# κεφάλαιο

# 11

# Πράξεις και Εντολές

**Ο στόχος μας σε αυτό το κεφάλαιο:**

- Να κάνουμε μια επανάληψη των πράξεων και των εντολών που έχουμε μάθει στο *Μέρος Α*.
- Να δούμε ιδιότητές τους που είχαμε αποσιωπίσει στο *Μέρος Α*.
- Να μάθουμε και μερικές εντολές που δεν τις μάθαμε καθόλου.

**Προσδοκώμενα αποτελέσματα:**

Όταν θα έχεις μελετήσει αυτό το τεύχος θα μπορείς να:

- Χρησιμοποιείς αποδοτικότερα τις δυνατότητες που σου προσφέρει η C++.
- Αποφεύγεις τις
  - «απαγορευμένες» εντολές,
  - «απαγορευμένες» χρήσεις εντολών

**Έννοιες κλειδιά:**

- *εκχώρηση*
- *παράσταση υπό συνθήκη*
- *εντολή* `for`
- *εντολή* `do-while`
- *επανάληψη n+½*
- *εντολή* `break`
- *εντολή* `switch`
- *ετικέτες - εντολή* `goto`

**Περιεχόμενα:**


**11.1 Εκτελέσιμες Δηλώσεις ........ 292**
**11.2 Περιορισμός Τύπου ........ 294**
**11.3 Όχι «Εντολή Εκχώρησης» αλλά «Πράξη Εκχώρησης» ........ 294**
11.3.1 * Ο “++” για τον Τύπο `bool` ........ 297
**11.4 Οι Εκχωρήσεις και οι Συνθήκες Επαλήθευσης ........ 297**
**11.5 Παράσταση Υπό Συνθήκη ........ 298**
**11.6 Η Εντολή “`for`” ........ 299**
**11.7 Η Εντολή “`do-while`” ........ 302**
**11.8 Η Επανάληψη “n+½” - Η Εντολή “`break`” ........ 304**
**11.9 Η Εντολή “`switch`” ........ 305**
11.9.1 Τοπικές Μεταβλητές στη “`switch`” ........ 307
**11.10 * Ετικέτες - Η Εντολή “`goto`” ........ 308**
11.10.1 Προβλήματα με τη Χρήση της Εντολής `goto` ........ 309
**11.11 * Η Εντολή “`continue`” ........ 310**
**11.12 * Ακολουθία Παραστάσεων ........ 310**

**11.13 Υπολογισμός Παράστασης ........ 311**
**11.14 Εν Κατακλείδι... ........ 312**
**Ασκήσεις ........ 312**
**Α Ομάδα ........ 312**

**Εισαγωγικές Παρατηρήσεις:**
Στο *Μέρος Α* μάθαμε τις βασικές έννοιες και τεχνικές προγραμματισμού και τις εντολές που μας χρειάζονται για να τις υλοποιήσουμε. Συνοπτικώς μάθαμε τις εξής εντολές:

- Εντολή εισόδου, με την οποία διαβάζουμε τιμές μεταβλητών από το πληκτρολόγιο ή από κάποιο αρχείο.
- Εντολή εξόδου, με την οποία γράφουμε τις τιμές παραστάσεων στην οθόνη ή σε κάποιο αρχείο.
- Εντολή εκχώρησης, με την οποία εκχωρούμε την τιμή κάποιας παράστασης σε μια μεταβλητή.
- Δήλωση, με την οποία παραχωρείται στο πρόγραμμά μας μνήμη για την υλοποίηση μεταβλητής και –αν το ζητήσουμε– ορίζεται η αρχική τιμή της.
- Εντολές επιλογής **if**, **ifelse**, που μας επιτρέπουν να ζητούμε εκτέλεση εντολών υπό συνθήκη.
- Εντολή επανάληψης **while**, που μας επιτρέπει να ζητούμε την επαναλαμβανόμενη εκτέλεση των ίδιων εντολών υπό συνθήκη.
- Εντολή επανάληψης **for**, που τη χρησιμοποιήσαμε για μετρούμενες επαναλήψεις.

Όπως θα μάθουμε στη συνέχεια οι τρεις πρώτες είναι περιπτώσεις μιας εντολής, της *εντολής-παράστασης*.

Είδαμε ακόμη και διάφορες *πράξεις*, μαζί με τους αντίστοιχους *τελεστές*: τις αριθμητικές πράξεις (+, -, *, /, %), τις συγκρίσεις (==, !=, <, <=, >, >=), τις λογικές πράξεις (&&, ||, !) και τις *ενικές* πράξεις:

- τα πρόσημα "+" και "-",
- "&", που μας δίνει τη διεύθυνση της μνήμης όπου βρίσκεται μια μεταβλητή,
- "*", που μας δίνει την τιμή της μεταβλητής που βρίσκεται σε κάποια διεύθυνση της μνήμης,
- "sizeof", που μας δίνει το "μέγεθος" μιας μεταβλητής σε ψηφιολέξεις και
- "typeid", που μας δίνει πληροφορίες για τον τύπο κάποιας παράστασης.

Τώρα, για την υλοποίηση των νέων εννοιών και τεχνικών που θα μάθουμε, θα χρειαστούμε και άλλες εντολές όπως και άλλες δυνατότητες εντολών που ήδη ξέρουμε.

Είναι σίγουρο ότι μπορείς να κάνεις την προγραμματιστική σου δουλειά χωρίς τα περισσότερα από τα «νέα» στοιχεία, αλλά αυτά μπορεί να κάνουν το πρόγραμμά σου πιο σίγουρο και πιο αποδοτικό.

Μερικά από τα νέα στοιχεία είναι επικίνδυνα και πρέπει να αποφεύγεις τη χρήση τους. Αυτά τα τονίζουμε. Άλλα μπορεί να είναι ενοχλητικά χωρίς να είναι επικίνδυνα. Το πώς και πότε θα τα χρησιμοποιείς είναι ζήτημα πείρας και ωριμότητας· θα βρεις την άκρη μόνος/η σου με τον καιρό.

## 11.1 Εκτελέσιμες Δηλώσεις

Όταν κάποιος μαθαίνει προγραμματισμό, ένα από τα πρώτα πράγματα που του διδάσκουν είναι να δηλώνει τις μεταβλητές του στην αρχή του προγράμματος ή του υποπρογράμματος. Αυτό άλλωστε απαιτούν και οι περισσότερες γλώσσες προγραμματισμού. Σε πολλές γλώσσες όλες οι μεταβλητές μιας συνάρτησης δημιουργούνται με την ενεργοποίηση της συνάρτησης και καταστρέφονται με τον τερματισμό της ενεργοποίησης.

Μέχρι τώρα και εμείς σου δίνουμε την ίδια συμβουλή, αλλά στη C++ τα πράγματα είναι διαφορετικά. Για να καταλάβεις τι θα αλλάξουμε και γιατί θα το αλλάξουμε δες το παρακάτω προγραμματάκι:

```
int main()
{
   MyType v1;

   cout << "going into block" << endl;
   { // <--- block start
      MyType v2( 5 );

      cout << " chkpoint 1" << endl;

      MyType v3( 15 );

      cout << " chkpoint 2" << endl;
   } // <--- block end
   cout << "coming out of block" << endl;
}
```

Ο τύπος *MyType* είναι σαν τον **`int`** με τη διαφορά ότι κάθε φορά που δημιουργείται ή καταστρέφεται μια μεταβλητή αυτού του τύπου το αναγγέλει[1]. Δες τι μας δίνει η εκτέλεση του προγράμματος:

```
creating a MyType variable. Initializing with 0
going into block
creating a MyType variable. Initializing with 5
 chkpoint 1
creating a MyType variable. Initializing with 15
 chkpoint 2
destroying a MyType variable having value 15
destroying a MyType variable having value 5
coming out of block
destroying a MyType variable having value 0
```

Αυτό που επιβεβαιώνεται εδώ, είναι το εξής:

- ***Μια μεταβλητή δημιουργείται όταν η εκτέλεση φτάσει στη δήλωσή της και καταστρέφεται όταν η εκτέλεση βγει από τη σύνθετη εντολή (block) στην οποίαν περιέχεται η δήλωση.***

Αργότερα θα τα ξαναπούμε πληρέστερα και ακριβέστερα.

Σκέψου τώρα το εξής: σε κάθε πρόγραμμα παραχωρούνται τρεις περιοχές μνήμης,

- η **στατική**, όπου υλοποιούνται οι καθολικές μεταβλητές (και μερικές άλλες που θα δούμε στη συνέχεια),
- η ***αυτόματη*** (automatic) ή μνήμη **στοίβας** (stack), όπου υλοποιούνται οι τοπικές μεταβλητές των σύνθετων εντολών και των συναρτήσεων και
- η **δυναμική**[2] (dynamic) μνήμη που θα δούμε στη συνέχεια.

Από αυτές η πιο περιορισμένη είναι η μνήμη στοίβας και θα πρέπει να τη χρησιμοποιούμε με φειδώ. Αλλά, αυτή ακριβώς είναι η μνήμη για την οποίαν συζητούμε στο παράδειγμά μας. Με βάση τα παραπάνω μπορούμε να δώσουμε νέον κανόνα για τη δήλωση των μεταβλητών:

- ***Δήλωνε τη μεταβλητή ακριβώς εκεί που τη χρειάζεσαι.***

Μερικοί προχωρούν ακόμη περισσότερο: *μη δηλώνεις μια μεταβλητή αν δεν ξέρεις τι αρχική τιμή να της δώσεις*. Ε, όχι και έτσι...

---

[1] Αργότερα θα μάθουμε πώς να υλοποιήσουμε έναν τέτοιον τύπο.

[2] Θα τη δεις και ως μνήμη **σωρού** (heap).

Αν ακολουθείς αυτόν τον κανόνα συνήθως θα συμμορφώνεσαι και με την (ακριβέστερη) σύσταση του (CERT 2009) που λέει:[3]

- ***Χρησιμοποίησε την ελάχιστη δυνατή εμβέλεια για όλες τις μεταβλητές και τις μεθόδους.***

## 11.2 Περιορισμός Τύπου

Πριν προχωρήσουμε να δούμε άλλες εντολές ας πούμε λίγα πράγματα για τον **περιορισμό τύπου** (type qualification).

Ήδη στην §2.4 είδαμε για πρώτη φορά τον ορισμό σταθεράς με όνομα:

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

Λέμε ότι το "`const`" είναι ένας **περιοριστής τύπου** (type qualifier)· εδώ *περιορίζει* τον τύπο `double`.

Η C++ μας δίνει άλλον έναν περιοριστή τύπου, τον "`volatile`" (ευμετάβλητος). Αν δηλώσεις:

```
volatile int flags;
```

πληροφορείς τον μεταγλωττιστή ότι η *flags* είναι μια μεταβλητή που *η τιμή της αλλάζει συχνά από συμβάντα εκτός προγράμματος* (π.χ. από κάτι που ήλθε στη σειριακή πόρτα). Επομενως, ο μεταγλωττιστής θα πρέπει να μην παίρνει την τιμή της από κάποιον καταχωριτή (για ταχύτερη εκτέλεση) αλλά από τη φυσική της διεύθυνση.

Οι τύποι με περιορισμό "`const`" ή "`volatile`" αναφέρονται συνοπτικώς και ως **τύποι με περιορισμό cv** (cv-qualified).

## 11.3 Όχι «Εντολή Εκχώρησης» αλλά «Πράξη Εκχώρησης»

Ναι, εντολή εκχώρησης δεν υπάρχει! Αυτό που υπάρχει είναι η *πράξη της εκχώρησης*: Αν έχουμε δηλώσει:

```
T v;
```

τότε με την `v` = *Π* γίνονται τα εξής:

- Υπολογίζεται η τιμή της παράστασης και μετατρέπεται στον τύπο της μεταβλητής `static_type<T>(`*Π*`)`.
- Η τιμή `static_type<T>(`*Π*`)` φυλάγεται ως τιμή της μεταβλητής.
- *Αποτέλεσμα της πράξης* είναι η `v`.[4]

Κι αυτά που λέγαμε μέχρι τώρα; Δεν είναι λάθος, διότι: όπως πιθανότατα έχεις ήδη καταλάβει (μάλλον από κάτι που θα έγραψες κατά λάθος σε κάποιο πρόγραμμα)

- ***Μια παράσταση είναι εντολή για τη C++.***

Εμείς θα *ονομάζουμε εντολή εκχώρησης μια εντολή της μορφής "*`v` = *Π", όπου η Π δεν θα περιέχει πράξεις εκχώρησης.*

Όπως καταλαβαίνεις, έχεις δικαίωμα να γράψεις:

```
w = v = u = 0;
```

Η C++ θα την εκτελέσει ως εξής:

```
w = (v = (u = 0));
```

[3] Σύσταση *DCL07*: "Use as minimal scope as possible for all variables and methods". «*Μέθοδοι*»!; Θα μάθεις αργότερα τι είναι αυτό...

[4] Πρόσεξε: «η `v`» και όχι «η τιμή της `v`». Θα καταλάβεις αργότερα...

Δηλαδή: η *u* θα πάρει τιμή 0, ενώ το αποτέλεσμα της πράξης, το 0, εκχωρείται στη *v*. Τέλος, το αποτέλεσμα αυτής της εκχώρησης, πάλι 0, εκχωρείται στη *w*. Άρα, με την εντολή αυτή, με τη μια ή την άλλη μορφή, εκχωρούμε την ίδια τιμή (στην περίπτωσή μας: 0) σε περισσότερες από μια μεταβλητές (στην περίπτωσή μας: τρεις). Θα την ονομάσουμε *εντολή πολλαπλής εκχώρησης*.

Ακόμη, έχεις δικαίωμα να γράψεις:

```
   cout << (v = 1.5) << endl;
```

Εδώ, η τιμή `static_type<T>(1.5)` θα εκχωρηθεί στη *v* και στη συνέχεια το αποτέλεσμα της πράξης (τιμή της *v*) θα εκτυπωθεί. Καλύτερα όμως να αποφεύγεις τέτοια χρήση. Δες το παρακάτω παράδειγμα (Borland C++) για να καταλάβεις:

```
   int v( 0 );
   cout << v << "    " << (v = 1.5) << endl;
```

Από εδώ θα περίμενες να πάρεις:

```
0   1
```

αλλά παίρνεις:

```
1   1
```

Αυτό γίνεται διότι τα ορίσματα της "`cout <<`" υπολογίζονται *από το τέλος προς την αρχή*. Έτσι, πρώτα υπολογίζεται η "`v = 1.5`" από όπου η *v* παίρνει τιμή 1.

Σε μια εκχώρηση, αριστερά του "=" δεν είναι απαραίτητο να υπάρχει μεταβλητή· ήδη είδαμε ότι μπορεί να υπάρχει και στοιχείο πίνακα. Γενικώς: *αριστερά του "=" μπορεί να υπάρχει μια* **τιμή-l** (*l*value)[5], *δηλαδή, παράσταση που καθορίζει μια περιοχή της μνήμης*. Φυσικά και το όνομα μιας σταθεράς καθορίζει μια περιοχή της μνήμης, αλλά απαγορεύεται να εμφανιστεί στο αριστερό μέρος μιας εκχώρησης. Στις εκχωρήσεις θέλουμε μια **τροποποιήσιμη** (modifiable) τιμή-*l*.

Τώρα όμως, στον υπολογισμό της εκχώρησης, έχουμε *άλλο ένα βήμα*: τον υπολογισμό της τιμής-*l*, δηλαδή: *τον καθορισμό της περιοχής της μνήμης όπου θα αποθηκευτεί η τιμή της παράστασης*. Και όταν η εκχώρηση τιμής γίνεται σε μια μεταβλητή, δεν έχουμε πρόβλημα· όταν όμως γίνεται σε στοιχείο πίνακα τότε...

**Παράδειγμα** ➢

Έστω ότι έχουμε:

```
   int k;
   double a[5];
// . . .
   a[0] = 5;  a[1] = 10.5;
// . . .
   k = 0;
   a[k] = a[k] + (k = k+1);
   cout << a[0] << "  " << a[1] << endl;
```

Από τον gcc (Dev-C++) θα πάρουμε:

```
6  10.5
```

που βγαίνει ως εξής: εδώ, πρώτα υπολογίζεται η παράσταση "`a[k]`" σε "`a[0]`" *και για το αριστερό και για το δεξιό μέλος*. Στη συνέχεια υπολογίζεται ο όρος: "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης. Τελικώς, αυτό που μένει να υπολογισθεί είναι το: "`a[0] = (a[0] + 1)`" και έτσι αυξάνεται κατά 1 η τιμή του a[0].

Αν το πρόγραμμά μας μεταγλωττισθεί στη MS Visual C++ v5, θα πάρουμε:

```
5  6
```

[5] Το "*l*" για ιστορικούς λόγους, από το *l*eft part (αριστερό μέρος). Όπως πιθανότατα μαντεύεις, υπάρχει και **τιμή-r** (*r*value) που εμφανίζεται στο δεξιό (*r*ight) μέρος του τελεστή της εκχώρησης.

που βγαίνει ως εξής: Πρώτα υπολογίζεται η παράσταση "`a[k] + (k = k+1)`" αλλά η VC++ προτάσει τον υπολογισμό του "`a[k]`" που είναι το a[0]. Στη συνέχεια υπολογίζεται η "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης "`a[k] + (k = k+1)`" ως "`a[0] + 1`". Τέλος έρχεται η ώρα υπολογισμού της τιμής-*l*, που είναι η a[k]. Αλλά τώρα η *k* έχει τιμή 1 και έτσι τελικώς η εντολή υπολογίζεται ως: "`a[1] = a[0]+1`". Έτσι υπολογίζεται η τιμή της παράστασης: Στη συνέχεια αυξάνεται κατά 1 η τιμή του a[1].

Αν το πρόγραμμά μας μεταγλωττισθεί στη Borland C++ v5, θα πάρουμε:

```
5  11.5
```

που βγαίνει ως εξής: Και εδώ, πρώτα υπολογίζεται η παράσταση "`a[k] + (k = k+1)`" αλλά τώρα πρώτα υπολογίζεται ο όρος: "`k = k+1`"· η τιμή της *k* αυξάνεται κατά 1 –γίνεται 1– και αυτή είναι η τιμή της παράστασης. Έτσι, αυτό που μένει να υπολογισθεί είναι το: "`a[1] = (a[1] + 1)`" και έτσι αυξάνεται κατά 1 η τιμή του a[1].

Οι BC++v5 και gcc έχουν ως κοινό σημείο το ότι η τιμή της "`a[k]`" υπολογίζεται *μια φορά μόνον* είτε ως "`a[1]`" (BC++) είτε ως "`a[0]`" (gcc).

Στο παράδειγμα φαίνεται παραστατικότατα ότι *έχουμε υπολογισμό της τιμής-l*. Λόγω διαφορών στη σειρά εκτέλεσης πράξεων μπορεί να πάρουμε διαφορετικά αποτελέσματα.

Όπως λέγαμε και στην §2.10, η C++ μας δίνει μερικές «συντομογραφίες» πολύ συνηθισμένων εκχωρήσεων. Π.χ., αν τα *x*, *b* είναι οποιουδήποτε αριθμητικού τύπου, αντί για:

```
x = x + b;
```

μπορούμε να γράψουμε:

```
x += b;
```

Παρομοίως και για τις άλλες πράξεις:

`x +=` *Π*`;` είναι ισοδύναμη με: `x = x +` *Π*`;`
`x -=` *Π*`;` είναι ισοδύναμη με: `x = x -` *Π*`;`
`x *=` *Π*`;` είναι ισοδύναμη με: `x = x *` *Π*`;`
`x /=` *Π*`;` είναι ισοδύναμη με: `x = x /` *Π*`;`
`x %=` *Π*`;` είναι ισοδύναμη με: `x = x %` *Π*`;`

Για τις `/=` και `%=` η παράσταση *Π* θα πρέπει να παίρνει τιμή μη μηδενική. Για την τελευταία: η *x* και η τιμή της *Π* θα πρέπει να είναι ακέραιου τύπου.

Ειδικώς για την περίπτωση που θέλουμε να αυξήσουμε ή να μειώσουμε την τιμή μιας μεταβλητής κατά 1, υπάρχουν και άλλες συντομογραφίες:

`++x;` που είναι ισοδύναμη με: `x = x + 1;` ή `x += 1;`
`--x;` που είναι ισοδύναμη με: `x = x - 1;` ή `x -= 1;`

όπως και οι παραλλαγές τους: οι `x++` και `x--` αυξάνουν / μειώνουν την τιμή της *x* κατά 1, *αλλά το αποτέλεσμα της πράξης είναι η αρχική τιμή της x*. Δες το παρακάτω παράδειγμα:

```
 0:   #include <iostream>
 1:   using namespace std;
 2:   int main()
 3:   {
 4:      int x, y; // οποιουδήποτε αριθμητικού τύπου
 5:
 6:      x = 5;  y = ++x;   cout << x << "   " << y << endl;
 7:      x = 5;  y = --x;   cout << x << "   " << y << endl;
 8:      x = 5;  y = x++;   cout << x << "   " << y << endl;
 9:      x = 5;  y = x--;   cout << x << "   " << y << endl;
10:   }
```

Αποτέλεσμα:

```
6   6
4   4
6   5
4   5
```

Όπως βλέπεις, στις γρ. 8-9, η *y* παίρνει την τιμή που είχε η *x* (5) *πριν* αυτή αλλάξει.

Και τι γίνεται με τις συντομογραφίες όταν πρέπει να υπολογιστεί το αριστερό μέρος (τιμή-*l*); Η C++ μας λέει ότι:

- `x θ= y` ***σημαίνει*** `x = x θ y` ***με τη διαφορά ότι η x υπολογίζεται μια φορά μόνον.***

Αν λοιπόν, στο παράδειγμα που είδαμε πιο πριν, γράψουμε:

```
   k = 0;
   a[k] += (k = k+1);
   cout << a[0] << "  " << a[1] << endl;
```

θα πάρουμε είτε (gcc):

```
6  10.5
```

είτε (Borland C++ v5)[6]:

```
5  11.5
```

Στο *Μέρος Α*, θέλοντας να εστιάσουμε την προσοχή μας σε πολύ βασικές και κρίσιμες έννοιες, *αποφύγαμε* –με πολλή προσοχή– τη χρήση αυτών των συντομογραφιών. Από εδώ και πέρα –και πάλι με πολλή προσοχή– *θα τις χρησιμοποιούμε*. Σχετικώς, διάβασε την επόμενη παράγραφο και το Πλ. 11.1.

### 11.3.1 * Ο "++" για τον Τύπο bool

Όπως είπαμε, οι "++" και "--" μπορούν να δράσουν σε μεταβλητή (γενικώς: τιμή-*l*) οποιουδήποτε αριθμητικού τύπου. Ειδικώς ο πρώτος μπορεί να δράσει και σε μεταβλητές τύπου `bool`. Αν η *b* είναι μια τέτοια μεταβλητή, μετά την εκτέλεση της "`++b`" (ή της "`b++`"), η τιμή της είναι `true`.

Δεν θα το χρησιμοποιήσουμε ποτέ· το λέμε απλώς για να το ξέρεις...[7]

## 11.4 Οι Εκχωρήσεις και οι Συνθήκες Επαλήθευσης

Ας πούμε ότι έχουμε:

```
// x == 5 && k == 0 && q > 0
   y = (x += q) + ++k;
```

Τι θα ισχύει μετά την εντολή εκχώρησης; Προφανώς όχι αυτά που λέγαμε στο Κεφ. 2. Παρόμοια προβλήματα θα βρεις και στην εφαρμογή των συμπερασματικών κανόνων των **if** και **while** (και αυτών που θα δεις στη συνέχεια) στην περίπτωση που οι συνθήκες περιέχουν εκχωρήσεις.

Βέβαια, ο απαιτητικός αναγνώστης θα πει: «Αν γράψω τα παραπάνω ως εξής:

```
// x == 5 && k == 0 && q > 0
   x += q;  ++k;
   y = x + (k-1);
```

μπορώ να βγάλω συμπέρασμα, έτσι δεν είναι; Γιατί να μην αλλάξουμε τους κανόνες ώστε να καλύπτουμε όλες τις περιπτώσεις;» Απάντηση: *Καλύτερα να κρατήσουμε τους κανόνες απλούς και να γράφουμε όπως στη δεύτερη περίπτωση και όχι όπως στην πρώτη.* Θα τηρούμε αυτήν την αρχή σε ό,τι γράφουμε από εδώ και πέρα, όπως το τηρήσαμε και μέχρι τώρα. Δες το Πλ. 11.1.

**Παρατήρηση:**▶

Όπως θα δεις στη συνέχεια, θα χρησιμοποιούμε τον προθεματικό τελεστή (`++k`, `--k`) και όχι τον μεταθεματικό (`k++`, `k--`). Ο λόγος είναι ο εξής:

---

[6] Δυστυχώς, η MS VC++ θα δώσει και πάλι: "`5  6`", υπολογίζοντας το a[*k*] μια φορά ως a[0] και μια ως a[1].

[7] Στο C++11 η χρήση του "`++`" για μεταβλητές τύπου `bool` αποθαρρύνεται.

**Πλαίσιο 11.1**

**Συντομογραφίες: Πότε Ναι και Πότε Όχι**

1) **ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ** την πολλαπλή εκχώρηση `x = y = z =` *Π*, με την προϋπόθεση, φυσικά, ότι α) η Π δεν έχει εκχωρήσεις και β) δεν υπάρχουν μπερδέματα από τις μετατροπές τύπου.

2) **ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ** εντολές σαν τις:

```
++k;
j++;
p += (q+4);
```

όπου μεταβάλλεται η τιμή μιας μεταβλητής μόνον.

**ΔΕΝ ΘΑ ΧΡΗΣΙΜΟΠΟΙΟΥΜΕ** εντολές σαν τις:

```
y = ++p + (x = q/2);
x = (y += p+1);
```

όπου μεταβάλλονται τιμές δύο ή περισσοτέρων μεταβλητών. Φυσικά, για εντολές σαν την:

```
a[k] += (k = k+1);
```

`ούτε λόγος!`

- η “`++k`” είναι ισοδύναμη με τις “`k += 1`” και “`k = k + 1`” και
- η “`--k`” είναι ισοδύναμη με τις “`k -= 1`” και “`k = k - 1`”

σε κάθε χρήση, «νόμιμη» ή «παράνομη».◄

## 11.5 Παράσταση Υπό Συνθήκη

Στο Κεφ. 5 είδαμε για πρώτη φορά το εξής πρόβλημα: έχουμε δύο πραγματικούς αριθμούς, *x*, *y*, και θέλουμε να εκχωρήσουμε την τιμή του μεγαλύτερου (όχι μικρότερου) από αυτούς σε έναν τρίτο, ας τον πούμε *maxxy*. Μια από τις λύσεις που δώσαμε ήταν η εξής:

```
if ( x > y )  maxxy = x;
        else  maxxy = y;
```

Η C++ σου επιτρέπει να λύσεις το πρόβλημα με μια εντολή εκχώρησης, με χρήση του τελεστή “?:”

```
maxxy = (x > y) ? x : y;
```

Η παράσταση δεξιά του “=” είναι μια **παράσταση υπό συνθήκη** (conditional expression).

**Παράδειγμα** ✎

Αντί για την παρακάτω συνάρτηση –που μας δίνει το πρόσημο του *x*:

**Πλαίσιο 11.2**

**Παράσταση Υπό Συνθήκη**

Ο τελεστής ? είναι *τριαδικός*, δηλαδή δέχεται τρία ορίσματα:

*συνθήκη*, "?", *παράσταση*-1, ":", *παράσταση*-2

όπου: η συνθήκη είναι λογική παράσταση και οι δύο παραστάσεις δίνουν αποτέλεσμα ίδιου τύπου.

Ο υπολογισμός γίνεται ως εξής: Πρώτα υπολογίζεται η συνθήκη. Αν ισχύει τότε υπολογίζεται η *παράσταση*-1 και η τιμή της είναι τιμή όλης της παράστασης αλλιώς (αν η συνθήκη δεν ισχύει) υπολογίζεται η *παράσταση*-2 και η τιμή της είναι τιμή όλης της παράστασης. Φυσικά, οι *παράσταση*-1 και *παράσταση*-2 μπορεί να είναι, και αυτές, παραστάσεις υπό συνθήκη.

```
int xSign( int x )
{
   int fv( 0 );
   if ( x < 0 )  fv = -1;
   else if ( x > 0 )  fv = 1;
   return fv;
} // xSign
```

μπορείς να γράψεις:

```
int xSign( int x )
{
   return ( x < 0 ) ? -1 : ( x == 0 ) ? 0 : 1;
} // sign
```

Αν το μόνο που μας ενδιαφέρει είναι το πρόσημο της *x* και δεν θέλουμε να γράψουμε συνάρτηση, μπορούμε να γράψουμε:

```
   cout << ( (x < 0) ? -1 : (x == 0) ? 0 : 1 ) << endl;
```

ꕥꕥꕥ

Οι παρενθέσεις στις συνθήκες δεν είναι απαραίτητες.

## 11.6 Η Εντολή "for"

Στο Μέρος Α (§6.4) είπαμε ότι «*Η **for** της C++ έχει πολύ περισσότερες δυνατότητες. Προς το παρόν είδαμε –και θα χρησιμοποιούμε– μόνον αυτές που μας χρειάζονται.*» Τώρα ήρθε η ώρα να μάθουμε και τις υπόλοιπες.

Η μορφή της εντολής **for** που χρησιμοποιήσαμε μέχρι τώρα ήταν για την μετρούμενη επανάληψη και μόνον:

```
for ( v = a; v <= t; v = v + b )  E;
```

ή

```
for ( v = a; v >= t; v = v - b )  E;
```

Αν $b > 0$,

- στην πρώτη περίπτωση η *v* παίρνει τιμές *a*, *a*+*b*, ... *a*+*nb* όπου η *n* έχει τιμή τέτοια ώστε: $a + nb \le t < a + (n+1)b$,
- στη δεύτερη περίπτωση η *v* παίρνει τιμές *a*, *a*-*b*, ... *a*-*nb* όπου η *n* έχει τιμή τέτοια ώστε: $a - (n+1)b < t \le a - nb$.

Και στις δύο περιπτώσεις, για κάθε τιμή της *v* εκτελείται η *E*.

Αλλά η for έχει μια πολύ γενικότερη μορφή και πολύ περισσότερες δυνατότητες. Το συντακτικό της είναι:

"`for`", "`(`", *αρχική εντολή της for*, [ *συνθήκη*, ] "`;`", [ *παράσταση*, ] "`)`", *εντολή*"`;`"

όπου:

*αρχική εντολή της for* = { *εντολή-παράσταση* | *απλή δήλωση* } "`;`"

Η *συνθήκη* μπορεί να μην υπάρχει· στην περίπτωση αυτή ότι έχει τιμή `true`. Η *παράσταση* μπορεί επίσης να μην υπάρχει· στην περίπτωση αυτή η φροντίδα για το τέλος της επανάληψης αφήνεται στην επαναλαμβανόμενη *εντολή*. Η *εντολή* και η *αρχική εντολή της for*, όπως θα δούμε στη συνέχεια, μπορεί να είναι κενές.

Ας ξεκινήσουμε με ένα παράδειγμα όπου η *αρχική εντολή της for* είναι απλή δήλωση.

**Παράδειγμα 1**

Έστω ότι έχουμε δύο μονοδιάστατους πίνακες και μια μεταβλητή

```
   double a[50], b[80], m( 0 );
```

και θέλουμε να αντιγράψουμε τις τιμές των στοιχείων a[0]..a[9] στα στοιχεία b[10]..b[19] και να υπολογίσουμε, στην *m*, τη μέση τιμή των τιμών που αντιγράφουμε. Γράφουμε:

```
   for ( int k(0), j(10);  k < 10;  ++k, ++j )
   {
```

```
      b[j] = a[k];  m += a[k];
   } // for
   m /= 10;
```

Ας ξεκινήσουμε από την *αρχική εντολή της **for***. Στην περίπτωσή μας είναι δήλωση δύο μεταβλητών:

```
int k(0), j(10);
```

με την οποία δίνουμε και αρχικές τιμές. Τώρα πρόσεξε το εξής: *Οι k και j είναι γνωστές μόνον μέσα στη **for**· δεν μπορείς να τις χρησιμοποιήσεις έξω από αυτήν.*

Η συνθήκη είναι από αυτές που ξέρουμε: "`k <= 9`".

Ποια είναι η παράσταση; Η "`++k, ++j`", δηλαδή μια *ακολουθία παραστάσεων* (δες παρακάτω την §11.11). Τι τιμή θα παίρνει κάθε φορά; Δεν μας νοιάζει! Αυτό που μας ενδιαφέρει είναι ότι κάθε φορά αυξάνει τις τιμές των *k* και *j*.

> *Ακολουθία Παραστάσεων*: Παράσταση που αποτελείται από άλλες παραστάσεις που χωρίζονται ανά δύο με ένα κόμμα. Τιμή της παράστασης είναι αυτή της τελευταίας από τις παραστάσεις που την απαρτίζουν. Για περισσότερα δες παρακάτω την §11.11.

Βλέπουμε λοιπόν ότι:

- ***Μπορούμε να δηλώνουμε, σε μια for, τοπικές μεταβλητές που α) είναι γνωστές μόνον μέσα στη for και β) ζουν όσο διαρκεί η εκτέλεση της for.***

Οι τοπικές μεταβλητές της **for** δεν διαφέρουν σε τίποτε από τις τοπικές μεταβλητές μιας συνάρτησης.

Όταν η *αρχική εντολή της **for*** είναι *εντολή-παράσταση* είναι συχνά μια ακολουθία εντολών που δίνει αρχικές τιμές σε ορισμένες μεταβλητές.

**Παράδειγμα 2** ✎

Ξαναγράφουμε το πρόγραμμα Μέση Τιμή 1 της §5.1:

```
 0:  #include <iostream>
 1:  // πρόγραμμα: Μέση Τιμή 1
 2:  using namespace std;
 3:
 4:  int main()
 5:  {
 6:     const int n( 10 );
 7:
 8:     int m;       // Μετρητής των επαναλήψεων
 9:     double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
10:     double sum;  // Το μερικό (τρέχον) άθροισμα.
11:                  // Στο τέλος έχει το ολικό άθροισμα.
12:     double avrg;// Μέση Αριθμητική Τιμή των x (<x>)
13:
14:     for ( sum = 0, m = 1; m <= n;  m = m + 1 )
15:     {
16:        cout << "Δώσε έναν αριθμό: ";  cin >> x;   // x == tm
17:        sum = sum + x;     // (x == tN) && (sum = Σ(j:1..m)tj)
18:     } // for
19:     avrg = sum / n;
20:     cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = "
21:          << avrg << endl;
22:  }
```

Τι κάναμε εδώ; Πήραμε τις δύο εντολές προετοιμασίας της **while**, τις:

```
sum = 0;  m = 1;
```

και τις βάλαμε για *αρχική εντολή της **for*** (γρ. 14). Πρόσεξε ότι αντικαταστήσαμε το '`;`' που υπήρχε μεταξύ τους με ένα κόμμα ('`,`') για να σχηματίσουμε μια *ακολουθία παραστάσεων*.

Ακόμη πήραμε τη συνθήκη της **while** και τη βάλαμε ως συνθήκη της **for**.

Τέλος, πήραμε την τελευταία εντολή της περιοχής επανάληψης, την `m = m + 1`, και τη βάλαμε στη θέση της παράστασης. Θα μπορούσαμε να αφήσουμε την εντολή στη θέση της και να μην βάλουμε παράσταση:

```
for ( sum = 0, m = 1; m <= n; )
{
   cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
   sum = sum + x;           // (x == tm) && (sum = Σ(j:1..m)tj)
   m = m + 1;
} // for
```

Αλλά και αντιστρόφως: θα μπορούσαμε να μεταφέρουμε στην παράσταση, εκτός από την `m = m + 1`, και την `sum = sum + x`:

```
for ( sum = 0, m = 1; m <= n; sum = sum + x, m = m + 1 )
{
   cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
} // for
```

Ε, αφού είναι έτσι, γιατί να μην τις πάμε όλες; Και δεν τις παμε…

```
for ( sum = 0, m = 1;
      m <= n;
      cout << "Δώσε έναν αριθμό: ", cin >> x,
      sum = sum + x, m = m + 1 );
```

ή, με τις συντομογραφίες:

```
for ( sum = 0, m = 1;
      m <= n;
      cout << "Δώσε έναν αριθμό: ", cin >> x,
      sum += x, ++m );
```

Και έτσι μείναμε χωρίς επαναλαμβανόμενη εντολή!

Όλες αυτές οι **for** που βλέπεις είναι ισοδύναμες με την αρχική **while** μαζί με τις δύο εντολές προετοιμασίας.

Πρόσεξε και κάτι άλλο: *η sum δεν μπορεί να δηλωθεί μέσα στη **for** διότι θέλουμε την τιμή της μετά το τέλος εκτέλεσης της **for***. Η *m* όμως μας χρειάζεται μόνο μέσα στη **for**. Μήπως θα μπορούσαμε να γράψουμε:

```
for ( sum = 0, int m( 1 ); m <= n; . . . );
```

Όχι! Είπαμε ότι η αρχική εντολή της **for** μπορεί να είναι *εντολή-παράσταση* ή *απλή δήλωση*. Δεν μπορεί να είναι λίγο από το ένα και λίγο από το άλλο.

Αν εγκαταλείψουμε την αρχή «*όλες οι μεταβλητές δηλώνονται στην αρχή της συνάρτησης*» και τηρήσουμε την αρχή «*κάθε μεταβλητή δηλώνεται εκεί που μας χρειάζεται*» το πρόγραμμά μας γράφεται:

```
// πρόγραμμα: Μέση Τιμή 1 με for
#include <iostream>
using namespace std;
int main()
{
   const int n( 10 );

   double sum( 0 );  // Το μερικό (τρέχον) άθροισμα.
                     // Στο τέλος έχει το ολικό άθροισμα.
   for ( int m(1); m <= n; ++m )
   {
      double x;      // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
      cout << "Δώσε έναν αριθμό: ";  cin >> x;           // x = tm
      sum += x;                 // (x == tm) && (sum = Σ(j:1..m)tj)
   } // while

   double avrg( sum / n );      // Μέση Αριθμητική Τιμή των x (<x>)
   cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
} // main
```

Γενικώς, η:

```
Ep1; Ep2; ... Epn;
while ( S )
{  Er1; Er2; ... Erk; Er(k+1); Er(k+2); ... Er(k+m);  }
```

μπορεί να γραφεί ως:

```
for ( Ep1, Ep2, ... Epn; S; Er(k+1), Er(k+2), ... Er(k+m) )
{  Er1; Er2; ... Erk; }
```

Αλλά προσοχή! *Οι εντολές που μεταφέρεις είτε στην αρχική εντολή της **for** είτε στην παράσταση θα πρέπει να είναι εντολές-παραστάσεις* ώστε να δημιουργούνται ακολουθίες παραστάσεων. Δηλαδή δεν μπορείς να μεταφέρεις εκεί εντολές **if**, **ifelse**, **while** ή **for**, από αυτές που ξέρουμε μέχρι τώρα[8]. Έτσι, αν θελήσεις να γράψεις με for την επανάληψη του Μέση Τιμή 4 (§5.1.2) θα γράψεις:

```
   for ( sum = 0,  n = 0,  selSum = 0,  selN = 0,
         cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x;
         x != FROUROS;
         cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x )
   {
      n = n + 1;                  // Αυτά γίνονται για
      sum = sum + x;              // όλους τους αριθμούς
      if (0 < x && x <= 10)       // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;      // Αυτά γίνονται μόνο για
         selN = selN + 1;          // τους επιλεγόμενους αριθμούς
      } // if
   } // for
```

και η περιοχή της επανάληψης δεν μπορεί να μικρύνει περισσότερο.

Το ότι μπορείς να βγάζεις τις εντολές από την περιοχή επανάληψης και να τα βάζεις στην επικεφαλίδα της **for** δεν σημαίνει ότι πρέπει να το κάνεις οπωσδήποτε. Δες στο προηγούμενο παράδειγμα τη **for** χωρίς περιοχή επανάληψης: αυτό φτάνει και περισσεύει.

Αν θέλεις να αποδείξεις την ορθότητα μιας **for**, σκέψου την ισοδύναμη **while**. Αν, ας πούμε, έχεις:

```
// P
for (Ep1, Ep2, ... Epn; S; Er(k+1), Er(k+2), ... Er(k+m))
{  Er1; Er2; ... Erk; }
// Q
```

θα πρέπει να αποδείξεις ($I$ η αναλλοίωτη):

1.
```
// P
   Ep1; Ep2; ... Epn;
// I
```

2.
```
// I && S
   Er1; Er2; ... Erk; Er(k+1), Er(k+2), ... Er(k+m);
// I
```

3. $(I \;\&\&\; !S) \Rightarrow Q$

## 11.7 Η Εντολή "do-while"

Εκτός από τη **while** και τη **for**, η C++ έχει και μια άλλη εντολή επανάληψης, τη **do-while**. Δες το Πλ. 11.3 και το Σχ. 11-1.

Η σημαντική διαφορά της από τη **while** είναι ότι

- στη **while** πρώτα ελέγχεται η συνθήκη και η εντολή εκτελείται μόνον αν ισχύει, ενώ
- στη **do-while** πρώτα εκτελείται η εντολή και μετά ελέγχεται η συνθήκη.

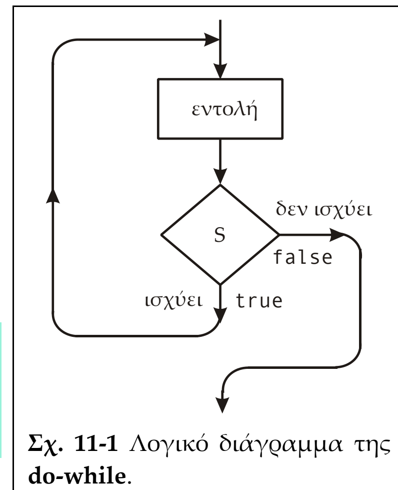
Έτσι:

[8] Και είναι εντολές-παραστάσεις οι εντολές εισόδου και εξόδου; Ναι, είναι! Διάβασε παρακάτω...

- ***Ενώ στη while η επαναλαμβανόμενη εντολή μπορεί να μην εκτελεσθεί ούτε μία φορά, στη do-while θα εκτελεσθεί μία φορά τουλάχιστον.***

Είναι σαφές ότι μπορούμε να ζήσουμε και χωρίς τη **do-while**. Ας δούμε όμως ένα παράδειγμα που είναι προτιμότερη από τη **while**. Έστω ότι θέλουμε να διαβάσουμε από το πληκτρολόγιο έναν *θετικό* αριθμό. Να τι θα κάναμε με τη **while**:

```
cout << " Δώσε θετικό: ";  cin >> x;
while ( x <= 0 )
{
   cout << " Δώσε θετικό: ";  cin >> x;
} // while
```




**Σχ. 11-1** Λογικό διάγραμμα της **do-while**.

Με τη **do-while** τα πράγματα είναι απλούστερα:

```
do {
   cout << " Δώσε θετικό: ";  cin >> x;
}  while ( x <= 0 );
```

Ο συμπερασματικός κανόνας της **do-while** είναι λίγο πιο πολύπλοκος από αυτόν της **while**. Και εδώ βέβαια υπάρχει *αναλλοίωτη* που ισχύει πριν από την εντολή και μετά από αυτήν. Φυσικά, μετά την εντολή δεν ισχύει η συνθήκη συνέχισης. Έστω ότι έχουμε:

```
// I
do E while ( S );
// I && !S
```

Έστω τώρα ότι:

$$I \{ E \} Q$$

Αν μετά την εκτέλεση της *E* ισχύει η *S* θα ξαναεκτελεσθεί η *E*. Θα πρέπει λοιπόν από την *Q* && *S* να συνάγεται η αναλλοίωτη. Έτσι, φτάνουμε στον συμπερασματικό κανόνα της **do-while**:

$$\frac{I\{E\}Q,\ Q\&\&S \Rightarrow I}{I\ \{\mathtt{do}\ E\ \mathtt{while}\ (S)\}\ I\ \&\&\ (!S)}$$

Πώς θα μπορούσαμε να υλοποιήσουμε τη **while** με τη **do-while**; Όπως είπαμε παραπάνω «στη **while** πρώτα ελέγχεται η συνθήκη και η εντολή εκτελείται μόνον αν ισχύει, ενώ στη **do-while** πρώτα εκτελείται η εντολή και μετά ελέγχεται η συνθήκη». Το πρόβλημά μας λοιπόν είναι να ελέγξουμε την πρώτη φορά εκτέλεσης της *E*. Αυτό μπορεί να γίνει με μια

> **Πλαίσιο 11.3**
>
> **Η Εντολή do-while**
>
> "do", *εντολή*, "while", "(", *παράσταση*, ")" ;
>
> και εκτελείται ως εξής:
>
> εκτελείται η *εντολή*
> υπολογίζεται η τιμή της *παράστασης*
> αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
> αλλιώς, αν είναι `true` τότε
>
>     εκτελείται η *εντολή*
>     υπολογίζεται η τιμή της *παράστασης*
>     αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
>     αλλιώς, αν είναι `true` τότε
>
>         εκτελείται η *εντολή*
>         υπολογίζεται η τιμή της *παράστασης*
>         κ.ο.κ.

**if**. Έτσι, οι:

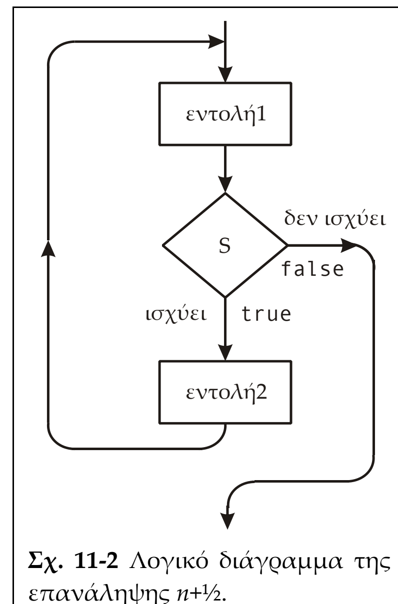
```
while ( S )                    if ( S )
{                              {
   E                              do { E } while ( S );
}                              }
```

είναι ισοδύναμες.

## 11.8 Η Επανάληψη "n+½" – Η Εντολή "break"

Μέχρι τώρα μάθαμε επαναληπτικές εντολές που ελέγχουν τη συνθήκη συνέχισης στην αρχή, όπως η **while** και η **for**, ή στο τέλος, όπως η **do-while**, των επαναλαμβανόμενων εντολών. Φυσικά, η πιο γενική περίπτωση είναι όταν έχουμε τον έλεγχο κάπου ενδιάμεσα. Π.χ. στη γλώσσα Ada υπάρχει η εντολή:

```
loop
  E1
when St exit;
  E2
endloop
```



**Σχ. 11-2** Λογικό διάγραμμα της επανάληψης *n*+½.

Εδώ, εκτελείται η εντολή *E*1 και ελέγχεται η τιμή της συνθήκης *St*:

- αν έχει τιμή `true` τότε διακόπτεται η επανάληψη και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί μετά το `endloop`,
- αν έχει τιμή `false` τότε εκτελούνται οι εντολές *E*2 και *E*1, ελέγχεται η τιμή της συνθήκης *St* κ.ο.κ.

Αυτή η επανάληψη λέγεται **επανάληψη *n*+½**, διότι την τελευταία φορά εκτελείται μόνο η «μισή» περιοχή επανάληψης (*E*1).

Η C++ δεν έχει τέτοια εντολή αλλά δεν είναι δύσκολο να την υλοποιήσουμε. Γράφουμε μια αέναη επανάληψη και μετά θα πρέπει να βρούμε έναν τρόπο για να βγαίνουμε από την επανάληψη όταν θέλουμε. Το «όταν θέλουμε» μπορεί να γίνει με μια **if**. Για την έξοδο από την επανάληψη η C++ μας δίνει την εντολή `break`· εκτέλεσή της έχει ως αποτέλεσμα να συνεχιστεί η εκτέλεση με την εντολή που ακολουθεί την επανάληψη.

Υλοποιούμε λοιπόν την επανάληψη *n*+½ ως εξής:

```
while ( true )          for ( ; ; )             do
{                       {                       {
   E1                      E1                      E1
if ( !S ) break;   ή    if ( !S ) break;   ή    if ( !S ) break;
   E2                      E2                      E2
} // while              } // for                } while ( true );
```

Όπως βλέπεις, σε σχέση με την εντολή της Ada, κάναμε μια μικρή αλλαγή: γράψαμε την **if** έτσι που η συνθήκη *S* να είναι συνθήκη συνέχισης, όπως συμβαίνει στις εντολές επανάληψης της C++.

Και εδώ θα έχουμε μια αναλλοίωτη, *I*, που θα ισχύει πριν ενώ μετά θα ισχύει η: *I* && (!*S*). Για την εντολή *E*1 θα έχουμε:

$$I \{ E1 \} Q$$

Μετά την *E*2 θα εκτελεσθεί ξανά η *E*1· θα πρέπει λοιπόν μετά την *E*2 να ξαναπαίρνουμε την αναλλοίωτη:

$$Q \text{ \&\& } S \{ E2 \} I$$

Αν μετά την *E*1 δεν ισχύει η *S* η εκτέλεση της επανάληψης διακόπτεται. Θα πρέπει λοιπόν να έχουμε:

$$(Q \&\& \,!S) \Rightarrow (I \&\& \,!S)$$

Μπορούμε να απλουστεύσουμε τα πράγματα ζητώντας η $Q$ να είναι η αναλλοίωτη $I$ και να γράψουμε τον κανόνα:

$$\frac{I\{E1\}I,\ I\&\&S\{E2\}I}{I\{\texttt{while(true)}\{E1\ \texttt{if(!}S\texttt{)break;}E2\}\}I\&\&(!S)}$$

Και κάτι ακόμη για τη `break`: Αν έχεις επαναληπτικές εντολές φωλιασμένες, τη μία μέσα στην άλλη, η `break` διακόπτει την εκτέλεση της πιο εσωτερικής επανάληψης μέσα στην οποία βρίσκεται. Για παράδειγμα, δες την παρακάτω περίπτωση:

```
      while ( S1 )
      {
          :
         do {
             . . .
             for (...)
             {
               :
               ... break; ...
             } // for
 A ------>   . . .
         } while ( S2 );
             . . .
      } // while
```

Τρεις επαναληπτικές εντολές, η μια μέσα στην άλλη και στην πιο εσωτερική μια `break`. Τι θα συμβεί με την εκτέλεσή της; Θα διακοπεί η εκτέλεση *της for μόνον* και η εκτέλεση του προγράμματος θα συνεχιστεί στο σημείο A.

## 11.9 Η Εντολή "`switch`"

Ξαναδές τα παραδ. 1 και 2 στην §5.3· και τα δύο είναι προγράμματα με πολλαπλές επιλογές, το δεύτερο όμως, για τον υπολογιστή τσέπης, έχει το εξής χαρακτηριστικό: η εντολή που θα εκτελεσθεί επιλέγεται από την τιμή μιας μεταβλητής (*oper*)· για διαφορετικές –αποδεκτές– τιμές της μεταβλητής εκτελούνται διαφορετικές εντολές.

Η C++ μας δίνει τη δυνατότητα να γράφουμε τέτοια προγράμματα πιο παραστατικά, με χρήση της εντολής **switch**:

```
switch ( παράσταση )
{
   case c1: E1
   case c2: E2
      :
   default: Ed
}
```

Η **switch**, μετά τη λέξη-κλειδί `switch`, περιμένει, μέσα σε παρενθέσεις, μια παράσταση που υπολογίζει τιμή ακέραιου τύπου (`int`, `long int`, `char`, `bool` κλπ). Στη συνέχεια έχει μια (σύνθετη) εντολή. Στην εντολή αυτή υπάρχουν εντολές με *ετικέτες* της μορφής "`case` *σταθερά*. Στο τέλος μπορεί να υπάρχει και εντολή με την ετικέτα `default`.

Όταν εκτελείται η **switch**:

- Υπολογίζεται κατ' αρχάς η τιμή της παράστασης.
- Στη συνέχεια η εκτέλεση συνεχίζεται από την εντολή που η ετικέτα της έχει σταθερά ίση με την τιμή της παράστασης.
- Αν δεν υπάρχει εντολή με τέτοια σταθερά η εκτέλεση συνεχίζεται με την εντολή που έχει ετικέτα `default` (αν υπάρχει).

Όπως καταλαβαίνεις, θα πρέπει να βάζεις ετικέτα `default` όταν οι σταθερές στις ετικέτες δεν καλύπτουν όλες τις τιμές που μπορεί να πάρει η παράσταση.

Αλλά προσοχή! Ας πούμε ότι η παράσταση πήρε την τιμή *c*1. Στην περίπτωση αυτή θα εκτελεσθούν οι εντολές *E*1 και η εκτέλεση θα συνεχιστεί με τις εντολές *E*2 κλπ. Αν θέλεις να εκτελεσθούν μόνον οι *E*1 θα πρέπει να βάλεις στο τέλος τους μια εντολή `break`.

Ας δούμε πώς γράφεται το πρόγραμμα του υπολογιστή τσέπης με τη **switch**:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;           // Τα ορισματα της πράξης
   char oper;               // Το σύμβολο της πράξης

   cin >> oper >> x1 >> x2;
   switch ( oper )
   {
      case '+':  cout << (x1 + x2) << endl;  break;
      case '-':  cout << (x1 - x2) << endl;  break;
      case '*':  cout << (x1 * x2) << endl;  break;
      case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                           else  cout << " Δεν γίνεται" << endl;
                 break;
       default:  cout << " Η πράξη (+,-,*,/) στην πρώτη θέση"
                      << endl;
   } // switch
} // main
```

Στην περίπτωσή μας παράσταση είναι η μεταβλητή *oper* τύπου `char`. Στη συνέχεια έχει μια (σύνθετη) εντολή. Στην εντολή αυτή υπάρχουν εντολές με ετικέτες `case '+'`, `case '-'`, `case '*'`, `case '/'`, `default`. Όταν εκτελείται η **switch** ελέγχεται κατ' αρχάς η τιμή της *oper*. Ας πούμε ότι αυτή βρίσκεται να είναι: `'*'`. Στην περίπτωση αυτήν εκτελείται η εντολή "`cout << (x1 * x2) << endl`" και η "`break`" που την ακολουθεί, οπότε και διακόπτεται η εκτέλεση της **switch**.

Έστω τώρα, ότι θέλουμε να αναγνωρίζει ως σύμβολο πολλαπλασιασμού και το (γράμμα) `'x'` αλλά, όταν δοθεί, εκτός από το αποτέλεσμα να γράφει και το μήνυμα `"προτιμότερο το '*'"`. Να πώς πρέπει να γραφεί το πρόγραμμά μας:

```
   switch ( oper )
   {
      case '+':  cout << (x1 + x2) << endl;  break;
      case '-':  cout << (x1 - x2) << endl;  break;
      case 'x':  cout << "προτιμότερο το \'*\'" << endl;
      case '*':  cout << (x1 * x2) << endl;  break;
      case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                           else  cout << " Δεν γίνεται" << endl;
                 break;
       default:  cout << " Η πράξη (+,-,*,/) στην πρώτη θέση"
                      << endl;
   } // switch
```

Αν δώσουμε:

```
* 3 4
```

παίρνουμε:

```
12
```

ενώ αν δώσουμε:

```
x 3 4
```

παίρνουμε:

```
προτιμότερο το '*'
12
```

Και αν θέλουμε να βγάζει το αποτέλεσμα αλλά όχι το μήνυμα; Τότε δεν βάζουμε την εντολή που βγάζει το μήνυμα. Η παρακάτω **switch** δέχεται τα `'x'`, `'*'` για πολλαπλασιασμό και τα `'/'`, `':'` για διαίρεση:

```
    . . .
    case '-':  cout << (x1 - x2) << endl;  break;
    case 'x':
    case '*':  cout << (x1 * x2) << endl;  break;
    case ':':
    case '/':  if ( x2 != 0 )  cout << (x1 / x2) << endl;
                         else  cout << " Δεν γίνεται" << endl;
               break;
    . . .
```

Πώς θα μπορούσαμε να υλοποιήσουμε τις **if** και **ifelse** μόνον με τη **switch**; Αφού λάβουμε υπόψη μας ότι ο `bool` είναι ακέραιος τύπος, έχουμε τις εξής ισοδυναμίες:

```
if ( S )                      switch ( S )
{                             {
   Et                            case true: Et;
}                             }

if ( S )                      switch ( S )
{                             {
   Et                            case true:  Et; break;
}
else                             case false: Ef;
{                             }
   Ef
}
```

- ***Όταν έχουμε πολλαπλές επιλογές που εξαρτώνται από την (ακέραιη) τιμή μιας παράστασης θα προτιμούμε την εντολή switch από τις φωλιασμένες ifelse.***

### 11.9.1 Τοπικές Μεταβλητές στη “switch”

Η **switch** έχει τη δική της σύνθετη εντολή (από τη ‘`{`‘ μέχρι τη ‘`}`’). Μπορούμε λοιπόν μέσα σε αυτήν να δηλώσουμε τοπικές μεταβλητές! Μπορούμε; Για δες ένα παράδειγμα:

```
switch ( oper )
{
   int q( 5 );

   case '+':  . . .
   case '-':  . . .
    . . .
} // switch
```

Πρόσεξε τώρα: ο μεταγλωττιστής θα πρέπει να βάλει εντολές δημιουργίας της μεταβλητής *q* «αμέσως μετά την ‘`{`‘» (ας πούμε...). Θα βάλει εντολές καταστροφής της *q* «αμέσως πριν από την ‘`}`‘» (ας πούμε πάλι...). Μόνο που η καταστροφή θα γίνει οπωσδήποτε ενώ η δημιουργία δεν θα γίνει ποτέ! Γιατί; Διότι μετά τον έλεγχο της τιμής της *oper* η εκτέλεση θα προχωρήσει κατ' ευθείαν σε κάποιαν από τις ετικέτες case παρακάμπτοντας τη δήλωση της *q*.

Το ίδιο θα συμβεί και σε περιπτώσεις σαν την:

```
switch ( oper )
{
   case '+':  int q( 5 ); . . .
   case '-':  . . .
    . . .
} // switch
```

αν δεν επιλεγεί η περίπτωση ‘`+`’.

Φυσικά, δεν έχεις κανένα τέτοιο πρόβλημα στην περίπτωση:

```
switch ( oper )
{
   case '+':  {  int q( 5 ); . . .  }
   case '-':  . . .
    . . .
```

```
    } // switch
```

## 11.10 * Ετικέτες – Η Εντολή “goto”

Οι περισσότερες γλώσσες προγραμματισμού περιλαμβάνουν την περίφημη (με την κακή έννοια) εντολή **goto** (Dijkstra 1988)· την έχει και η C++. Μέχρι τώρα αποφύγαμε να την παρουσιάσουμε, μια και θα έπρεπε να τη συνοδεύουμε με τη συμβουλή «μην τη χρησιμοποιείς». Τώρα, μπορούμε να την παρουσιάσουμε, να δούμε από που ξεκινούν τα προβλήματα και να καταλάβουμε πότε μπορούμε να τη χρησιμοποιούμε και πότε όχι. Πάντως, η χρήση της δεν είναι απαραίτητη, όπως αποδεικνύεται με το θεώρημα Bohm-Jacopini (Bohm & Jacopini 1966).

Ας ξεκινήσουμε όμως με τις **ετικέτες** (labels), που είναι απαραίτητες για τη χρήση της **goto**.

Εκτός από τις ετικέτες "`default`" και "`case`", *σταθερά*, που είδαμε ότι χρησιμοποιούνται στη **switch**, η C++ σου επιτρέπει να βάλεις μπροστά από οποιαδήποτε εντολή, μια **ετικέτα** π.χ.:

```
gvl: cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
```

Το `gvl` είναι η ετικέτα της εντολής

`cout << " Δώσε...ΤΕΛΟΣ: "`

Στη C++ η ετικέτα είναι ένα αναγνωριστικό και μπαίνει μπροστά από οποιαδήποτε εντολή, ακολουθούμενη από το χαρακτήρα `':'`.

Η εντολή **goto** είναι η πιο «ωμή» παρέκκλιση από τη σειριακή εκτέλεση. Τό νόημά της είναι: «Μη συνεχίσεις με την επόμενη εντολή αλλά με την εντολή που έχει την ετικέτα που σου δίνω».

Η διαφορά της από τις εντολές υπό συνθήκη ή τις επαναληπτικές είναι η εξής: εκεί λέγαμε: παράκαμψε αυτές τις εντολές (ή εκτέλεσε ξανά και ξανά αυτές τις εντολές) αν ισχύει αυτή η συνθήκη. Εδώ είναι σαν να λέμε: πήγαινε να συνεχίσεις εκεί, γιατί έτσι μου αρέσει! Ή τουλάχιστον, έτσι φαίνεται, πολύ συχνά.

Δες το παρακάτω παράδειγμα:

```
     if ( x < 0 )  goto ts;
        absX = x;
        goto nxt;
  ts: absX = -x;
 nxt: cout << x << "  " << absX << endl;
```

Τί κάνουν αυτές οι εντολές; Αυτά τα απλά:

```
      if ( x >= 0 )  absX = x;
               else  absX = -x;
     cout << x << "  " << absX << endl;
```

Πριν προχωρήσουμε να απαριθμήσουμε τα δεινά που θα πέσουν στο κεφάλι μας (ή τουλάχιστον στο πρόγραμμά μας) αν χρησιμοποιούμε την εντολή **goto**, ας δούμε μερικούς περιορισμούς σχετικά με τη χρήση της.

Και ας ξεκινήσουμε με την **εμβέλεια** (scope) μιας ετικέτας.

- Μια ετικέτα μπαίνει σε μια μόνο εντολή στο σώμα μιας συνάρτησης και είναι γνωστή μόνο μέσα στη συνάρτηση αυτή.

Φυσική συνέπεια του παραπάνω περιορισμού είναι ο περιορισμός: *με τη **goto** μπορείς να πηγαίνεις σε άλλα σημεία της ίδιας συνάρτησης.*

Η C++ βάζει άλλον ένα περιορισμό σχετικά με τη χρήση της **goto**, αλλά τον αφήνουμε για αργότερα, αφού τώρα δεν μπορείς να τον κατανοήσεις. Θα βάλουμε όμως έναν περιορισμό που έχει σχέση με τον σωστό προγραμματισμό:

♦ ***Η goto δεν πρεπει να παραπέμπει απ' έξω προς το εσωτερικό δομημένων εντολών.***

Τα παρακάτω *δεν επιτρέπονται*:

```
    . . .
    goto lb55;
    . . .
    while ( i <= n )
    {
        . . .
lb55:   x = x - 1;
        . . .
    } // while
```

(η **goto lb55** παραπέμπει στο εσωτερικό της **while**)

```
    . . .
    if (x <= 0)  goto lb66;
    . . .
    if (x == 1)
       a = a + 1;
    else
lb66:  n = n + 1;
```

(η **goto lb66** παραπέμπει στο εσωτερικό της **ifelse**)

Όπως θα κατάλαβες, από το παράδειγμα που δώσαμε πιο πάνω, η χρήση της **goto** κάνει ένα πρόγραμμα δυσανάγνωστο. Και επειδή είναι παραδεκτό απ' όλους ότι «ένα καλό πρόγραμμα είναι και ευανάγνωστο», η χρήση της **goto** βάει ερωτηματικά για την ποιότητα του προγράμματος.

Τα προβλήματα που προέρχονται από τη χρήση της **goto**, πιο συγκεκριμένα, είναι:

- δυσκολία επαλήθευσης του προγράμματος,
- δυσκολία τροποποίησης του προγράμματος.

Θα τα μελετήσουμε στην επόμενη παράγραφο.

### 11.10.1 Προβλήματα με τη Χρήση της Εντολής goto

Κοίταξε το κομμάτι (υποθετικού) προγράμματος που δίνουμε παρακάτω:

```
          . . .
_____________________ A
      goto lb75;
          . . .
_____________________ B
 lb75: . . .
       . . .
```

Η εντολή που υπάρχει στην **lb75** θα εκτελείται είτε μετά από εκτέλεση της εντολής **goto lb75** είτε, με την κανονική (σειριακή) ροή του προγράμματος, μετά την εκτέλεση της προηγούμενής της εντολής. Έτσι, η συνθήκη που απαιτούμε να ισχύει πριν από την εκτέλεση αυτής της εντολής θα πρέπει να συνάγεται και από την $P_A$ και από την $P_B$. Αυτό δεν είναι τετριμμένο. Και αν εξασφαλισθεί όταν γράφεται αρχικά το πρόγραμμα, είναι πολύ πιθανό ότι θα πάψει να ισχύει όταν γίνει μια διόρθωση ή μια τροποποίηση.

Αν έχεις βέβαια περισσότερες από μια **goto** για την ίδια ετικέτα, τα πράγματα χειροτερεύουν.

Υπάρχουν περιπτώσεις που δεν υπάρχει πρόβλημα με τη χρήση της **goto**; Ναι, όταν δεν υπάρχουν απαιτήσεις από την εντολή που έχει την ετικέτα ή υπάρχουν απαιτήσεις περιορισμένες. Χαρακτηριστική περίπτωση η διακοπή εκτέλεσης λόγω κάποιας απρόβλεπτης κατάστασης (εξαίρεσης), π.χ.

```
lb99: switch ( errCode )
      {
         case 1: cout << " Διαίρεση δια 0" << endl;

         case 2: cout << " Απρόβλεπτο τέλος στο Αρχείο" << endl;

         case 3: cout << " Πολλές τιμές για τον πίνακα Z"
```

```
                  << endl;
            . . .
    } // switch
    exit( EXIT_FAILURE );
```

Τί απαίτηση έχουμε όταν εκτελείται κάποια `goto lb99`; Η *errCode* να έχει το σωστό κωδικό σφάλματος, ώστε να γραφεί το σωστό μήνυμα. Μετά... έχουμε "`exit(EXIT_FAILURE)`". Αυτή η χρήση της **goto** δεν είναι τραγική.

Τώρα ξέρεις!

- Η C++ σου δίνει τη **goto** αλλά και όσα εργαλεία σου χρειάζονται για να μη τη χρησιμοποιείς. Στο κεφάλαιο αυτό προσπαθήσαμε να σου παρουσιάσουμε αυτά τα εργαλεία σε βάθος: για να καταλάβεις το λόγο ύπαρξής τους και την ανάγκη για τη χρήση τους. Χρησιμοποίησέ τα και *απόφυγε τη χρήση της **goto***.
- Οι μεγάλοι μάστορες ξέρουν καλά τους κανόνες, όπως «*μη χρησιμοποιείς τη **goto***», ξέρουν όμως και πότε να τους παραβιάζουν. Αυτό προσπαθήσαμε να σου δείξουμε στην παράγραφο αυτήν. Όταν θα νοιώσεις ότι έγινες μεγάλος μάστορας χρησιμοποίησε και τη **goto**! Φυσικά, ο τρόπος που θα τη χρησιμοποιήσεις θα δείχνει και πόσο μεγάλος μάστορας είσαι...

Ένας μεγάλος μάστορας, ο D. Knuth, έχει γράψει μια, κλασική πια, ανάλυση σε βάθος σχετικά με την εντολή **goto** (Knuth 1977). Αξίζει να τη μελετήσεις· θα σου μάθει πολλά.

## 11.11 * Η Εντολή "continue"

Η εντολή `continue` χρησιμοποιείται μέσα στην περιοχή επανάληψης μιας **while** ή μιας **do-while** ή μιας **for** και σου δίνει τη δυνατότητα να τερματίσεις μια εκτέλεσή της.

- Αν η επανάληψη γίνεται με **while** ή με **do-while**: μετά την (εκτέλεση της) `continue` η εκτέλεση συνεχίζεται με υπολογισμό της συνθήκης.
- Αν η επανάληψη γίνεται με **for**: μετά την `continue` υπολογίζεται η *παράσταση* (τρίτο τμήμα της **for**) και στη συνέχεια υπολογίζεται η συνθήκη.

Η επανάληψη του *Μέση Τιμή* 1, που γράψαμε με **for** στην §11.7, θα μπορούσε να γραφεί ως εξής:

```
for ( sum = 0,  n = 0,  selSum = 0,  selN = 0,
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x;
      x != FROUROS;
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ",  cin >> x )
{
   n = n + 1;                         // Αυτά γίνονται για
   sum = sum + x;                     // όλους τους αριθμούς
   if ( x <= 0 || 10 < x ) continue;  // Έλεγχος - Επιλογή
   selSum = selSum + x;               // Αυτά γίνονται μόνο για
   selN = selN + 1;              // τους επιλεγόμενους αριθμούς
} // for
```

Βέβαια, όπως βλέπεις, η συνθήκη αυτής της **if** είναι η *αντίθετη* της συνθήκης που είχαμε αρχικώς.

Όπως καταλαβαίνεις, μπορείς να ζήσεις και χωρίς την **continue**.

## 11.12 * Ακολουθία Παραστάσεων

Η C++ έχει κληρονομήσει από τη C και μια «περίεργη» πράξη: την ακολουθία παραστάσεων, όπου ο τελεστής της πράξης είναι το κόμμα (`,`). Ας πούμε ότι δηλώνουμε:

```
double x( 0.1 ), y( 3.7 ), z( 7.4 ), q;
```

και δίνουμε:

```
q = (x, y, z);
```

```
    cout << q << endl;
```

Μέσα στην παρένθεση έχουμε τρεις (πολύ απλές) παραστάσεις. Αυτό που θα γίνει είναι το εξής: θα υπολογισθούν οι τιμές των τριών παραστάσεων –δηλαδή: 0.1, 3.7 και 7.4 αντιστοίχως– και η τιμή της τελευταίας είναι τιμή της ακολουθίας παραστάσεων που θα εκχωρηθεί στην *q*. Πράγματι, η εντολή εξόδου μας δίνει:

```
7.4
```

Η παρένθεση χρειάζεται; Ναι! Δες τι θα γίνει αν γράψουμε:

```
    q = x, y, z;
    cout << q << endl;
```

Αποτέλεσμα:

```
0.1
```

Γιατί; Διότι ο τελεστής = της εκχώρησης έχει μεγαλύτερη προτεραιότητα από το "`,`" και έτσι πρώτα θα εκχωρηθεί η τιμή της *x* στην *q* και μετά θα υπολογισθεί η τιμή της ακολουθίας (που θα είναι και πάλι 7.4).

Να και μια πιο δύσκολη περίπτωση: Οι

```
    q = (x = 8.3, y = x - 4, z = y - 4);
    cout << q << "  " << x << "  " << y << "  " << z << endl;
```

θα δώσουν:

```
0.3  8.3  4.3  0.3
```

Όπως είδες, χρησιμοποιήσαμε την ακολουθία παραστάσεων στην εντολή **for** και μόνον. Γενικώς, δεν θα τη δεις και πολύ συχνά μπροστά σου.

## 11.13 Υπολογισμός Παράστασης

Στον πίνακα του Παρ. Ε βλέπεις τα χαρακτηριστικά των πράξεων που έχουμε μάθει μέχρι τώρα. Δεν βλέπεις την προσεταιριστικότητα των πράξεων, αλλά γι' αυτήν ο κανόνας είναι απλός:

- η προσεταιριστικότητα των δυϊκών πράξεων, εκτός από την εκχώρηση, είναι από τα αριστερά προς τα δεξιά,
- η προσεταιριστικότητα των ενικών πράξεων και της εκχώρησης είναι από τα δεξιά προς τα αριστερά.

Για παράδειγμα: η `a + b + c` υπολογίζεται ως `(a + b) + c`, ενώ η `a = b = c` υπολογίζεται ως `a = (b = c)`.

Αλλά προσοχή! Αν έχουμε `a*b + c/d + f(e)` ο παραπάνω κανόνας μας λέει *μόνον* ότι ο υπολογισμός θα γίνει ως εξής: `(a*b + c/d) + f(e)`.

- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `(a*b + c/d)` και μετά η `f(e)`.
- *Δεν μας εγγυάται* ότι πρώτα θα υπολογιστεί η `a*b` και μετά η `c/d`.

Κάθε υλοποίηση της C++ θα κάνει τους υπολογισμούς με τη σειρά που «θέλει». Αν θέλεις να επιβάλεις συγκεκριμένη σειρά εκτέλεσης των πράξεων θα πρέπει να σπάσεις την παράσταση σε μικρότερες και να τις γράψεις με την κατάλληλη σειρά.

Στις λογικές πράξεις:

- Αν ένας από τους παράγοντες της πράξης "&&" υπολογιστεί `false` δεν είναι καθόλου σίγουρο ότι θα υπολογιστεί και ο άλλος, αφού το αποτέλεσμα της πράξης θα είναι `false`.
- Αν ένας από τους παράγοντες της πράξης "`||`" υπολογιστεί `true` δεν είναι καθόλου σίγουρο ότι θα υπολογιστεί και ο άλλος, αφού το αποτέλεσμα της πράξης θα είναι `true`.

Στο Παρ. Ε μπορείς να βρεις επίσης τις *Συνήθεις Αριθμητικές Μετατροπές* (ΣΑΜ), δηλαδή ορισμένες μετατροπές τύπων που γίνονται αυτομάτως όταν υπολογίζεται μια αριθ-

μητική παράσταση. Οι ΣΑΜ εφαρμόζονται όταν γίνονται οι πράξεις "+", "-", "*", "/" μεταξύ αριθμητικών τιμών και καθορίζουν τον τύπο του αποτελέσματος.

## 11.14 Εν Κατακλείδι...

Όπως βλέπεις, η C++ μας δίνει πολλά εργαλεία. Πότε θα τα χρησιμοποιούμε και πότε όχι; Μια γενική συμβουλή είναι η εξής:

♦ ***Χρησιμοποιούμε τα προγραμματιστικά εργαλεία στο βαθμό που το πρόγραμμά μας γράφεται έτσι που να είναι καθαρή η λογική του και απλή η απόδειξη ορθότητας.***

Εξειδικεύουμε τη συμβουλή μας:

1. Μπορείς να χρησιμοποιείς τις συντομογραφίες της εκχώρησης σε ανεξάρτητες εντολές αλλά όχι μέσα σε παραστάσεις. Δηλαδή, δεν υπάρχει πρόβλημα αν γράψεις `x += y*a/q` ή `++p` αλλά απόφευγε να γράφεις `y = ++p + (x = q/2)`. *Σε κάθε εντολή εκχώρησης θα πρέπει να αλλάζει η τιμή μιας μεταβλητής μόνον*. Η μόνη περίπτωση που εξαιρείται από τον κανόνα είναι η πολλαπλή εκχώρηση: `x = y = z =` *Π*, με την προϋπόθεση, φυσικά, ότι η *Π* δεν έχει εκχωρήσεις.
2. Η **for** είναι βολική. Γενικά είναι πολύ βολικό να βλέπεις *σε μια γραμμή*, αρχικές εντολές, συνθήκη συνέχισης και τρόπο μεταβολής των μεταβλητών σου. Χρησιμοποίησε ως οδηγό τη *μία γραμμή*. Αν την υπερβαίνεις καλύτερα να χρησιμοποιείς **while**.
3. Στη **switch** μη ξεχνάς τις **break**. Αν πρέπει να μη βάλεις **break** (σαν το προτελευταίο παράδειγμα της §11.10) γράψε σχόλιο που να το τονίζει.
4. Απόφευγε τη χρήση της **goto**.

Αυτά ως συμβουλές· βέβαια υπάρχει και η πείρα που οδηγεί σε προσωπικές προτιμήσεις και σε προσωπικό ύφος. Στο τέλος-τέλος, δημοκρατία έχουμε και... de gustibus et coloribus non disputandum est.[9]

## Ασκήσεις

### Α Ομάδα

**11-1** Στην άσκ. 5-4 (*Μέρος Α*) ρωτούσαμε «*ποια μαθηματική συνάρτηση υλοποιούν οι παρακάτω εντολές (όπου οι μεταβλητές a, b, c και d είναι τύπου* `int`)»:

```
if ( a > b )  y = a; else y = b;
if ( c > y )  y = c;
if ( d > y )  y = d;
```

Προφανώς, η *y* παίρνει ως τιμή, τελικώς, τη μέγιστη από τις τιμές των *a*, *b*, *c*, *d*.

Μπορείς να δώσεις στη *y* την ίδια τιμή με παράσταση υπό συνθήκη;

**11-2** Ξαναλύσε την άσκ. 5-6 (*Μέρος Α*) χρησιμοποιώντας τη **switch**: "*Γράψε πρόγραμμα που θα διαβάζει τις τιμές των $R_1$ και $R_2$ (θετικούς αριθμούς) και θα υπολογίζει και θα τυπώνει τη συνολική αντίσταση R. Πριν πάρει τις τιμές των αντιστάσεων, το πρόγραμμα θα ζητάει να διαβάσει, από το πληκτρολόγιο, τον τρόπο σύνδεσης των αντιστάσεων. Οι δεκτές απαντήσεις θα είναι:*

- `'P'` ή `'p'` για παράλληλη σύνδεση,
- `'S'` *ή* `'s'` *για σύνδεση εν σειρά.*

*Να διατυπωθεί η προϋπόθεση και το πρόγραμμα να την ελέγχει.*"

[9] για γεύσεις και χρώματα δεν (πρέπει να) υπάρχει διένεξη.

**11-3** Ξαναλύσε την άσκ. 6-15 (*Μέρος Α*) χρησιμοποιώντας τις εντολές που έμαθες σε αυτό το μάθημα: “*Θέλουμε ένα πρόγραμμα που θα παρακολουθεί έναν παίκτη του μπάσκετ κατά τη διάρκεια ενός παιγνιδιού. Το πρόγραμμα θα παίρνει τα εξής στοιχεία:*
`'1'` *κάθε φορά που ο παίκτης επιτυγχάνει καλάθι με ελεύθερη βολή,*
`'2'` *κάθε φορά που ο παίκτης επιτυγχάνει δίποντο,*
`'3'` *κάθε φορά που ο παίκτης επιτυγχάνει τρίποντο και*
`'4'` *κάθε φορά που ο παίκτης κάνει φάουλ.*
*Πληκτρολογώντας* `'0'` *δείχνουμε στο πρόγραμμα ότι τελείωσε το παιχνίδι. Όταν ο παίκτης συμπληρώσει πέντε φάουλ, θα πρέπει να βγαίνει το μήνυμα:* `"ΒΓΑΙΝΕΙ ΕΞΩ ΜΕ 5 ΦΑΟΥΛ"`.

*Όταν τελειώσει η συμμετοχή του παίκτη –είτε λόγω αποβολής είτε λόγω τέλους του παιχνιδιού– θα γράφεται στην οθόνη η στατιστική του.*”

**11-4** Στο *Μέρος Α* μάθαμε να διαβάζουμε τις τιμές των στοιχείων ενός πίνακα με *N* θέσεις, από ένα αρχείο μέσω του ρεύματος *t* με τις εξής εντολές:

```
m = 0;  t >> x[m];
while ( !t.eof() && m < N-1 )
{
   m = m + 1;  t >> x[m];
} // while
if ( t.eof() )  count = m;
          else  count = m + 1;
```

Τώρα που έμαθες όλες της δυνατότητες της **for**, ξαναγράψε τα παραπάνω με **for**.

**11-5** Ξαναλύσε την άσκ. 9-1 χρησιμοποιώντας τις εντολές που έμαθες σε αυτό το μάθημα: “*Έστω ένας πίνακας*

```
double a[10];
```

*Γράψε εντολές που θα τον «αναποδογυρίσουν». Δηλαδή να αντιμεταθέσουν τις τιμές των στοιχείων του:*

(a[0] ↔ a[9])   (a[1] ↔ a[8])   (a[2] ↔ a[7])   (a[3] ↔ a[6])   (a[4] ↔ a[5])”