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:

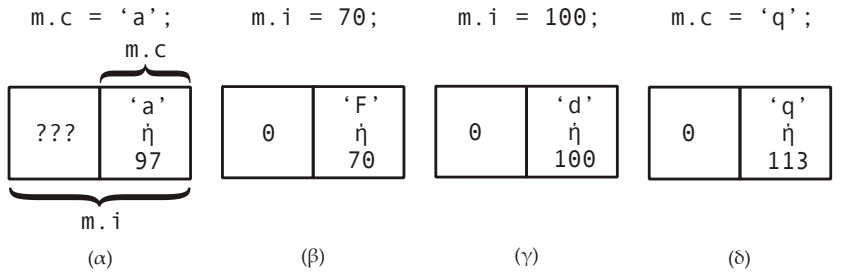
```
   m.c = 'a';  m.i = 70;
   cout << m.c << endl;
```

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:

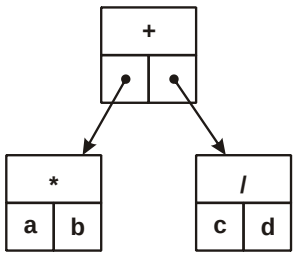
```
a * b + c / d
```

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.