αν

- $S(y)$ είναι η $\exists x \bullet x^2 + y == 0$ και έχουμε αποδείξει ότι ισχύει και
- $z == y$

τότε ισχύει και η:

- $\exists x \bullet x^2 + z == 0$

Όταν λέμε «μπορείς να αντικαταστήσεις το n με το m» δεν εννούμε όλες υποχρεωτικά τις περιπτώσεις που εμφανίζεται το n, αλλά μόνον όσες μας ενδιαφέρουν.

Παράδειγμα 1

Ας δούμε πώς μπορούμε να αποδείξουμε τη **συμμετρικότητα** της ισότητας, δηλαδή:

αν ισχύει η: $x == y$ τότε ισχύει και η: $y == x$

1. $x == y$ υπόθεση
2. $x == x$ αξίωμα της ανακλαστικότητας

Έχουμε ότι x == y (από το 1.) και θεωρούμε ως $S(x)$ την x == x του 2. Σύμφωνα με τον κανόνα (Α2) αν αντικαταστήσουμε *μόνον το πρώτο* x με y θα πάρουμε μια πρόταση που ισχύει:

3. y == x.

ό.έ.δ.

Παράδειγμα 2

Απόδειξε ότι:

αν ισχύει η: (!($k \neq N$))&& ((y == 2k+1) && (x == (k+1)²))

τότε ισχύει και η: (k == N)) && ((y == 2N+1) && (x == (N+1)²))

1. (!($k \neq N$))&&((y == 2k+1)&&(x == (k+1)²))	υπόθεση
2. !($k \neq N$)	1. και σ.κ. (3) της §Α.4
3. !(!(k == N))	2. και ορισμός της ≠
4. k == N	3. και $A \equiv$ (!(!A)) §Α.3
5. (y == 2k+1) && (x == (k+1)²)	1. και σ.κ. (4) της §Α.4
6. (y == 2N+1) && (x == (N+1)²)	4.,5. και σ.κ. (Α2)
7. (k == N) && ((y == 2N+1) && (x == (N+1)²)	4.,6. και σ.κ. (1) της §Α.4

ό.έ.δ.

Α.7 Οι Δικές μας Αποδείξεις

Στις αποδείξεις ορθότητας των προγραμμάτων δεν θα είμαστε και τόσο αυστηροί (αν και στο Παρ. 2, της προηγούμενης παραγράφου, στο βήμα 4 τα πράγματα δεν είναι και τόσο αυστηρά).

Θα ασχολούμαστε κατά κανόνα με τύπους του Κ.Λ. και οι μεταβλητές μας θα παίρνουν τιμές στα σύνολα `int` (υποσύνολο του $\mathbb{Z}$) και `double` (υποσύνολο του $\mathbb{R}$).

Συχνά θα μας χρειάζονται οι ισοδυναμίες:

$$(a == b) \equiv (!(a \neq b))$$
$$(a \neq b) \equiv (!(a == b)) \equiv ((a < b) \;||\; (a > b))$$
$$(!(a < b)) \equiv (a \geq b) \equiv ((a > b) \;||\; (a == b))$$
$$(!(a > b)) \equiv (a \leq b) \equiv ((a < b) \;||\; (a == b))$$
$$((a \geq b) \;\&\&\; (a \leq b)) \equiv (a == b)$$
$$((a \geq b) \;||\; (a < b)) \equiv \texttt{true} \text{ και } ((a > b) \;||\; (a \leq b)) \equiv \texttt{true} \quad [(A \;||\; (!A)) \equiv \texttt{true}]$$
$$((a \geq b) \;\&\&\; (a < b)) \equiv \texttt{false} \text{ και } ((a > b) \;\&\&\; (a \leq b)) \equiv \texttt{false} \quad [(A \;\&\&\; (!A)) \equiv \texttt{false}]$$
$$(A \;\&\&\; (B \;||\; C)) \equiv ((A \;\&\&\; B) \;||\; (A \;\&\&\; C))$$
$$(A \;||\; (B \;\&\&\; C)) \equiv ((A \;||\; B) \;\&\&\; (A \;||\; C))$$
$$(!(A \;||\; B)) \equiv ((!A) \;\&\&\; (!B))$$
$$(!(A \;\&\&\; B)) \equiv ((!A) \;||\; (!B))$$

και οι συμπερασματικοί κανόνες (1)-(6) της §Α.4.

Ασκήσεις

Α-1 Απόδειξε ότι:

α) $(A \Rightarrow B) \equiv ((!\,A) \;||\; B)$ $\quad$ β) $A \Rightarrow (A \;||\; B)$ και $B \Rightarrow (A \;||\; B)$

Α-2 Απόδειξε ότι:

α) (A **xor** B) ≡ ((A || B) && (!(A && B)))

β) (A **xor** B) ≡ ((A && (!B)) || ((!A) && B))

γ) (A **xor** B) ≡ (!($A \Leftrightarrow B$))

παράρτημα

Β

Μαθηματικές Συναρτήσεις

Βάζοντας στο πρόγραμμά σου την οδηγία "`#include <cmath>`" μπορείς να χρησιμοποιήσεις τις μαθηματικές συναρτήσεις που σου δίνει η C++ και τις έχει κληρονομήσει από τη C.

Στον πίνακα, στις επόμενες δύο σελίδες, αντιγράψαμε τα χαρακτηριστικά αυτών των συναρτήσεων.

Όπως θα δεις, όλες σχεδόν οι συναρτήσεις υπάρχουν σε δύο μορφές: μια για τον τύπο `double` και μια για τον τύπο `long double`. Η *abs()* –για την απόλυτη τιμή– υπάρχει σε τέσσερις διαφορετικούς τύπους: *abs()* (`int`), *labs()* (`long`), *fabs()* (`double`), *fabsl()* (`long double`).

Να πώς θα διαβάζεις τον πίνακα: Στη στήλη 1 (Τι δίνει) βλέπεις την τιμή της συνάρτησης, π.χ. «τόξο εφαπτομένης». Στη στήλη 2 βλέπεις το ίδιο πράγμα συμβολικά, π.χ. «$τοξεφ\frac{a_1}{a_2}$». Στη στήλη 3 βλέπεις το πλήθος ορισμάτων, π.χ. «2». Αν το πλήθος είναι 1, στη στήλη 2 το όρισμα παριστάνεται ως *a*. Αν το πλήθος είναι 2, τότε το πρώτο όρισμα παριστάνεται ως a_1 και το δεύτερο ως a_2. Στις επόμενες τρεις στήλες, για το παράδειγμά μας βλέπεις τα εξής:

- Υπάρχει μια συνάρτηση με όνομα *atan2* (στ. 4), που παίρνει δύο ορίσματα τύπου `double` (στ. 5) και επιστρέφει τιμή τύπου `double` (στ. 6). Ακόμη:
- Υπάρχει μια συνάρτηση με όνομα *atan2l* (στ. 4), που παίρνει δύο ορίσματα τύπου `long double` (στ. 5) και επιστρέφει τιμή τύπου `long double` (στ. 6).

Αριθμητικές Συναρτήσεις της C++
Επικεφαλίδες στο cmath

Τι δίνει	Ορισμός	Πλ. Ορισ	Όνομα	Τύπος Ορίσματος	Τύπος Συνάρτησης

απόλυτη τιμή	$\lvert a \rvert$	1	*abs*
			labs
			fabs
			fabsl
				`int`
				`long`
				`double`
				`long double`
					`int`
					`long`
					`double`
					`long double`

στρογγ. στον πλησ. μικρότ. ακέραιο	$\lfloor a \rfloor$	1	*floor* *floorl*	`double` `long double`	`double` `long double`
στρογγ. στον πλησ. μεγαλ. ακέραιο.	$\lceil a \rceil$	1	*ceil* *ceill*	`double` `long double`	`double` `long double`
υπόλοιπο	a_1 **mod** a_2	2	*fmod* *fmodl*	`double` `long double`	`double` `long double`
τετραγ. ρίζα	$\sqrt{a}$	1	*sqrt* *sqrtl*	`double` `long double`	`double` `long double`
υποτείνουσα	$\sqrt{a_1^2 + a_2^2}$	2	*hypotl* *hypotl*	`double` `long double`	`double` `long double`
ημίτονο	ημ*a*	1	*sin* *sinl*	`double` `long double`	`double` `long double`
συνημίτονο	συν*a*	1	*cos* *cosl*	`double` `long double`	`double` `long double`
εφαπτομένη	εφ*a*	1	*tan* *tanl*	`double` `long double`	`double` `long double`
τόξο ημιτόνου	τοξημ*a*	1	*asin* *asinl*	`double` `long double`	`double` `long double`
τόξο συνημιτόνου	τοξσυν*a*	1	*acos* *acosl*	`double` `long double`	`double` `long double`
τόξο εφαπτομένης	τοξεφ*a*	1	*atan* *atanl*	`double` `long double`	`double` `long double`
τόξο εφαπτομένης	$\tau o \xi \varepsilon \varphi \frac{a_1}{a_2}$	2	*atan2* *atan2l*	`double` `long double`	`double` `long double`
εκθετική	e^a	1	*exp* *expl*	`double` `long double`	`double` `long double`
υπερβολικό ημίτονο	sinh*a*	1	*sinh* *sinhl*	`double` `long double`	`double` `long double`
υπερβολικό συνημίτονο	cosh*a*	1	*cosh* *coshl*	`double` `long double`	`double` `long double`
υπερβολική εφαπτομένη	tanh*a*	1	*tanh* *tanhl*	`double` `long double`	`double` `long double`
φυσικός λογάριθμος	`ln`*a*	1	*log* *logl*	`double` `long double`	`double` `long double`
δεκαδικός λογάριθμος	λογ*a*	1	*log10* *log10l*	`double` `long double`	`double` `long double`
δύναμη	$a_1^{a_2}$	2	*pow* *powl*	`double` `long double`	`double` `long double`
δύναμη του 10	10^a	1	*pow10* *pow10l*	`int` `int`	`double` `long double`

παράρτημα

C

Λέξεις-Κλειδιά της C++

Στον Πίν. C-1, παρακάτω, βλέπεις τις λέξεις-κλειδιά της C++ (που δεν επιτρέπεται να χρησιμοποιηθούν ως ονόματα) όπως δίνονται στο (ISO 2011). Οι λέξεις που σημειώνονται με αστερίσκο δεν υπήρχαν στο (ISO 2003).

Πίν. C-1: Λέξεις-κλειδιά της C++

alignas *	continue	friend	register	true
alignof *	decltype *	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert *	using
char	enum	new	static_cast	virtual
char16_t *	explicit	noexcept *	struct	void
char32_t *	export	nullptr *	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr *	float	protected	thread_local *	
const_cast	for	public	throw	

Στον επόμενο Πίν. C-2 δίνουμε λέξεις-κλειδιά της C++, που δεν υπήρχαν στη C και έχουν σχέση με λογικές πράξεις και ψηφιοπράξεις. Είναι πολύ πιθανόν ο μεταγλωττιστής που χρησιμοποιείς να μην αναγνωρίζει αυτές τις λέξεις. Η δυνατότητα παράστασης των τελεστών με λέξεις δόθηκε διότι σε ορισμένα εθνικά σύνολα χαρακτήρων, μερικά σύμβολα τελεστών έχουν αντικατασταθεί από γράμματα.

Πίν. C-2: Εναλλακτικά Λεξικά Σύμβολα της C++

εναλλακτικό	αρχικό	εναλλακτικό	αρχικό	εναλλακτικό	αρχικό
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

Όπως βλέπεις όμως, στην πρώτη στήλη η εναλλακτική παράσταση δεν γίνεται με λέξεις αλλά με ζεύγη χαρακτήρων. Υπάρχουν και εναλλακτικές παραστάσεις με τριάδες χαρακτήρων. Τις βλέπεις στον πίνακα που ακολουθεί.

Πίν. C-3: Παράσταση Λεξικών Συμβόλων με 2 ή 3 Χαρακτήρες

2 χαρακτ	3 χαρακτ	αρχικό	2 χαρακτ	3 χαρακτ	αρχικό	2 χαρακτ	3 χαρακτ	αρχικό
%:	??=	#	<:	??(	[	<%	??<	{
	??/	\	:>	??)	]	%>	??>	}
	??'	^		??!	|		??-	~

παράρτημα

Δ

Πρότυπα Σύνολα Χαρακτήρων

Περιεχόμενα:

D.1 ISO 646 (ASCII)

Το γνωστό ASCII (American Standard Code for Information Interchange) είναι μια εθνική εκδοχή του διεθνούς προτύπου ISO 646. Αποτελείται από 128 χαρακτήρες που αριθμούνται από το 0 μέχρι το 127. Οι πρώτοι 32 χαρακτήρες (0 .. 31) και ο τελευταίος (127) δεν είναι εκτυπώσιμοι (Πίν. Δ-1).

Στις θέσεις 48 μέχρι 57 βρίσκονται τα ψηφία του δεκαδικού συστήματος, κατ' αύξουσα τάξη:

`static_cast<char>`(48)=`'0'` και `static_cast<char>`(57)=`'9'`

Στις θέσεις 65 μέχρι 90 βρίσκονται τα κεφαλαία γράμματα του λατινικού αλφαβήτου σε αλφαβητική διάταξη. Στις θέσεις 97 μέχρι 122 βρίσκονται τα πεζά γράμματα του λατινικού αλφαβήτου. Η θέση ενός κεφαλαίου γράμματος από το αντίστοιχο πεζό διαφέρει κατά 32:

`static_cast<int>(`κεφαλαίο`) == static_cast<int>(`πεζό`)-`32

Οι χαρακτήρες των θέσεων 0 μέχρι 31 δίνονται από το πληκτρολόγιο ως εξής: με πιεσμένο το πλήκτρο <Ctrl> πιέζουμε τον χαρακτήρα που βρίσκεται στο σύνολο μετά από 64 θέσεις. Π.χ. αν θέλουμε να δώσουμε τον χαρακτήρα <ETX> = `static_cast<char>`(3), με πιεσμένο το πλήκτρο <Ctrl> πιέζουμε το πλήκτρο <C> ('C' = `static_cast<char>`(67)). Αυτή η πληκτρολόγηση συμβολίζεται και ως <Ctrl-C> ή "^C". Το τι θα πετύχεις με μια τέτοια πληκτρολόγηση δεν καθορίζεται από κανένα πρότυπο, αλλά εξαρτάται από το πρόγραμμα που θα δεχτεί την πληκτρολόγηση.

Αν προσπαθήσεις να «γράψεις» μερικούς από αυτούς τους χαρακτήρες, δεν θα δεις κάποιο σύμβολο να εμφανίζεται στην οθόνη σου αλλά θα κάποια άλλη λειτουργία. Π.χ. συνηθέστατα γράψιμο των παρακάτω χαρακτήρων έχουν τα εξής αποτελέσματα:

BEL (`static_cast<char>`(7)):

«Γράφοντας» αυτόν τον χαρακτήρα –`cout<<static_cast<char>(7)` ή `cout << '\a'`– στην οθόνη του τερματικού μας, μάλλον δεν θα δούμε κάτι αλλά θα το ακούσουμε: θα ηχήσει το μεγάφωνό του (alert).

BS (`static_cast<char>`(8)):

«Γράψιμο» αυτού του χαρακτήρα –`cout<<static_cast<char>(8)` ή `cout << '\b'`– έχει ως αποτέλεσμα να σβηστεί ο προηγούμενος χαρακτήρας που είχε γραφεί (Back Space).

```
  0 NUL  16 DLE 32 SP  48 0   64 @   80 P   96 `  112 p
  1 SOH  17 DC1 33 !   49 1   65 A   81 Q   97 a  113 q
  2 STX  18 DC2 34 "   50 2   66 B   82 R   98 b  114 r
  3 ETX  19 DC3 35 #   51 3   67 C   83 S   99 c  115 s
  4 EOT  20 DC4 36 $   52 4   68 D   84 T  100 d  116 t
  5 ENQ  21 NAK 37 %   53 5   69 E   85 U  101 e  117 u
  6 ACK  22 SYN 38 &   54 6   70 F   86 V  102 f  118 v
  7 BEL  23 ETB 39 '   55 7   71 G   87 W  103 g  119 w
  8 BS   24 CAN 40 (   56 8   72 H   88 X  104 h  120 x
  9 HT   25 EM  41 )   57 9   73 I   89 Y  105 i  121 y
 10 LF   26 SUB 42 *   58 :   74 J   90 Z  106 j  122 z
 11 VT   27 ESC 43 +   59 ;   75 K   91 [  107 k  123 {
 12 FF   28 FS  44 ,   60 <   76 L   92 \  108 ℓ  124 |
 13 CR   29 GS  45 -   61 =   77 M   93 ]  109 m  125 }
 14 SO   30 RS  46 .   62 >   78 N   94 ^  110 n  126 ~
 15 SI   31 US  47 /   63 ?   79 O   95 _  111 o  127 DEL
```

Πίν. Δ-1 Πρότυπο Σύνολο Χαρακτήρων ISO 646 (ASCII).

HT (`static_cast<char>`(9)):
Με `cout<<static_cast<char>(9)` ή `cout << '\t'` συνεχίζεις το γράψιμο από τον επόμενο οριζόντιο στηλοθέτη (Horizontal Tab).

LF (`static_cast<char>`(10)):
Με `cout<<static_cast<char>(10)` ή `cout << '\n'` κατεβάζεις τον οθονοδείκτη στην επόμενη γραμμή (Line Feed).

VT (`static_cast<char>`(11)):
Με `cout<<static_cast<char>(11)` ή `cout << '\v'`– συνεχίζεις το γράψιμο από τον επόμενο κατακόρυφο στηλοθέτη (Vertical Tab).

FF (`static_cast<char>`(12)):
Με `cout<<static_cast<char>(12)` ή `cout << '\f'` καθαρίζεις την οθόνη (στον εκτυπωτή αλλάζεις σελίδα) (Form Feed).

CR (`static_cast<char>`(13)):
Με `cout << static_cast<char>(13)` ή `cout << '\r'` φέρνεις τον οθονοδείκτη στην αρχή της γραμμής (Carriage Return).

DEL (`static_cast<char>`(127)):
Σβήνει τη γραμμή που βρίσκεται (DElete Line).

Ο `static_cast<char>`(26) (<Ctrl>Z), στο MS DOS, σημειώνει το τέλος ενός αρχείου text. Σε άλλα συστήματα το ίδιο πράγμα γίνεται με τον <ETX> (`static_cast<char>`(3) = ^C - End of TeXt).

Τα συμβολικά ονόματα των χαρακτήρων ελέγχου ξεκινούν από τη λειτουργία τους στις επικοινωνίες.

D.2 Πρότυπο ΕΛΟΤ 928

Το Πρότυπο ΕΛΟΤ 928 (Πίν. Δ-3), βγήκε από τη Τεχνική Επιτροπή 48 που λειτουργεί με ευθύνη των ΕΛΟΤ και ΕΛΚΕΠΑ.

Στις θέσεις 0..31 και 127..159 δεν υπάρχουν εκτυπώσιμοι χαρακτήρες, σύμφωνα με τους κανόνες του ISO και με τα πρότυπα της σειράς ISO 8859. Θα τονίσουμε την παρουσία δυο χαρακτήρων:

NBSP (`static_cast<char>`(160)): No-Break SPace = Διάστημα Χωρίς Διακοπή. "Γράφεται" όπως το *διάστημα*. Η παρουσία του δείχνει σε προγράμματα **μορφοποίησης κειμένων** (text formatters) ότι στο σημείο αυτό δεν θα πρέπει να γίνει διακοπή του κειμένου με αλλαγή γραμμής.

SHY (`static_cast<char>`(173)): Soft HYphen = Μη-Σταθερό Ενωτικό. Γράφεται όπως η παύλα (`static_cast<char>`(45)). Η παρουσία του δείχνει σε προγράμματα μορφοποίησης

0	NUL	32	SP	64	@	96	`	128	□	160	NBSP	192	Ἱ	224	ὗ
1	SOH	33	!	65	A	97	a	129	□	161	‘	193	A	225	α
2	STX	34	"	66	B	98	b	130	□	162	’	194	B	226	β
3	ETX	35	#	67	C	99	c	131	□	163	£	195	Γ	227	γ
4	EOT	36	$	68	D	100	d	132	□	164	¤	196	Δ	228	δ
5	ENQ	37	%	69	E	101	e	133	□	165	¥	197	E	229	ε
6	ACK	38	&	70	F	102	f	134	□	166	¦	198	Z	230	ζ
7	BEL	39	'	71	G	103	g	135	□	167	§	199	H	231	η
8	BS	40	(	72	H	104	h	136	□	168	¨	200	Θ	232	θ
9	HT	41	)	73	I	105	i	137	□	169	©	201	I	233	ι
10	LF	42	*	74	J	106	j	138	□	170	□	202	K	234	κ
11	VT	43	+	75	K	107	k	139	□	171	«	203	Λ	235	λ
12	FF	44	,	76	L	108	ℓ	140	□	172	¬	204	M	236	μ
13	CR	45	-	77	M	109	m	141	□	173	SHY	205	N	237	ν
14	SO	46	.	78	N	110	n	142	□	174	®	206	Ξ	238	ξ
15	SI	47	/	79	O	111	o	143	□	175	―	207	O	239	ο
16	DLE	48	0	80	P	112	p	144	□	176	°	208	Π	240	π
17	DC1	49	1	81	Q	113	q	145	□	177	±	209	P	241	ρ
18	DC2	50	2	82	R	114	r	146	□	178	²	210	□	242	ς
19	DC3	51	3	83	S	115	s	147	□	179	³	211	Σ	243	σ
20	DC4	52	4	84	T	116	t	148	□	180	΄	212	T	244	τ
21	NAK	53	5	85	U	117	u	149	□	181	΅	213	Y	245	υ
22	SYN	54	6	86	V	118	v	150	□	182	Ά	214	Φ	246	φ
23	ETB	55	7	87	W	119	w	151	□	183	·	215	X	247	χ
24	CAN	56	8	88	X	120	x	152	□	184	Έ	216	Ψ	248	ψ
25	EM	57	9	89	Y	121	y	153	□	185	Ή	217	Ω	249	ω
26	SUB	58	:	90	Z	122	z	154	□	186	Ί	218	Ϊ	250	ϊ
27	ESC	59	;	91	[	123	{	155	□	187	»	219	Ÿ	251	ü
28	FS	60	<	92	\	124	|	156	□	188	Ό	220	ὰ	252	ὸ
29	GS	61	=	93	]	125	}	157	□	189	½	221	έ	253	ὺ
30	RS	62	>	94	^	126	~	158	□	190	Ύ	222	ή	254	ώ
31	US	63	?	95	_	127	DEL	159	□	191	Ώ	223	ί	255	□

Πίν. Δ-2 Πρότυπο Σύνολο Χαρακτήρων ΕΛΟΤ 928.

κειμένων ότι στο σημείο αυτό μπορεί να εισαχθεί ή να αφαιρεθεί μια παύλα χωρισμού λέξης (τέλος συλλαβής).

Στο περιβάλλον MS Windows χρησιμοποιείται σύνολο χαρακτήρων που είναι παραλλαγή του ΕΛΟΤ 928 (Πίν. Δ-4). Οι διαφορές βρίσκονται κυρίως στη θέση του 'Ά' και στο ότι στις «απαγορευμένες θέσεις» 128 .. 159 υπάρχουν εκτυπώσιμοι χαρακτήρες. Στο περιβάλλον Windows χρησιμοποιούνται πολυγλωσσικοί πίνακες χαρακτήρων (με βάση τα πρότυπα ISO 8859 και τις ταυτότητες των συμβόλων) και η προσαρμογή τους σε συγκεκριμένη γλώσσα γίνεται με κατάλληλα προγράμματα.

D.3 Πρότυπο ΕΛΟΤ 927

Το Πρότυπο ΕΛΟΤ 927 έγινε για να καλύψει τις ανάγκες παλιών ΗΥ και εκτυπωτών που έχουν σύνολο 128 χαρακτήρων και πρέπει να έχουν δυνατότητα εκτύπωσης (τουλάχιστον κεφαλαίων) ελληνικών και λατινικών χαρακτήρων. Πρακτικά έχει πια αχρηστευθεί.

παράρτημα

Ε

Η Προτεραιότητα των Πράξεων

Στον πίνακα, που δίνεται στις επόμενες σελίδες, βλέπεις τα χαρακτηριστικά των πράξεων που έχουμε μάθει μέχρι τώρα. Οι πράξεις δίνονται κατά τη σειρά προτεραιότητας.

Δεν βλέπεις την προσεταιριστικότητα των πράξεων, αλλά γι' αυτήν ο κανόνας είναι απλός:

- η προσεταιριστικότητα των δυϊκών πράξεων, εκτός από την εκχώρηση, είναι από τα αριστερά προς τα δεξιά,
- η προσεταιριστικότητα των ενικών πράξεων και της εκχώρησης είναι από τα δεξιά προς τα αριστερά.

Για παράδειγμα: η `a + b + c` υπολογίζεται ως `(a + b) + c`, ενώ η `a = b = c` υπολογίζεται ως `a = (b = c)`.

Αλλά προσοχή! Αν έχουμε `a*b + c/d + f(e)` ο παραπάνω κανόνας μας λέει *μόνον* ότι ο υπολογισμός θα γίνει ως εξής: `(a*b + c/d) + f(e)`.

- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `(a*b + c/d)` και μετά η `f(e)`.
- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `a*b` και μετά η `c/d`.

Κάθε υλοποίηση της C++ θα κάνει τους υπολογισμούς με τη σειρά που "θέλει". Αν θέλεις να επιβάλεις συγκεκριμένη σειρά εκτέλεσης των πράξεων θα πρέπει να σπάσεις την παράσταση σε μικρότερες και να τις γράψεις με την κατάλληλη σειρά.

Όταν υπολογίζεται μια αριθμητική παράσταση γίνονται και ορισμένες μετατροπές τύπων, που θα τις ονομάζουμε *Συνήθεις Αριθμητικές Μετατροπές* (ΣΑΜ). Ας πούμε ότι έχουμε κάποια πράξη a *θ* b. Τότε:

1. Αν κάποια από τις τιμές a, b είναι τύπου `long double`, τότε μετατρέπεται στον τύπο `long double` και η άλλη τιμή.
2. Αλλιώς, αν κάποια από τις τιμές a, b είναι τύπου `double`, τότε μετατρέπεται στον τύπο `double` και η άλλη τιμή.
3. Αλλιώς, αν κάποια από τις τιμές a, b είναι τύπου `float`, τότε μετατρέπεται στον τύπο `float` και η άλλη τιμή.
4. Αλλιώς, τα a και b (είναι ακέραιου τύπου) υφίστανται **ακέραιη προαγωγή** (integral promotion), δηλαδή: αν κάποια (ή και οι δύο) από τις τιμές a, b είναι "μικρού ακέραιου τύπου" (`char`, `short int`, `signed` ή `unsigned`), αυτή μετατρέπεται σε τύπο `int` ή `unsigned int` (αν δεν μπορεί να παρασταθεί στον `int`) και στη συνέχεια

   αν κάποια από τις τιμές a, b είναι τύπου `unsigned long`, τότε

   μετατρέπεται στον τύπο `unsigned long` και η άλλη τιμή.

Πράξεις και Προτεραιότητες		
Προτερ.	**Τελεστής**	**Σύμβολα**
0	Παράστ. σε παρενθέσεις	`(`*παράσταση*`)`
1	Επίλυση Εμβέλειας Καθολικό	*όνομα κλάσης*`::`*μέλος* *όνομα ονοματοχώρου*`::`*μέλος* `::`*αναγνωριστικό*

2	Επιλογή μέλους
	Στοιχείο Πίνακα (δείκτης)
	Κλήση Συνάρτησης
	Κατασκευή τιμής
	Μεταθεματικές Παραστάσεις
	Χαρακτηριστικά Τύπου
	Ελεγχ. μετατρ. τύπου (εκτ)
	Μετατροπή τύπου (μετγλ.)
	Μη ελεγχ. μετατρ. τύπου(εκτ)
	Μετατροπή const
		αντικείμενο`.`*μέλος*, *βέλος*`->`*μέλος*
		βέλος`[]`
		όνομα`(`*λίστα παραστάσεων*`)`
		τύπος`(`*λίστα παραστάσεων*`)`
		τιμή-`l++`
		τιμή-`l--`
		`typeid(`*παράσταση*`)`, `typeid(`*τύπος*`)`
		`dynamic_cast<` *τύπος*`>(`*παράσταση*`)`
		`static_cast<` *τύπος*`>(`*παράσταση*`)`
		`reinterpret_cast<`*τύπος*`>(`*παράσταση*`)`
		`const_cast<`*τύπος*`>(`*παράσταση*`)`

3	Μέγεθος
	Προθεματικές Παραστ.
	Συμπλήρωμα (ψηφιοπρ.)
	Λογική Άρνηση
	Πρόσημα
	Διεύθυνση
	Αποπαραπομπή
	Δημιουργία Δυν. Μετ.
	Δημ Δυν. Μετ. με αρχ. τιμή
	Δημιουργία σε θέση
	Δημ. σε θέση με αρχ. τιμή
	Καταστροφή
	Καταστροφή πίνακα
	Τυποθεώρηση (C)
		`sizeof` *παράσταση*, `sizeof(`*τύπος*`)`
		`++` *τιμή*-`l`, `--` *τιμή*-`l`
		`~` *παράσταση*
		`!` *παράσταση*
		`-` *παράσταση*, `+` *παράσταση*
		`&` *όνομα*
		`*` *παράσταση*
		`new` *τύπος*
		`new` *τύπος* `(`*παράσταση*`)`
		`new (`*παράσταση*`)` *τύπος*
		`new (`*παράσταση*`)` *τύπος* `(`*παράσταση*`)`
		`delete` *βέλος*
		`delete []` *βέλος*
		`(`*όνομα*`)` *παράσταση*

4	Επιλογή μέλους	*αντικείμενο*`.*`*βέλος προς μέλος* *βέλος*`->*`*βέλος προς μέλος*
5	Πολλαπλασιασμός Διαίρεση Υπόλοιπο	*παράσταση* `*` *παράσταση* *παράσταση* `/` *παράσταση* *παράσταση* `%` *παράσταση*
6	Πρόσθεση Αφαίρεση	*παράσταση* `+` *παράσταση* *παράσταση* `-` *παράσταση*

5. Αλλιώς, αν κάποια από τις τιμές a, b είναι τύπου `long int` και η άλλη `unsigned int`, τότε

   αν κάθε τιμή `unsigned int` μπορεί να παρασταθεί σε `long int`

   η τιμή `unsigned int` μετατρέπεται στον τύπο `long int`

   αλλιώς

   και οι δύο μετατρέπονται σε `unsigned long int`.

6. Αλλιώς, αν κάποια από τις τιμές a, b είναι τύπου `long`, τότε μετατρέπεται σε `long` και η άλλη τιμή.

7. Αλλιώς, αν κάποια από τις τιμές a, b είναι τύπου `unsigned`, τότε μετατρέπεται στον τύπο `unsigned` και η άλλη τιμή.

8. Αλλιώς και οι δυο τιμές είναι τύπου `int`.

Πράξεις και Προτεραιότητες (συνέχεια)		
Προτερ.	**Τελεστής**	**Σύμβολα**
5	Πολλαπλασιασμός Διαίρεση Υπόλοιπο	*παράσταση* ***** *παράσταση* *παράσταση* **/** *παράσταση* *παράσταση* **%** *παράσταση*
6	Πρόσθεση Αφαίρεση	*παράσταση* **+** *παράσταση* *παράσταση* **-** *παράσταση*
7	Ολίσθηση αριστερά Ολίσθηση δεξιά	*παράσταση* **<<** *παράσταση* *παράσταση* **>>** *παράσταση*
8	Μικρότερο Μικρότερο ή ίσο Μεγαλύτερο Μεγαλύτερο ή ίσο	*παράσταση* **<** *παράσταση* *παράσταση* **<=** *παράσταση* *παράσταση* **>** *παράσταση* *παράσταση* **>=** *παράσταση*
9	Ίσο Διάφορο	*παράσταση* **==** *παράσταση* *παράσταση* **!=** *παράσταση*
10	Ψηφιοπράξη **and**	*παράσταση* **&** *παράσταση*
11	Ψηφιοπράξη **xor**	*παράσταση* **^** *παράσταση*
12	Ψηφιοπράξη **or**	*παράσταση* **|** *παράσταση*
13	Λογικό **and**	*παράσταση* **&&** *παράσταση*
14	Λογικό **or**	*παράσταση* **||** *παράσταση*
15	Απλή εκχώρηση Πολλαπλασιαστική Εκχ. Διαιρετική Εκχώρηση Διαίρεση Υπολοίπου Προσθετική Εκχώρηση Αφαιρετική Εκχώρηση Εκχώρ. με αριστ. ολίσθηση Εκχώρ. με δεξιά ολίσθηση Εκχώρ. με ψηφιοπρ. **and** Εκχώρ. με ψηφιοπρ. **or** Εκχώρ. με ψηφιοπρ. **xor**	*τιμή-l* **=** *παράσταση* *τιμή-l* ***=** *παράσταση* *τιμή-l* **/=** *παράσταση* *τιμή-l* **%=** *παράσταση* *τιμή-l* **+=** *παράσταση* *τιμή-l* **-=** *παράσταση* *τιμή-l* **<<=** *παράσταση* *τιμή-l* **>>=** *παράσταση* *τιμή-l* **&=** *παράσταση* *τιμή-l* **|=** *παράσταση* *τιμή-l* **^=** *παράσταση*
16	Υπό Συνθήκη	*παράσταση* **?** *παράσταση* **:** *παράσταση*
17	Εξαίρεση	**throw** *παράσταση*
18	Ακολουθία Παραστάσεων	*παράσταση* **,** *παράσταση*

Οι ΣΑΜ εφαρμόζονται όταν γίνονται οι πράξεις **+**, **-**, *****, **/** μεταξύ αριθμητικών τιμών και καθορίζουν τον τύπο του αποτελέσματος. Ακέραιη προαγωγή γίνεται και στις ενικές πράξεις **+**, **-** (πρόσημα). Ο τύπος του αποτελέσματος είναι αυτός που προκύπτει από την προώθηση.

Ευρετήρια

Ευρετήριο

Αγγλοελληνικό Λεξικό Όρων

Συναρτήσεις – Περιγράμματα Συναρτήσεων

Η ένδειξη (C++) σημαίνει ότι η συνάρτηση είναι από βιβλιοθήκη της C++ ή της C.

Τύποι – Κλάσεις – Περιγράμματα Κλάσεων

Η ένδειξη (C++) σημαίνει ότι η κλάση είναι από βιβλιοθήκη της C++.

ΥΠΟΥΡΓΕΙΟ ΕΘΝΙΚΗΣ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ ΕΠΕΑΕΚ
ΕΥΡΩΠΑΪΚΗ ΕΝΩΣΗ
ΣΥΓΧΡΗΜΑΤΟΔΟΤΗΣΗ
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
ΕΥΡΩΠΑΪΚΟ ΤΑΜΕΙΟ ΠΕΡΙΦΕΡΕΙΑΚΗΣ ΑΝΑΠΤΥΞΗΣ

ΕΛΛΑΔΑ 2008
Ανάπτυξη παντού. Ανάπτυξη για όλους.

Η ΠΑΙΔΕΙΑ ΣΤΗΝ ΚΟΡΥΦΗ
Επιχειρησιακό Πρόγραμμα
Εκπαίδευσης και Αρχικής
Επαγγελματικής Κατάρτισης