Μέρος Β
Τεχνικές Προγραμματισμού

11. Πράξεις και Εντολές 291
12. Πίνακες ΙΙ – Βέλη 315
13. Συναρτήσεις ΙΙ 343
14. Συναρτήσεις ΙΙΙ 389
15. Δομές - Αρχεία ΙΙ 431
    project 1: Αυτοκίνητα στον Δρόμο 477
    project 2: Διανύσματα στις 3 Διαστάσεις 485
16. Δυναμική Παραχώρηση Μνήμης 493
17. * Εσωτερική Παράσταση Δεδομένων 539
18. Προετοιμάζοντας Βιβλιοθήκες 581

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;
```

🖘🖘🖘

Οι παρενθέσεις στις συνθήκες δεν είναι απαραίτητες.

11.6 Η Εντολή "for"

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

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

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

ή

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

Αν $b > 0$,

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

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

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

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

όπου:

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

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

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

Παράδειγμα 1🖘

Έστω ότι έχουμε δύο μονοδιάστατους πίνακες και μια μεταβλητή

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

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

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

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

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

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

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

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

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

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

Βλέπουμε λοιπόν ότι:

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

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

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

Παράδειγμα 2

Ξαναγράφουμε το πρόγραμμα Μέση Τιμή 1 της §5.1:

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

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

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

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

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

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

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

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

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

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

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

ή, με τις συντομογραφίες:

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

Και έτσι μείναμε χωρίς επαναλαμβανόμενη εντολή!

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

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

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

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

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

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

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

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

Γενικώς, η:

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

μπορεί να γραφεί ως:

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

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

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

και η περιοχή της επανάληψης δεν μπορεί να μικρύνει περισσότερο.

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

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

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

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

1. ```
   // P
      Ep1; Ep2; ... Epn;
   // I
   ```
2. ```
   // I && S
      Er1; Er2; ... Erk; Er(k+1), Er(k+2), ... Er(k+m);
   // I
   ```
3. (*I* && !*S*) ⇒ *Q*

11.7 Η Εντολή "do-while"

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

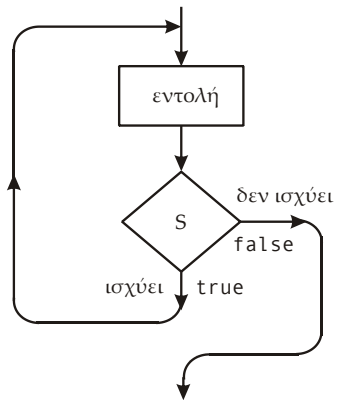
Η σημαντική διαφορά της από τη **while** είναι ότι

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

Έτσι:

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

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



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

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

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

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

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

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

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

Έστω τώρα ότι:

$$I\ \{\ E\ \}\ Q$$

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

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

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

Πλαίσιο 11.3

Η Εντολή do-while

"`do`", *εντολή*, "`while`", "`(`", *παράσταση*, "`)`" ;

και εκτελείται ως εξής:

εκτελείται η *εντολή*
υπολογίζεται η τιμή της παράστασης
αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
αλλιώς, αν είναι `true` τότε

> εκτελείται η *εντολή*
> υπολογίζεται η τιμή της παράστασης
> αν είναι `false` τερματίζεται η εκτέλεση της **do-while**
> αλλιώς, αν είναι `true` τότε
>
> > εκτελείται η *εντολή*
> > υπολογίζεται η τιμή της παράστασης
> > κ.ο.κ.

if. Έτσι, οι:

```
while ( S )
{
   E
}
```

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

είναι ισοδύναμες.

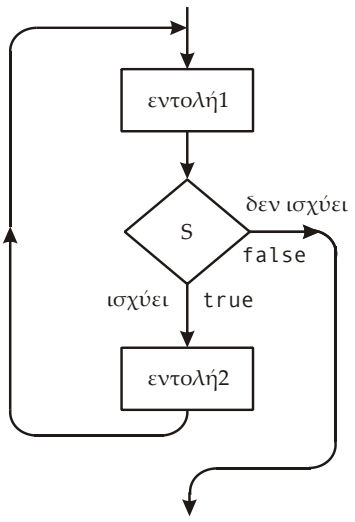
11.8 Η Επανάληψη "n+½" – Η Εντολή "break"

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

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

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

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



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

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

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

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

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

ή

```
for ( ; ; )
{
   E1
if ( !S ) break;
   E2
} // for
```

ή

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

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

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

$$I \{ E1 \} Q$$

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

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

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

$$(Q \texttt{ \&\& !}S) \Rightarrow (I \texttt{ \&\& !}S)$$

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

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

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

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

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

11.9 Η Εντολή "switch"

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

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

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

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

Όταν εκτελείται η **switch**:

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

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

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

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

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

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

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

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

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

Αν δώσουμε:

```
* 3 4
```

παίρνουμε:

```
12
```

ενώ αν δώσουμε:

```
x 3 4
```

παίρνουμε:

```
προτιμότερο το '*'
12
```

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

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

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

```
if ( S )
{
   Et
}
```

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

```
if ( S )
{
   Et
}
else
{
   Ef
}
```

```
switch ( S )
{
   case true:  Et; break;

   case false: Ef;
}
```

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

11.9.1 Τοπικές Μεταβλητές στη "switch"

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

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

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

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

Το ίδιο θα συμβεί και σε περιπτώσεις σαν την:

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

αν δεν επιλεγεί η περίπτωση '+'.

Φυσικά, δεν έχεις κανένα τέτοιο πρόβλημα στην περίπτωση:

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

```
   } // switch
```

11.10 * Ετικέτες – Η Εντολή "goto"

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

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

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

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

Το `gvl` είναι η ετικέτα της εντολής

```
cout << " Δώσε...ΤΕΛΟΣ: "
```

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

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

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

Δες το παρακάτω παράδειγμα:

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

Τί κάνουν αυτές οι εντολές; Αυτά τα απλά:

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

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

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

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

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

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

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

Τα παρακάτω *δεν επιτρέπονται*:

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

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

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

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

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

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

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

Θα τα μελετήσουμε στην επόμενη παράγραφο.

11.10.1 Προβλήματα με τη Χρήση της Εντολής goto

Κοίταξε το κομμάτι (υποθετικού) προγράμματος που δίνουμε παρακάτω:

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

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

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

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

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

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

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

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

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

Τώρα ξέρεις!

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

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

11.11 * Η Εντολή "continue"

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

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

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

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

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

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

11.12 * Ακολουθία Παραστάσεων

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

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

και δίνουμε:

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

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

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

```
7.4
```

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

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

Αποτέλεσμα:

```
0.1
```

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

Να και μια πιο δύσκολη περίπτωση: Οι

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

θα δώσουν:

```
0.3  8.3  4.3  0.3
```

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

11.13 Υπολογισμός Παράστασης

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

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

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

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

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

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

Στις λογικές πράξεις:

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

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

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

11.14 Εν Κατακλείδι...

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

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

Εξειδικεύουμε τη συμβουλή μας:

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

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

Ασκήσεις

Α Ομάδα

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

κεφάλαιο

12

Πίνακες ΙΙ – Βέλη

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να χρησιμοποιείς πολυδιάστατους (συνήθως δισδιάστατους) πίνακες και βέλη. Με τα βέλη μόλις αρχίζουμε...

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς πολυδιάστατους πίνακες στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *πολυδιάστατος πίνακας*
- *αποθήκευση στοιχείων πίνακα*
- *παράμετρος-πίνακας*
- *βέλος* (*pointer*)
- *αριθμητική βελών*
- *τυποθεώρηση* `const`
- *παράμετροι της* **main**

Περιεχόμενα:

12.1 Πίνακες και Βέλη ... 316
12.2 Για τον Περιορισμό "const" ... 318
12.2.1 Τυποθεώρηση "const" ... 318
12.3 Πράξεις με Βέλη ... 320
12.3.1 Μια Θέση Μετά το Τέλος ... 321
12.3.2 Αρχική Τιμή και Εκχώρηση ... 321
12.3.3 Πρόσθεση και Αφαίρεση ... 323
12.3.4 Ο Τύπος "ptrdiff_t" ... 326
12.3.5 Συγκρίσεις ... 326
12.4 Πολυδιάστατοι Πίνακες ... 327
12.5 Η Σειρά Αποθήκευσης ... 333
12.5.1 Τρισδιάστατοι και Πολυδιάστατοι Πίνακες ... 334
12.6 Παράμετρος Πίνακας (ξανά) ... 335
12.6.1 Και Άλλα Τεχνάσματα ... 337
12.7 Οι Παράμετροι της main ... 338
12.8 Τελικώς 339
Ασκήσεις ... 339
Α Ομάδα ... 339
Β Ομάδα ... 340
Γ Ομάδα ... 340

Εισαγωγικές Παρατηρήσεις:
Μέχρι τώρα, στο Μέρος Α, μάθαμε να χρησιμοποιούμε −και χρησιμοποιήσαμε− μονοδιάστατους πίνακες. Φυσικά, σε πολλές περιπτώσεις χρειαζόμαστε πολυδιάστατους πίνακες· συνήθως δισδιάστατους.

Δυνατότητα για χρήση τέτοιων πινάκων μας δίνουν όλες οι γλώσσες προγραμματισμού. Εδώ όμως η C++ (και η C) έχουν μια ιδιαιτερότητα: Για να μπορέσεις όμως να τους αξιοποιήσεις πλήρως θα πρέπει να ξέρεις τα «μυστικά» της υλοποίησής τους.

Το βασικό εργαλείο για τον σκοπό αυτόν είναι το **βέλος** (pointer). Θα αρχίσουμε λοιπόν να το μαθαίνουμε καλύτερα και −καλώς ή κακώς− θα το βρούμε μπροστά μας πολλές φορές στη συνέχεια,

12.1 Πίνακες και Βέλη

Στην §10.13 λέγαμε ότι αν ορίσεις:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ','π','ό','ν','τ','ο','ι','\0' };
```

και δώσεις:

```
cout << a << endl;
```

θα δεις στην οθόνη σου:

```
πάν μέτρον εκατό πόντοι
```

Και τι θα γίνει αν, ας πούμε, δηλώσεις:

```
double x[] = { 0.1, 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9 };
```

και δώσεις:

```
cout << x << endl;
```

Έκπληξη! Να το αποτέλεσμα:

```
0x22ff20
```

(ή κάτι παρόμοιο.)

Τι είναι αυτό; Αλλά, κάτι μας θυμίζει... Είχαμε δει κάτι τέτοιο στην §2.7.1. Λέγαμε ότι πρόκειται για μια ακέραιη τιμή γραμμένη στο δεκαεξαδικό σύστημα, μια *τιμή-βέλος*, που δείχνει μια *διεύθυνση* (θέση) της μνήμης. Λες να είναι το ίδιο πράγμα; Ας ζητήσουμε τη διεύθυνση του πρώτου στοιχείου (x[0]) του πίνακα· θα χρησιμοποιήσουμε τον τελεστή "`&`":

```
cout << &(x[0]) << endl;
```

Αποτέλεσμα:

```
0x22ff20
```

Η ίδια τιμή!

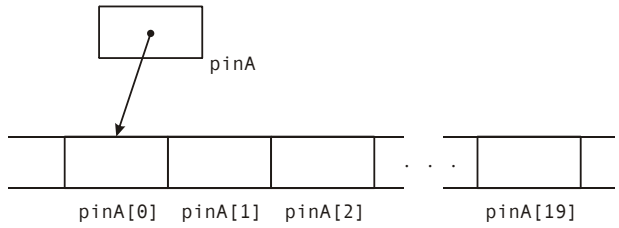
Κάτι παρόμοιο θα συμβεί όποιος και αν είναι ο τύπος (εκτός από `char`) των συνιστωσών του πίνακα.

Φυσικά δεν είναι τυχαίο. Για τη C++:

- ***Ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.***

Στο Σχ. 12-1 ξαναδίνουμε το Σχ. 9-1 αναθεωρημένο σε συμφωνία με τα παραπάνω. Όπως καταλαβαίνεις, αν κάνεις αποπαραπομπή στο όνομα του πίνακα, θα πρέπει να πάρεις το x[0]:

```
cout << (*x) << endl;
```



Σχ. 12-1 »Διόρθωση» του Σχ. 9-1. Η σχηματική παράσταση πίνακα και των στοιχείων του με το βέλος προς το πρώτο στοιχείο (`pinA[0]`) του πίνακα.

Αποτέλεσμα:

```
0.1
```

Τι διαφορά έχει το βέλος `x` από το `p` που δηλώνουμε ως:

```
double* p;
```

Το `x` είναι ένα *σταθερό βέλος* ενώ η `p` είναι μια *μεταβλητή-βέλος*. Αν έχουμε δηλώσει:

```
double r1, r2;
```

μπορούμε να γράψουμε:

```
p = &r1;  p = &r2;
```

ενώ απαγορεύεται να αλλάξουμε την τιμή του βέλους `x`.

Με βάση αυτά, δες πώς μπορεί να γραφεί η *vectorSum()* που πρωτοείδαμε στην §9.3:

```
double vectorSum( const double* x, int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Πρόσεξε ότι το μόνο που άλλαξε είναι η επικεφαλίδα, που ήταν:

```
double vectorSum( const double x[], int n, int from, int upto )
```

Στο σώμα της συνάρτησης δεν υπάρχει αλλαγή. Επίσης, δεν υπάρχει αλλαγή στη χρήση.

Παρατηρήσεις: ▶

1. Πάντως αξίζει να κάνουμε μερικές αλλαγές μέσα στο σώμα της συνάρτησης που δεν έχουν σχέση με το βέλος:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   double sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

2. Ο συμβολισμός με το βέλος δεν είναι κατ' ανάγκην προτιμότερος.

- Θα δεις στη συνέχεια ότι πολλές φορές θέλουμε να περάσουμε σε μια συνάρτηση (μέσω παραμέτρου) βέλος προς κάποιο αντικείμενο.
- Μέσα στη συνάρτηση δεν έχεις τρόπο να ξεχωρίσεις αν μια παράμετρος-βέλος δείχνει μια μεταβλητή ή έναν πίνακα.

Για λόγους τεκμηρίωσης λοιπόν, γενικώς,

♦ ***Όταν έχουμε παράμετρο-πίνακα θα προτιμούμε τον συμβολισμό με τις αγκύλες.***

και θα χρησιμοποιούμε παράμετρο-βέλος όταν περνούμε βέλος προς ένα αντικείμενο.

Ειδικώς για ορμαθούς χαρακτήρων της C –δηλαδή πίνακες χαρακτήρων με τον φρουρό `'\0'` στο τέλος– και οι δύο συμβολισμοί είναι καλοί αφού ναι μεν ο ορμαθός παριστάνεται με πίνακα αλλά θεωρείται ένα αντικείμενο.◀

Στην §9.3 γράφαμε ακόμη: «*Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων* [*ενός πίνακα*]*; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη vectorSum()* [*χωρίς παραμέτρους περιοχής επεξεργασίας*] *και για την περίπτωση αυτή.*» Τώρα μπορούμε να το δούμε αυτό. Η απλή *vectorSum()* είναι:

```
double vectorSum( const double* x, int n )
{
```

```
    double sum( 0 );

    for ( int m(0); m < n; ++m ) sum += x[m];
    return sum;
} // vectorSum
```

Για τον πίνακα που δηλώσαμε πιο πάνω, τα πέντε τελευταία στοιχεία είναι τα x[4], x[5], x[6], x[7], x[8]. Αφού η συνάρτηση περιμένει το πρώτο όρισμα να δείχνει την αρχή του πίνακα και το δεύτερο το πλήθος των στοιχείων του την καλούμε ως εξής:

```
cout << vectorSum( &x[4], 5 ) << endl;
```

Δηλαδή, «ξεγελάμε» τη συνάρτηση περνώντας της για αρχή του πίνακα το στοιχείο `x[4]`.

Προσοχή όμως: Αυτό είναι ένα *τέχνασμα* που στηρίζεται στο ότι ξέρουμε πώς γίνεται η υλοποίηση των εννοιών στη C++ (και στη C)[1]. Το να περνούμε τον πίνακα (`const double x[], int n`) και την περιοχή επεξεργασίας (`int from, int upto`) είναι *πάγια τεχνική*.

12.2 Για τον Περιορισμό "const"

Λέγαμε στην προηγούμενη παράγραφο, μετά τη δήλωση "`double x[] = {...}`": «*Το* `x` *είναι ένα σταθερό βέλος* [...] *απαγορεύεται να αλλάξουμε την τιμή του βέλους* `x`.» Πράγματι, αν έχεις δηλώσει:

```
double r1;
```

η εντολή "`x = &r1`" δεν θα γίνει δεκτή από τον μεταγλωττιστή.

Παρ' όλο που τα διαγνωστικά είναι διαφορετικά, αυτό μας θυμίζει την περίπτωση που δηλώναμε (§2.4):

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

και –στη συνέχεια– ο μεταγλωττιστής δεν μας επιτρέπει τη "`g = 5.32`".

Πάντως, αν ξεχάσουμε την υλοποίηση και το βέλος και δεχθούμε ότι τιμή ενός πίνακα είναι οι τιμές όλων των στοιχείων του, τότε πιο κοντά σε αυτήν τη χρήση του "`const`" είναι αυτό που μάθαμε στην §9.3, στην παράμετρο-πίνακα μιας συνάρτησης:

```
double vectorSum( const double x[], int n, int from, int upto )
```

την οποία μπορούμε να γράψουμε και ως: "`const double* x`".

Με αυτά που μαθαίνουμε τώρα, βλέπουμε ότι, όταν πρόκειται για βέλος, υπάρχει και άλλη δυνατότητα: να αλλάζει ή να μην αλλάζει η τιμή του βέλους (δηλαδή η διεύθυνση της μνήμης που δείχνει). Μπορείς να επιβάλεις τη *σταθερότητα του βέλους* με την εξής δήλωση:

```
double* const x( &r1 );
```

Τελος, για να έχουμε *σταθερό βέλος προς σταθερή τιμή* θα πρέπει να δηλώσουμε:

```
const double* const x( &g );
```

Τέτοιες δηλώσεις θα βρεις συνήθως στις τυπικές παραμέτρους συναρτήσεων. Εκεί φυσικά δεν βάζουμε αρχική τιμή· αυτή καθορίζεται αυτομάτως όταν καλείται η συνάρτηση και είναι η αντίστοιχη πραγματική παράμετρος.

12.2.1 Τυποθεώρηση "const"

Ας ξαναγυρίσουμε στην §9.5.1. Στη συνάρτηση *linSearch()* (με φρουρό) υποχρεωθήκαμε να αλλάξουμε το "`const int v[]`" σε "`int v[]`" διότι ο μεταγλωττιστής δεν δεχόταν τις εντολές:

```
        v[upto+1] = x;       // φρουρός
    // . . .
```

[1] ... και –καλώς ή κακώς– θα το δεις σε πολλά βιβλία, σε πολλούς διαδικτυακούς τόπους κλπ.

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Αν σκεφτούμε ότι στην πραγματικότητα "`const int v[]`" σημαίνει "`const int* v`" μπορούμε να κάνουμε το εξής: Ορίζουμε μια μεταβλητή

```
        int* ncv( v );
```

Η *ncv* είναι βέλος προς `int`, όπως και η *v*, αλλά δεν είναι `const`. Βάζοντας την *ncv* να δείχνει όπου και η *v* θα μπορούμε να τροποιούμε τα στοιχεία του πίνακα *v* μέσω αυτής. Αυτό όμως δεν μπορεί να γίνει ή τουλάχιστον δεν μπορεί να γίνει έτσι· δεν το επιτρέπει ο μεταγλωττιστής. Μπορεί να γίνει με τη λεγόμενη **τυποθεώρηση const**:

```
        int* ncv( const_cast<int*>(v) );
```

δηλαδή: το ncv δείχνει τον ίδιο πίνακα που δείχνει και το v, αλλά, αφού δεν έχει "`const`" (`const_cast<int*>`) μας δίνει συνατότητα τροποποίησης.

Να η νέα μορφή της *linSearch()*:

```
 0: int linSearch( const int v[], int n,
 1:                int from, int upto, int x )
 2: {
 3:    int fv( -1 );
 4:
 5:    if ( 0 <= from && from <= upto && upto < n )
 6:    {
 7:       int* ncv( const_cast<int*>(v) );
 8:       int save( v[upto+1] ); // φύλαξε το v[upto+1]
 9:       ncv[upto+1] = x;       // φρουρός
10:       int k( from );
11:       while ( v[k] != x )  ++k;
12:       if ( k <= upto )
13:          fv = k;
14:       ncv[upto+1] = save;    // όπως ήταν στην αρχή
15:    // (from <= fv <= upto && v[fv] == x) ||
16:    //       (fv == -1 && (για κάθε j:from..upto • v[j] != x))
17:    }
18:    return fv;
19: } // linSearch
```

Πρόσεξε τώρα τα εξής:

- Στην επικεφαλίδα (γρ. 0) *δεν αλλάξαμε* το "`int v[]`" σε "`int* v`". Όπως είπαμε, το πρώτο είναι προτιμότερο μια και δείχνει πίνακα.
- Μεταφέραμε τις δηλώσεις των *save* (γρ. 8) και *k* (γρ. 10) εκεί που μπορούμε να ορίσουμε και την αρχική τους τιμή.
- Στη γρ. 7 βάλαμε τη δήλωση του *ncv* με την τυποθεώρηση `const`.
- Έτσι, όταν στη γρ. 9 και στη γρ. 14 αλλάζουμε την τιμή του ncv`[`*upto*+1`]` αλλάζουμε στην πραγματικότητα την τιμή του v`[`*upto*+1`]`.

Με την τυποθεώρηση **const** μπορείς να αφαιρείς ή να προσθέτεις τον περιορισμό "`const`" (ή τον "`volatile`"). Για παράδειγμα, ας πούμε ότι θέλουμε να έχουμε έναν πίνακα που οι τιμές των στοιχείων του δεν θα αλλάζουν κατά τη διάρκεια εκτέλεσης του προγράμματος. Αλλά, αυτές οι τιμές δεν είναι σταθερές γνωστές όταν γράφουμε το πρόγραμμα. Θα πρέπει κάθε φορά που εκτελείται να διαβάζονται από κάποιο αρχείο. Μπορούμε να κάνουμε το εξής:

```
   double xf[100];
// ανάγνωση τιμών του xf από το αρχείο
   const double* x( const_cast<const double*>(xf) );
```

Τώρα, μέσω του x, μπορούμε να χρησιμοποιούμε τα στοιχεία χωρίς να υπάρχει περίπτωση να αλλάξουμε κάποια τιμή κατά λάθος. Δυστυχώς όμως και το xf υπάρχει και η ζημιά μπορεί να γίνει από εκεί.

Να ένας άλλος τρόπος πιο ασφαλής:

```
   const double x[100] = { 0 };
   double* xf( const_cast<double*>(x) );
```

```
// ανάγνωση τιμών του xf από το αρχείο
   xf = 0;
```

Εδώ χρησιμοποιούμε το βέλος xf για να δώσουμε τιμές στα στοιχεία του x, που είναι δηλωμένος `const`. Μετά από αυτό, του βάζουμε τιμή 0 και τον «αποσυνδέουμε» από τον πίνακα x. Για την τιμή βέλους 0 τα λέμε παρακάτω.

Αργότερα θα δούμε παράδειγμα τυποθεώρησης **const** και σε παράμετρο αναφοράς

Τα ίδια μπορείς να κάνεις και με το "`volatile`" αλλά κάτι τέτοιο δεν είναι και τόσο χρήσιμο.

12.3 Πράξεις με Βέλη

Τώρα μπορούμε να δούμε και κάτι άλλο: Με τα βέλη προς στοιχεία πίνακα μπορείς να κάνεις και πράξεις, για την ακρίβεια πρόσθεση και αφαίρεση. Έτσι, έχουν νόημα τα `x+1`, `x+2`,... που μας δίνουν τις θέσεις των στοιχείων `x[1]`, `x[2]`,... Με αποπαραπομπή (`*(x+1)`, `*(x+2)`,...) παίρνουμε τα στοιχεία:

```
   cout << (*x) << "  " << (*(x+1)) << "  " << (*(x+2)) << endl;
```

Αποτέλεσμα:

```
0.1  1.2  2.3
```

Εδώ πρόσεξε το εξής: Οι παρενθέσεις *είναι απαραίτητες* διότι *η πράξη της αποπαραπομπής έχει μεγαλύτερη προτεραιότητα από την πρόσθεση*. Πράγματι από την

```
   cout << (*x+1) << "  " << (*(x+1)) << endl;
```

θα πάρουμε:

```
1.1  1.2
```

Το `1.1` είναι στην πραγματικότητα το αποτέλεσμα της πράξης: `*x+1` που δεν είναι τίποτε άλλο από το `x[0]+1`.

Να λοιπόν η σχέση μεταξύ *δείκτη στοιχείου* και *βέλους προς στοιχείο* πίνακα: Αν

```
T x[N];
```

και *k* φυσικός από 0 μέχρι *N*-1, τότε:

```
x[k] == *(x + k)
```

Πρόσεξε το εξής: το `*(x+k)` δεν είναι η τιμή του στοιχείου αλλά το ίδιο το στοιχείο (δηλαδή μια *τιμή-l*). Αυτό σημαίνει ότι μπορείς να δώσεις:

```
   *(x+3) = 7.77;
```

για να αλλάξεις την τιμή του `x[3]`.

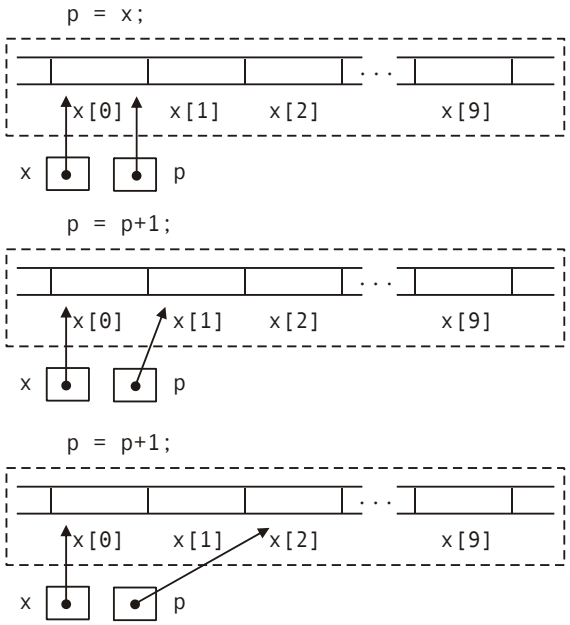
Είδαμε πιο πριν ότι μπορούμε να υπολογίσουμε το άθροισμα των στοιχείων ενός πίνακα με τις:

```
      double sum( 0 );
      for ( int m(0); m < N; ++m ) sum += x[m]; // άθροισμα
```

Μπορούμε να κάνουμε τον ίδιο υπολογισμό με βέλη. Ας πούμε ότι έχουμε μια μεταβλητή-βέλος *p* προς μεταβλητή τύπου `double`. Μπορούμε να δώσουμε τιμή στην *p* με την εντολή: `p = x`. Με την `*p` παίρνουμε την τιμή του x[0]. Αυξάνοντας την τιμή της *p* κατά 1 (`p=p+1` ή `++p`) βάζουμε την *p* να δείχνει το x[1] και με αποπαραπομπή (`*p`) παίρνουμε την τιμή του. Δες το Σχ. 12-2. Για να πάρουμε λοιπόν το άθροισμα μπορούμε να γράψουμε:

```
      double sum( 0 );
      for ( double* p(x); p != &x[N]; ++p )  sum += *p;
```

Δηλαδή: Θέλω μια μεταβλητή *p* τέτοια ώστε το `*p` να είναι τύπου `double`. Αρχικώς θα δείχνει το στοιχείο `x[0]` του πίνακα x (`p(x)`). Στη συνέχεια θα πηγαίνει στο επόμενο στοιχείο (`++p`) και θα σταματήσει όταν βρεθεί να δείχνει την πρώτη θέση μετά το τελευταίο στοιχείο του πίνακα (`&x[N]`). Αυτή η δήλωση και η τελευταία **for** εξετάζονται πιο εκτεταμένα στην επόμενη υποπαράγραφο.



Σχ. 12-2 Ο πίνακας και τα βέλη. Τιμή του πίνακα (`x`) είναι η διεύθυνση του πρώτου στοιχείου (x[0]). Εδώ, βάζουμε ένα βέλος που δείχνει το x[0], αντί να βάλουμε ως τιμή της x την `0x0064fdbc` (ή κάτι παρόμοιο),. Μετά την εκτέλεση της εκχώρησης "`p = x`" την ίδια τιμή έχει και η *p*. Όταν αυξήσουμε την τιμή της *p* κατά 1 (`p=p+1`) η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[1]). Αν αυξήσουμε ξανά την τιμή της *p* κατά 1 η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[2]) κ.ο.κ..

Πολλά πράγματα μαζί! Ας τα δούμε ένα-ένα.

12.3.1 Μια Θέση Μετά το Τέλος

Πριν από οποιαδήποτε πράξη να πούμε το εξής: Στο παραπάνω παράδειγμα εμφανίστηκε ένα "`&x[N]`". Τι είναι αυτό; Τα στοιχεία του πίνακα καταλαμβάνουν τις θέσεις από 0 μέχρι *N*-1. Σωστό! Αλλά η C++ σου επιτρέπει να έχεις βέλος που να δείχνει στο `x[N]` (past the end pointer), αρκεί:

- Να μήν προσπαθήσεις να κάνεις αποπαραπομπή.
- Να το χρησιμοποιείς μόνο σε συγκρίσεις με άλλα βέλη (π.χ.: "`p != &x[N]`")

12.3.2 Αρχική Τιμή και Εκχώρηση

Ας ξεκινήσουμε από τον ορισμό αρχικής τιμής "`double* p(x)`". Σε μια μεταβλητή-βέλος μπορείς να δίνεις ως αρχική τιμή την τιμή μιας παράστασης-βέλος του ίδιου τύπου. Από όσα ξέρουμε μέχρι τώρα μπορείς να δώσεις:

- μια άλλη μεταβλητή-βέλος ίδιου τύπου· αν *p*1 τύπου `double*`: "`double* p(p1)`", οπότε το *p* δείχνει την ίδια θέση της μνήμης που δείχνει και το *p*1 (*p* == *p*1 ή `*`*p* και `*`*p*1 είναι το ίδιο αντικείμενο)
- το όνομα ενός πίνακα, όπως εδώ: "`double* p(x)` ", οπότε το *p* δείχνει το πρώτο στοιχείο του πίνακα (*p* == &x[0])
- τη διεύθυνση μιας μεταβλητής, π.χ.: "`double* p(&sum)`", οπότε το *p* δείχνει τη μεταβλητή *sum* (*p* == &*sum* ή `*`*p* είναι η *sum*).

  Όπως θα δούμε στη συνέχεια, μπορείς ακόμη να δώσεις:
- τιμή μηδέν (0): "`double* p( 0 )`",

- την τιμή αριθμητικής πράξης με βέλη, π.χ.: "`double* p(x+1)`".

Προσοχή! ▶

Σε ένα βέλος τύπου `const T*` μπορείς να βάλεις ως αρχική τιμή βέλος τύπου `T*`, π.χ.:

```
int a[17];
const int* pa( a );
```

Το αντίθετο δεν επιτρέπεται. Μπορείς να το πετύχεις με την κατάλληλη τυποθεώρηση, π.χ.:

```
unsigned int myStrLen( const char* cs )
{
   char* p( const_cast<char*>(cs) );
// . . .
```

(από ένα παράδειγμα που θα δεις στη συνέχεια.) ◀

Όταν δηλώνεις ένα βέλος, χωρίς να του δώσεις αρχική τιμή αυτό δείχνει σε κάποια τυχαία θέση μέσα στη μνήμη. Αν προσπαθήσεις να το χρησιμοποιήσεις μπορεί να κάνεις και κάποια ζημιά. Γι' αυτό οι προγραμματιστές συνηθίζουν να δίνουν αρχική τιμή στα βέλη, με τη δήλωσή τους. Αν δεν ξέρουν τι τιμή να δώσουν δίνουν την τιμή μηδέν ("`0`")[2] που:

- είναι συμβατή με οποιονδήποτε τύπο-βέλους,
- σημαίνει (κατά κοινή συμφωνία): *αυτό το βέλος δεν δείχνει κάτι* (που μας ενδιαφέρει) και
- είναι τιμή που μπορεί να ελεγχθεί (`if (p == 0)...`), όπως θα δούμε στη συνέχεια.

Με τους ίδιους τρόπους μπορείς να αλλάζεις, στη συνέχεια, την τιμή ενός βέλους με μια εντολή εκχώρησης. Στο δεξιό μερος μπορεί να υπάρχει "`0`" ή παράσταση που θα μας δώσει τιμή-βέλος ίδιου τύπου. Για παράδειγμα:

```
p = p1;
p = x;
p = &sum;
p = 0;
p = x + 1;
```

Θυμίσου, για παράδειγμα, ότι στην §12.2, αποσυνδέσαμε το βέλος *xf* από τον πίνακα x βάζοντας "`xf = 0`".

Για τον περιορισμό "`const`", ισχύει και για την εκχώρηση ο περιορισμός που είπαμε πιο πάνω για την απόδοση αρχικής τιμής.

Ένα σημείο που χρειάζεται προσοχή είναι το εξής: Ας πούμε ότι έχουμε δηλώσει:

```
int a( 10 ), b( 20 );
int* pa;
int* pb;
```

και στη συνέχεια δίνουμε:

```
pa = &a;  pb = &b;
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 12-3(α): το βέλος *pa* δείχνει την *a*, που έχει τιμή 10, και το βέλος *pb* δείχνει την *b* που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή "`a = b`". Η εικόνα της μνήμης είναι αυτή του Σχ. 12-3 (β). Η τιμή της *a* γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

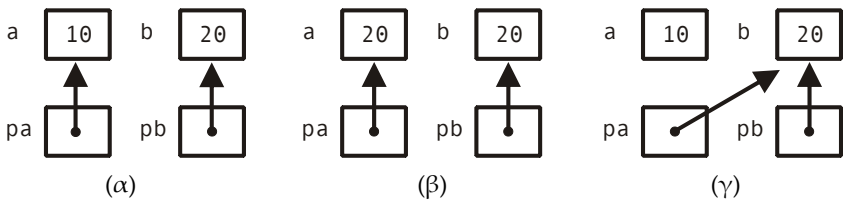
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: "`pa = pb`". Οι τιμές των μεταβλητών *a* και *b* δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *pa*: το βέλος *pa* δείχνει εκεί που δείχνει και το *pb*: τη *b* (Σχ. 12-3 (γ)).

Φυσικά, και στις δύο περιπτώσεις, αν δώσουμε την εντολή:

```
cout << *pa << "  " << *pb << endl;
```

---

[2] Οι προερχόμενοι από τη C αντί "`0`" (μηδέν) βάζουν τη σταθερά "`NULL`" που ορίζεται με τη μάκρο "`#define NULL 0`". Ο B. Stroustrup συμβουλεύει να προτιμούμε το "`0`". Πάντως στη νέα τυποποίηση C++11 εισάγεται το όνομα (*std*`::`)"*nullptr*".



Σχ. 12-3 Αρχικώς, μετά την εκτέλεση των εντολών "`pa = &a; pb = &b;`" έχουμε την εικόνα (α). Η εικόνα (β) προκύπτει αν στη συνέχεια εκτελεσθεί η εκχώρηση: "`a = b`". Η εικόνα (γ) προκύπτει αν, μετά την (α), εκτελεσθεί η εκχώρηση "`pa = pb`".

θα πάρουμε αποτέλεσμα:

```
20  20
```

Υπάρχει και ένας περιορισμός σχετικά με την εκχώρηση: *η μεταβλητή και η παράσταση θα πρέπει να είναι του ίδιου τύπου*. Έτσι, αν έχουμε δηλώσει:

```
double v;
```

δεν επιτρέπεται να γράψουμε[3]: "`pa = &v`".

Προσοχή! ▶

Πριν προχωρήσουμε να επισημάνουμε κάτι που μπορεί να σε οδηγήσει σε «παράξενα» λάθη μεταγλώττισης. Αν θελήσεις να μαζέψεις τις δύο δηλώσεις:

```
int* pa;
int* pb;
```

σε μία, θα πρέπει να γράψεις:

```
int *pa, *pb;
```

Αν γράψεις κατά λάθος:

```
int* pa, pb;
```

θα σημαίνει ότι η *pa* είναι βέλος προς `int` αλλά η *pb* είναι `int`! ◀

12.3.3 Πρόσθεση και Αφαίρεση

Αν έχεις δηλώσει:

```
const unsigned int N = ...;
T x[N];
```

ορίζονται οι πράξεις: `+`: *T*`*`×`int` ↠ *T*`*` και `-`: *T*`*`×`int` ↠ *T*`*` ή, ακριβέστερα:

`+`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`+`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

Πιό πάνω είδαμε ότι αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] `..` x[*N*`-1`], τότε *το* `p+1` *δείχνει το επόμενο στοιχείο*. Αντιστοίχως, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[1] .. x[*N*], τότε *το "*`p-1`*" δείχνει το προηγούμενο στοιχείο*.

Γενικώς, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] .. x[*N*`-1`], τότε

- *Το "*`p + k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***μετά*** *το στοιχείο που δείχνει το p*.
- *Το "*`p - k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***πριν*** *το στοιχείο που δείχνει το p*.

[3] Αν επιμένεις πολύ γίνεται, με την κατάλληλη τυποθεώρηση· αλλά, πού θα σου χρησιμεύσει κάτι τέτοιο (εκτός από το "σκαλίζεις" τη μνήμη);

Προσοχή! ▶

1. Δεν ειναι τυχαίες οι συνεχείς αναφορές σε πίνακες. *Οι πράξεις αυτές ορίζονται μόνο για βέλη προς στοιχεία ενός πίνακα.*

2. Παρ' όλο που λέμε ότι «τιμή του βέλους *p* είναι μια διεύθυνση της μνήμης» *το* "`p + 1`" ***δεν*** *είναι η επόμενη διεύθυνση* (π.χ. η διεύθυνση της επόμενης ψηφιολέξης). Είναι η διεύθυνση του επόμενου στοιχείου του πίνακα.◀

Γιατί γράψαμε τις συναρτήσεις μας μερικές; Διότι, για παράδειγμα, αν το *p* δείχνει το x[2] απαγορεύονται πράξεις σαν τις `p-6` ή `p+(-5)`.

- ***Είναι υποχρέωση του προγραμματιστή να φροντίσει ώστε το αποτέλεσμα πράξης βελών να είναι βέλος προς στοιχείο πίνακα ή προς την πρώτη θέση μετά το τελευταίο στοιχείο.***

Οι συντομογραφίες "`++`", "`--`", "`+=`", "`-=`" ισχύουν και έχουν το νόημα που ξέρεις προσαρμοσμένο στα παραπάνω. Έτσι, αν το *p* δείχνει το x[2] το "`++p`" αλλάζει την τιμή του *p* ώστε να δείχνει το x[3] (ενώ αντιθέτως το "`--p`" θα το πήγαινε στο x[1]). Αν βάλεις "`p += 2`", το *p* θα πάει από το x[2] στο x[4].

Υπάρχει μια ακόμη συνάρτηση (πράξη)· είναι μεταξύ βελών και είναι ολική:

`-`: [`&x[0] .. &x[`*N*`]`] × [`&x[0] .. &x[`*N*`]`] → [`-`*N* `..` *N*]

Αν δύο βέλη, `p1`, `p2`, δείχνουν στοιχεία του ίδιου πίνακα, ας πούμε τα `x[k1]` και `x[k2]`, έχει νόημα η διαφορά "`p1-p2`", που δείχνει τον αριθμό των στοιχείων που πρέπει να διανύσουμε για να φτάσουμε από το ένα στοιχείο στο άλλο· στην περίπτωσή μας "`k1-k2`".

Ας δούμε δύο παραδείγματα που χρησιμοποιούν πράξεις μεταξύ βελών.

Παράδειγμα 1 – Μήκος ορμαθού C

Όπως μάθαμε στο Κεφ. 10 (§10.13), στη C παριστάνουμε τα κείμενα σε πίνακες με στοιχεία τύπου `char` που έχουν στο τέλος (ως φρουρό) τον χαρακτήρα `'\0'`. Μια από τις συναρτήσεις που μας δίνει η C για τον χειρισμό τέτοιων πινάκων είναι η *strlen()* που μας δίνει το μήκος του κειμένου. Εδώ θα γράψουμε μια τέτοια συνάρτηση, ας την πούμε *myStrLen()*.

Αφού παρατηρήσουμε ότι αν *l* το μήκος ενός κειμένου αποθηκευμένου στον πίνακα cs τοτε ο φρουρός βρίσκεται στο στοιχείο cs`[`*l*`]`, γράφουμε τη συνάρτησή μας ως εξής:

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l] != '\0' )  ++l;
   return l;
} // myStrLen
```

Γράφουμε τώρα το ίδιο πράγμα με βέλη, για να πάρουμε μια πιο γρήγορη συνάρτηση:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *p != '\0' )  ++p;
   return p-cs;
} // myStrLen
```

Το *p* θα διατρέξει ολόκληρον τον πίνακα ξεκινώντας από το `cs[0]` (διεύθυνση `&cs[0]` ή απλώς `cs`). Ναι, αλλά πώς θα γίνει αυτό; Έχουμε βάλει "`const char* p(cs)`"! Αν έχεις αυτήν την απορία γύρισε πίσω, στην §12.2. Αυτό που λέμε εδώ είναι ότι *δεν επιτρέπεται μεταβολή αυτού που δείχνει το βέλος· μεταβολή του βέλους επιτρέπεται.*

Αν όμως σε ενοχλεί το "`const`" πρόσεξε τα εξής: Ο μεταγλωττιστής δεν θα επιτρέψει να γράψουμε `char* p(cs)` αφού το cs είναι "`const char`". Η τυποθεώρηση **const** είναι απαραίτητη:

```
   char* p( const_cast<char*>(cs) );
```

Ο *p* προχωρεί με τη `++p` όσο αυτό που δείχνει δεν έχει τον φρουρό (`*p != '\0'`).

Τελικώς, η συνάρτηση επιστρέφει τη διαφορά του βέλους προς τον φρουρό από το βέλος προς την αρχή.

Ένας προγραμματιστής C θα απορούσε ήδη με το ότι στην πρώτη μορφή επιμένουμε να γράφουμε πολλά αντί να χρησιμοποιήσουμε τον μεταθεματικό "`++`":

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l++] != '\0' );
   return l-1;
} // myStrLen
```

Πρόσεξε ότι εδώ παίρνουμε την τιμή του *l* για να υπολογίσουμε το στοιχείο cs`[`*l*`]` και αυξάνουμε την τιμή του *l*. Έτσι, αφ'ενός δεν έχουμε (άλλη) επαναλαμβανόμενη εντολή στη **while**, αφ' ετέρου η τιμή του *l* θα αυξηθεί και στην τελευταία επανάληψη, όταν βρούμε τον φρουρό. Γι' αυτό και επιστρέφουμε *l*-1.

Να το γράψουμε με βέλη; Νάτο:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) != '\0' );
   return p-cs-1;
} // myStrLen
```

Πρόσεξε ότι χρησιμοποιούμε μεταθεματικό "`++`" στο βέλος. Φυσικά, θα πρέπει και εδώ να αφαιρέσουμε 1.

Και βέβαια, ένας προγραμματιστής C δεν θα έλεγχε ποτέ για μηδέν με αυτόν τον τρόπο· θα το έγραφε πολύ πιο απλά:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) );
   return p-cs-1;
} // myStrLen
```

🖘🖘🖘

Παράδειγμα 2 – Επισύναψη ορμαθού C 🖙

Στην §10.13 λέγαμε και για τη *strcat*: «*Η* `strcat(a, b)` *κάνει το ίδιο πράγμα με την* `a.append(b)`*: επισυνάπτει στο τέλος της τιμής του a την τιμή του b.*» Η επικεφαλίδα της είναι:

```
char* strcat( char* dest, const char* src )
```

Η *strcat*, αφού επισυνάψει στο τέλος του *dest* ένα αντίγραφο του *src*, επιστρέφει ως τιμή βέλος προς την αρχή του *dest*.

Ας υλοποιήσουμε μια τέτοια *myStrCat* με πίνακες και δείκτες· χωρίς βέλη:

```
char* myStrCat( char* dest, const char* src )
{
   int j( 0 );
   while ( dest[j] != '\0' ) ++j;
   int k(0);  // εδώ έχουμε: dest[j] == '\0'
   while ( src[k] != '\0' )
   {
      dest[j] = src[k];
      ++j;  ++k;
   } // for
   dest[j] = '\0';
   return dest;
} // myStrCat
```

Πρόσεξε ότι δεν έχουμε βάλει οποιονδήποτε έλεγχο. Σκέψου, ως άσκηση, τους ελέγχους που θα πρέπει να βάλουμε.

Η ίδια συνάρτηση με βέλη μπορεί να γραφεί ως εξής:

```
char* myStrCat( char* dest, const char* src )
{
   char* pd( dest );
   while ( *(pd++) != '\0' );
   --pd;       // εδώ έχουμε το pd να δείχνει τον φρουρό
   const char* ps( src );
   while ( *ps != '\0' )
   {
      *pd = *ps;
      ++pd;  ++ps;
   }
   *pd = '\0';
   return dest;
} // myStrCat
```

Γιατί "`--pd`"; Ξαναδιάβασε το προηγούμενο παράδειγμα...

12.3.4 Ο Τύπος "ptrdiff_t"

Στο Παράδ. 1 της προηγούμενης παραγράφου, σε συμμόρφωση με τη φιλοσοφία της C++, βάλαμε ως τύπο επιστροφής "`size_t`".

Εδώ θα πρέπει να αναφέρουμε ότι το αποτέλεσμα της πράξης

`-`: [`&x[0] .. &x[`N`]`] × [`&x[0] .. &x[`N`]`] → [`-`N `..` N]

είναι τύπου "`std::ptrdiff_t`" (*poin*ter *diff*erence) . Αν ψάξεις στο `cstddef` (ή το `stddef.h`) θα βρεις:

```
typedef int     ptrdiff_t;
```

ή κάτι παρόμοιο.

Όταν γράφουμε "`return p-cs`" ζητούμε να επιστραφεί μια τιμή τύπου `ptrdiff_t`. Επειδή όμως αυτή είναι σίγουρα μη αρνητική μετατρέπεται σε τύπο `size_t`.

12.3.5 Συγκρίσεις

Ένα άλλο πράγμα που μπορείς να κάνεις με τα βέλη είναι η σύγκριση για ισότητα ("`==`") ή για ανισότητα ("`!=`"). Έτσι, βλέπεις στη **for** της άθροισης τη σύγκριση: "`p != x+N`" ή "`p != &x[N]`".

Μπορείς να συγκρίνεις δύο βέλη του ίδιου τύπου. Αργότερα θα δούμε ότι μερικές φορές χρειάζεται να συγκρίνουμε και βέλη διαφορετικού τύπου. Αυτό γίνεται με την κατάλληλη τυποθεώρηση.

Παράδειγμα

Στη *vectorSum* δεν έχουμε βάλει ελέγχους. Έτσι, μπορεί να κληθεί με $n \le 0$, με *from* ή/και *upto* έξω από την περιοχή 0..n-1. Αυτά σου τα αφήνουμε ως άσκηση.

Εδώ θα δούμε ένα άλλο πρόβλημα: Όπως καταλαβαίνεις, η *vectorSum* μπορεί να κληθεί με πρώτη παράμετρο 0 (μηδέν), δηλαδή *χωρίς πίνακα*. Και αν μεν έχουμε και n `==` 0 μπορούμε να βγάλουμε μηδενικό άθροισμα. Αν όμως έχουμε n `>` 0 τότε έχουμε προφανώς λάθος. Να λοιπόν η συνάρτηση με αυτόν τον έλεγχο:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   if ( x == 0 && n > 0 )
   {  cerr << "η vectorSum κλήθηκε με ανύπαρκτο πίνακα" << endl;
      exit( EXIT_FAILURE );  }
// άλλοι έλεγχοι
   double sum( 0 );
   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

12.4 Πολυδιάστατοι Πίνακες

Ας πούμε ότι έχουμε ορίσει:

```
typedef int   intArr [ 51 ];
```

και δηλώνουμε:

```
   intArr    mat [ 11 ];
```

Τι είναι μια συνιστώσα του mat, π.χ. η `mat[7]`; Ένας *πίνακας* με 51 συνιστώσες τύπου `int`. Πώς θα αναφερθούμε στην 23η συνιστώσα αυτού του πίνακα; Αν θεωρήσουμε ότι έχουμε έναν μονοδιάστατο πίνακα με όνομα `mat[7]` το 23ο στοιχείο του θα είναι: `(mat[7])[23]`. Η C++ μας επιτρέπει να γράψουμε απλούστερα: `mat[7][23]`.

Η δήλωση του mat θα μπορούσε να δοθεί ισοδυνάμως και ως εξής:

```
   int   mat[ 11 ][ 51 ];
```

Ο mat είναι ένας **πίνακας δυο διαστάσεων** (two-dimensional array) και έχει 11×51 στοιχεία τύπου `int`.

Όταν δηλώνουμε έναν πίνακα, μονοδιάστατο ή πολυδιάστατο, ο τύπος συνιστωσών μπορεί να είναι οποιοσδήποτε. Έτσι, οι πίνακες pinA, pinB, pinC και pinD, που δηλώνονται στο παρακάτω παράδειγμα, είναι πίνακες μιας διάστασης, δύο, τριών και τεσσάρων διαστάσεων αντίστοιχα.

```
   int    pinA[ 10 ];
   char   pinB[ 10 ][ 30 ];
   double pinC[ 5 ][ 10 ][ 20 ];
   bool   pinD[ 5 ][ 10 ][ 20 ][ 8 ];
```

- Ο πίνακας pinA αποτελείται από 10 στοιχεία τύπου `int`,
- ο πίνακας pinB από 300 (= 10×30) στοιχεία τύπου `char`,
- ο πίνακας pinC από 1000 (= 5×10×20) στοιχεία τύπου `double` και τέλος
- ο πίνακας pinD από 8000 (= 5×10×20×8) στοιχεία τύπου `bool`.

Να μερικά παραδείγματα γραφής στοιχείων πολυδιάστατων πινάκων, σύμφωνα με τις παραπάνω δηλώσεις:

```
α) pinB[1][1], pinB[1][29], pinB[2][1], pinB[2][29],
   pinB[9][1], pinB[9][29]
β) pinC[1][1][1], pinC[5][1][1], pinC[1][9][1],
   pinC[1][1][19], pinC[5][9][19]
γ) pinD[1][1][1][1], pinD[1][1][1][2], pinD[4][9][19][7],
   pinD[4][9][19][8]
```

Με τη δήλωση μπορείς να δώσεις και αρχική τιμή. Αφού, όπως είπαμε, ένας δισδιάστατος πίνακας είναι (μονοδιάστατος) πίνακας μονοδιάστατων πινάκων θα πρέπει να μπορούμε να δώσουμε κάτι σαν:

```
   double x[4][2] = { {0.1, 1.2}, {2.3, 3.4}, {4.5, 5.6},
                      {6.7, 7.8} };
```

Και πράγματι, αυτό είναι σωστό! Και ποια είναι τα στοιχεία αυτού του μονοδιάστατου πίνακα; Τα `x[0]`,… `x[3]`. Οι εντολές:

```
   for ( int r(0); r < 4; ++r )
      cout << x[r] << "  " << &x[r][0] << endl;
```

θα μας δώσουν:

```
0x22ff30  0x22ff30
0x22ff40  0x22ff40
0x22ff50  0x22ff50
0x22ff60  0x22ff60
```

Δηλαδή, το `x[r]` είναι βέλος προς το `x[r][0]`.

Στη C++ τα στοιχεία ενός δισδιάστατου πίνακα *αποθηκεύονται κατά γραμμές*: πρώτα τα στοιχεία της γραμμής 0, στη συνέχεια τα στοιχεία της γραμμής 1 κλπ. Αυτό όμως δεν σημαίνει ότι έχουμε την υποχρέωση να επεξεργαζόμαστε τα στοιχεία με κάποια συγκεκριμένη σειρά. *Δεν υπάρχει κανένας περιορισμός σχετικά με τη σειρά επεξεργασίας των στοιχείων του πίνακα. Η σειρά αυτή καθορίζεται μόνο από τις ανάγκες και τη λογική του προγράμματος και επιλέγεται ελεύθερα από τον προγραμματιστή.*

Ας δούμε μερικά παραδείγματα:

Παράδειγμα 1 - Άθροισμα στοιχείων τρισδιάστατου πίνακα

Αν έχουμε δηλώσει:

```
double pinC[ 5 ][ 10 ][ 20 ];
```

οι τρεις παρακάτω φωλιασμένες **for** υπολογίζουν το άθροισμα, *sum*, των στοιχείων του pinC:

```
sum = 0;
for ( int k(0); k < 5; ++k ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int m(0); m < 20; ++m ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int m . . .
   } // for ( int l . . .
} // for ( int k . . .
```

Όπως καταλαβαίνεις, ο τρίτος δείκτης, δηλαδή ο *m*, είναι ο δείκτης που αλλάζει τιμή πιο συχνά από τους άλλους και αυτό επειδή βρίσκεται στην πιο εσωτερική **for**. Ο δείκτης *m* διατρέχει όλες τις τιμές του, δηλ.τις τιμές 0 μέχρι 19, για κάθε νέο συνδυασμό τιμών των δεικτών *k* και *l* (δηλ. για 50 συνδυασμούς). Έτσι, η εσωτερική **for** εκτελείται 1000 (=20×50) φορές. Αντίθετα, ο δείκτης *k* είναι ο δείκτης που *μόνο μια φορά* διατρέχει τις τιμές του (από 0 ως 4), ενώ ο δείκτης *l* διατρέχει 5 φορές τις τιμές του (από 0 μέχρι 9). Έτσι, η ενδιάμεση **for** εκτελείται 5 φορές.

Θα μπορούσαμε να υπολογίσουμε το άθροισμα και με τις:

```
sum = 0;
for ( int m(0); m < 20; ++m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(0); k < 5; ++k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
} // for ( int m . . .
```

Στη περίπτωση αυτή, ο πρώτος δείκτης, *k*, είναι αυτός που αλλάζει τιμή πιο συχνά και διατρέχει 200 φορές τις τιμές του, δηλ. τις τιμές 0 μέχρι 4. Ο δείκτης *m* διατρέχει μόνο μιά φορά τις τιμές του (0 ως 19) και ο *l* διατρέχει 20 φορές τις τιμές του (από 0 μέχρι 9).

Τέλος, δες και μια τρίτη περίπτωση:

```
sum = 0;
for ( int m(19); m >= 0; --m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(9); l >= 0; --l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(4); k >= 0; --k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
```

```
   } // for ( int m . . .
```

Στην περίπτωση αυτήν έχουμε το ίδιο φώλιασμα με τη δεύτερη, αλλά οι δείκτες διατρέχουν τις τιμές από το τέλος προς την αρχή.

Παράδειγμα 2 - Ανάγνωση τιμών στοιχείων δισδιάστατου πίνακα

Ας δούμε όμως και το εξής πρόβλημα: Σε ένα αρχείο, που το διαβάζουμε μέσω του ρεύματος *ins*, έχουμε τις τιμές των στοιχείων ενός δισδιάστατου πίνακα:

```
   double   a[ 5 ][ 7 ];
```

Οι τιμές είναι γραμμένες *κατά γραμμές*. Τι θα πει αυτό; Σε κάθε γραμμή του αρχείου υπάρχουν οι (επτά) τιμές μιας γραμμής του πίνακα. Στην 1η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 0 του πίνακα, στη 2η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 1 του πίνακα κ.ο.κ. Πώς διαβάζουμε τις τιμές των στοιχείων του πίνακα;

```
   for ( int r(0); r < 5; ++r )
   {
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][c];
      } // for ( int c . . .
   } // for ( int r . . .
```

Όπως καταλαβαίνεις για κάθε τιμή της *r* διαβάζουμε τις τιμές των στοιχείων της γραμμής *r* του πίνακα. Αυτό γίνεται με την:

```
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][s];
      } // for ( int s . . .
```

και φυσικά εδώ *δεν μπορούμε να αλλάξουμε τη σειρά* (φωλιάσματος) των δύο **for**.

Βέβαια, υπάρχει και η περίπτωση να δοθούν οι τιμές των στοιχείων *κατά στήλες*. Δηλαδή, σε κάθε γραμμή του αρχείου υπάρχουν οι (πέντε) τιμές μιας στήλης του πίνακα Στην περίπτωση αυτή είμαστε υποχρεωμένοι να διαβάσουμε ως εξής:

```
   for ( int c(0); c < 7; ++c )
   {
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
   } // for ( int c . . .
```

Τώρα, η

```
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
```

διαβάζει τη στήλη *c* του πίνακα.

Παρομοίως γίνεται και το γράψιμο των στοιχείων δισδιάστατου πίνακα κατά γραμμές ή κατά στήλες (άσκ. 8-4).

Παράδειγμα 3 - Συμμετρικότητα τετραγωνικού πίνακα

Ένας τετραγωνικός πίνακας –δηλαδή: δισδιάστατος πίνακας με ίσους αριθμούς γραμμών και στηλών– λέγεται *συμμετρικός* (ως προς την πρώτη διαγώνιο) αν για όλα τα στοιχεία του ισχύει η ιδιότητα $a_{rc} == a_{cr}$. Οι παρακάτω εντολές ελέγχουν αν ο

```
   double a[N][N];
```

(όπου *N* σταθερά τύπου `int` με θετική τιμή) είναι συμμετρικός:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
```

```
   {
      for ( int c(0); c < N; ++c )
      {
         if( a[r][c] != a[c][r] )  symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Πράγματι, κάνουμε την υπόθεση ότι ο a είναι συμμετρικός (`symmet = true`) και στη συνέχεια αν βρούμε κάποιο ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$ διορθώνουμε την τιμή της *symmet* σε `false`.

Αυτό το κομμάτι προγράμματος είναι σπάταλο από άποψη χρόνου επεξεργασίας διότι, πριν απ' όλα:

- Ελέγχει κάθε ζευγάρι δυο φορές, π.χ. ενώ θα ελέγξει αν $a_{23} \neq a_{32}$, αργότερα θα ελέγξει και αν $a_{32} \neq a_{23}$, πράγμα άχρηστο.
- Ελέγχει –και μάλιστα δυο φορές– αν κάθε στοιχείο της πρώτης διαγωνίου είναι ίσο με τον εαυτό του.

Αυτές οι ατέλειες διορθώνονται αν αρχίζουμε την εξέταση της κάθε γραμμής ένα στοιχείο μετά τη διαγώνιο. Αν πάρουμε υπόψη μας ότι το στοιχείο της διαγωνίου στη γραμμή *r* είναι το a_{rr}, θα πρέπει να γράψουμε:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Στη χειρότερη περίπτωση –όταν ο πίνακας *είναι* συμμετρικός– η `if( a[r][c] != a[c][r] )...` θα εκτελεσθεί $\frac{1}{2}N(N-1)^2$ φορές, ενώ στον πρώτο αλγόριθμο θα εκτελεσθεί N^2 φορές.

Υπάρχει όμως και άλλη σπατάλη: αν βρούμε *ένα* ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$, δεν έχει νόημα να συνεχίσουμε τις συγκρίσεις· *ξέρουμε ότι ο πίνακας δεν είναι συμμετρικός*.

Στο παρακάτω κομμάτι έχουμε διορθώσει και αυτήν την ατέλεια:

```
   symmet = true;
   for ( int r(0); symmet && r < N; ++r )
   {
      for ( int c(r+1); symmet && c < N; ++c )
      {
         if ( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Δηλαδή, έχουμε πάλι μετρούμενες επαναλήψεις με τη διαφορά ότι τώρα βάλαμε και πρόσθετους ελέγχους για το αν ανιχνεύθηκε παραβίαση της συμμετρίας, οπότε σταματούμε αμέσως.

Μπορούμε να αποφύγουμε τη διπλή συνθήκη στις **for** αν ξαναγράψουμε, χρησιμοποιώντας τη `break`:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if ( a[r][c] != a[c][r] )
         {  symmet = false;   break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
```

Αν εκτελεσθεί η πρώτη **break** θα διακόψει την εκτέλεση της εσωτερικής **for**. Για να διακόψουμε και την εκτέλεση της εξωτερικής χρειάζεται η δεύτερη **break** που εκτελείται όταν βρει τη *symmet* με τιμή **false**. Αυτή η μετατροπή επιταχύνει σημαντικά το πρόγραμμά μας αφού επιταχύνει την εσωτερική **for** με την απλούστευση της συνθήκης συνέχισης.

Πάντως χρησιμοποιώντας την «καταραμένη» εντολή **goto**, μπορούμε να το κάνουμε ακόμη ταχύτερο:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
      {
         for ( int c(r+1); c < N; ++c )
         {
            if ( a[r][c] != a[c][r] )
            {  symmet = false;  goto lb99;  }
         } // for ( int c . . .
      } // for ( int r . . .
lb99: .....
```

Αυτή είναι μια θεμιτή χρήση της **goto** και κανείς δεν μπορεί να ισχυρισθεί ότι οι άλλες γραφές είναι καλύτερες από αυτήν. Πάντως, πρόσεξε και μια λεπτομέρεια: η εντολή με την ετικέτα **lb99** θα πρέπει να βρίσκεται ακριβώς μετά το τέλος των **for** και όχι οπουδήποτε αλλού μέσα στο πρόγραμμά μας.

Το πρώτο κομμάτι προγράμματος, με τις δύο **for**, περνάει από *όλα* τα στοιχεία του πίνακα (όλοι οι δυνατοί συνδυασμοί των *r* και *c*). Το δεύτερο κομμάτι σαρώνει τα στοιχεία του πίνακα που βρίσκονται *πάνω από την κύρια διαγώνιο* του πίνακα. Τα κομμάτια προγράμματος τρίτο, τέταρτο και πέμπτο σαρώνουν τα στοιχεία του πίνακα που βρίσκονται πάνω από την κύρια διαγώνιο του πίνακα στη χειρότερη περίπτωση.

Παράδειγμα 4 - Πολλαπλασιασμός πινάκων

Έστω ότι ο δισδιάστατος πίνακας a αποτελείται από *l* γραμμές και *m* στήλες, ενώ ο επίσης δισδιάστατος πίνακας b αποτελείται από *m* γραμμές και *n* στήλες. Να υπολογιστεί το γινόμενό τους c, που είναι δισδιάστατος πίνακας *l* γραμμών και *n* στηλών.

Τα στοιχείο του πίνακα c, στη γραμμή *r* και τη στήλη *c* (c_{rc}), υπολογίζεται, όπως γνωρίζουμε από τα μαθηματικά, με τον παρακάτω τύπο:

$$c_{rc} = \sum_{k=0}^{m-1} a_{rk} b_{kc}$$

όπου: $r = 0..l\text{-}1$, $c = 0..n\text{-}1$.

Δηλαδή, το στοιχείο c_{rc} είναι το άθροισμα όλων των όρων $a_{rk}b_{kc}$, όπου ο δείκτης *k* διατρέχει τις τιμές $0..m\text{-}1$.

Αυτός ο υπολογισμός μπορεί να δοθεί στη C++ με μιά **for** και μια εντολή εκχώρησης:

```
      c[r][c] = 0;
      for ( int k(0); k < m; ++k )
         c[r][c] += a[r][k]*b[k][c];
```

Το παρακάτω πρόγραμμα διαβάζει τα στοιχεία των πινάκων a και b, υπολογίζει το γινόμενό τους, δηλ. βρίσκει τα στοιχεία του πίνακα c και μετά τυπώνει τα στοιχεία του γινομένου c (καθώς και των πινάκων a και b), όπως βλέπεις στη συνέχεια:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   const int l = 3,  m = 5,  n = 2;

   int a[ l ][ m ],  b[ m ][ n ];
   int c[ l ][ n ];
```

```
   ifstream atx( "arr.txt" );
// Διάβασε τα στοιχεία του a
   for ( int r(0); r < l; ++r )
      for ( int c(0); c < m; ++c )  atx >> a[r][c];
// Διάβασε τα στοιχεία του b
   for ( int r(0); r < m; ++r )
      for ( int c(0); c < n; ++c )  atx >> b[r][c];
   atx.close();

// Πολλαπλασιασμός Πινάκων
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         c[r][c] = 0;
         for ( int k(0); k < m; ++k )
            c[r][c] += a[r][k]*b[k][c];
      } // for (c . . .
   } // for (r . . .

// Γράψε τα στοιχεία των a, b, c
   cout << " Στοιχεία του πίνακα a" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < m; ++c )
      {
         cout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα b" << endl;
   for ( int r(0); r < m; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << b[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα c" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << c[r][c] << "  ";
      }
      cout << endl;
   }
}
```

Για να δοκιμάσουμε το πρόγραμμα γράφουμε με τον κειμενογράφο το παρακάτω αρχείο:

```
1    2    3    4    5
6    7    8    9    1
1    4    7    2    5
10   10
12   45
47   18
12   31
18   29
```

και το φυλάγουμε με όνομα "arr.txt" (σε μορφή text). Όταν εκτελεσθεί το πρόγραμμα μας δίνει:

```
 Στοιχεία του πίνακα a
  1    2    3    4    5
  6    7    8    9    1
  1    4    7    2    5
```

```
 Στοιχεία του πίνακα b
 10   10
 12   45
 47   18
 12   31
 18   29
 Στοιχεία του πίνακα c
313  423
646  827
501  523
```

12.5 Η Σειρά Αποθήκευσης

Συνήθως, η σειρά αποθήκευσης των στοιχείων ενός πολυδιάστατου πίνακα θα σου είναι αδιάφορη. Υπάρχει όμως μια τουλάχιστον περίπτωση, που θα δούμε στην επόμενη παράγραφο, όπου σου χρειάζεται. Ας την δούμε λοιπόν.

Όπως αναφέραμε προηγουμένως, «τα στοιχεία ενός δισδιάστατου πίνακα *απόθηκεύονται κατά γραμμές*». Ας δούμε τι σημαίνει αυτό πιο συγκεκριμένα.

Έστω ότι έχουμε δηλώσει:

```
double a[5][7];
```

Η αποθήκευση των στοιχείων ξεκινάει από το a[0][0]. Η επόμενη θέση είναι για το a[0][1], η μεθεπόμενη για a[0][2] κ.ο.κ. Έξη θέσεις μετά το a[0][0] υπάρχει το τελευταίο στοιχείο της γραμμής 0, το a[0][6]. Μετά από αυτό έχουμε τη θέση για το a[1][0], το πρώτο στοιχείο της γραμμής 1. Όπως καταλαβαίνεις, το στοιχείο a[*r*][*c*] βρίσκεται 7*r* + *c* θέσεις μετά το a[0][0].

Προσοχή: ▶
Όταν λέμε «θέση», εννοούμε με την έννοια που είδαμε στην §12.3.2, δηλαδή θέση μνήμης που μπορεί να αποθηκεύσει ένα στοιχείο του πίνακα, στην περίπτωσή μας, μια τιμή τύπου `double` (συχνότατα αυτό σημαίνει 8 ψηφιολέξεις).◀

Ας κάνουμε ένα πείραμα για να επιβεβαιώσουμε τον παραπάνω τύπο. Δίνουμε τιμές στα στοιχεία του a ως εξής:

```
for ( int r(0); r <= 4; ++r )
{
   for ( int c(0); c <= 6; ++c )
   {
      a[r][c] = r + 0.1*c;
   } // for ( int c . . .
} // for ( int r . . .
```

Έτσι, τα στοιχεία της τελευταίας γραμμής (γραμμή 4) παίρνουν τιμές: 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6.

Τώρα, θα προσπαθήσουμε να δούμε τον a ως μονοδιάστατο. Δηλώνουμε:

```
double* p;
```

και βάζουμε:

```
p = &a[0][0];
```

Αφού η C++ καταλαβαίνει τα βέλη και τους πίνακες με τον ίδιο τρόπο, μπορούμε να δούμε το *p* ως μονοδιάστατο πίνακα, με στοιχεία τύπου `double`, που ξεκινάει από το a[0][0]. Από εκεί όμως ξεκινάει η αποθήκευση του a! Με το *p* λοιπόν μπορούμε να χειριστούμε τον a ως μονοδιάστατο πίνακα.

Σύμφωνα με τον τύπο που δώσαμε παραπάνω, τα στοιχεία της γραμμής 4 του a βρίσκονται στις θέσεις από 7×4 + 0 μέχρι 7×4 + 6 μετά το a[0][0]. Ζητούμε λοιπόν και μεις να δούμε τις τιμές αυτών των στοιχείων του πίνακα *p*:

```
r = 4;
for ( int c(0); c <= 6; ++c )
{
```

```
      cout << p[r*7+c] << "  ";
   }
   cout << endl;
```

Αποτέλεσμα; Αυτό ακριβώς που περιμένουμε:

```
4  4.1  4.2  4.3  4.4  4.5  4.6
```

Αν λοιπόν έχεις δηλώσει:

```
   T a[Nr][Nc];
```

τότε:

- ***το στοιχείο* a[*r*][*c*] *βρίσκεται* $r \cdot N_c + c$ *θέσεις (τύπου T) μετά το* a[0][0].**

12.5.1 Τρισδιάστατοι και Πολυδιάστατοι Πίνακες

Γενικότερα, αν έχεις δηλώσει:

```
   T a[N1][N2]...[Nm];
```

τότε:

- ***το στοιχείο* a[k_1][k_2]...[k_m] *βρίσκεται***

  (...($k_1 \times N_2 + k_2$)$\times N_3$ + ... + k_{m-1})$\times N_m + k_m$

  ***θέσεις (τύπου T) μετά το* a[0][0]...[0].**

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  a[2][2][2];
   int* ip;

   for ( int l(0); l < 2; ++l )
      for ( int r(0); r < 2; ++r )
         for ( int c(0); c < 2; ++c )
            a[l][r][c] = 100*(l+1) + 10*(r+1) + c+1;
   ip = &a[0][0][0];
   for ( int k(0); k < 8; ++k )
      cout << ip[k] << "  ";
   cout << endl;
}
```

Αποτέλεσμα:

```
111  112  121  122  211  212  221  222
```

Στα πρώτα τέσσερα στοιχεία το πρώτο ψηφίο, που βγαίνει από την τιμή της l(+1), έχει τιμή 1. Αυτά είναι τα στοιχεία που βρίσκονται στο «επίπεδο» $l = 0$ ενώ τα τέσσερα τελευταία βρίσκονται στο επίπεδο $l = 1$. Τα δύο πρώτα στοιχεία βρίσκονται στη γραμμή $r = 0$ του επιπέδου $l = 0$.

Το στοιχείο `a[1][0][1]`, που έχει τιμή 212, το βλέπουμε ως ip[((1×2 + 0)×2 + 1], δηλαδή ip[5].

Και το `a[1][0]` τι είναι; Τι είναι στην περίπτωση αυτήν το `a[1]`; Αντί για άλλη απάντηση δοκίμασε τις εντολές:

```
   for ( int l(0); l < 2; ++l )
   {
      cout << a[l] << "  " << &a[l][0][0] << endl;
      for ( int r(0); r < 2; ++r )
         cout  << "   " << a[l][r] << "  " << &a[l][r][0] << endl;
   }
```

που θα δώσουν:

```
0x22ff50  0x22ff50
   0x22ff50  0x22ff50
```

```
   0x22ff58  0x22ff58
0x22ff60  0x22ff60
   0x22ff60  0x22ff60
   0x22ff68  0x22ff68
```

Δηλαδή: το `a[l]` είναι βέλος προς το στοιχείο `a[l][0][0]` ενώ το `a[l][r]` είναι βέλος προς το `a[l][r][0]`.

12.6 Παράμετρος Πίνακας (ξανά)

Ας πούμε ότι έχουμε ορίσει έναν τύπο:

```
typedef char   ProSthlh[13];
```

με στόχο να παραστήσουμε σε στοιχεία αυτού του τύπου στήλες ΠροΠο. Αν δηλώσουμε π.χ.:

```
   ProSthlh c;
```

μπορούμε να δώσουμε:

```
   c[0] = '1';  c[1] = '1';  c[2] = 'X';...
   c[11] = 'X';  c[12] = '2';
```

Ύστερα από αυτό, μπορούμε να παραστήσουμε ένα δελτίο ΠροΠο, με *N* στήλες, με έναν πίνακα:

```
   ProSthlh deltio[N];
```

Θέλουμε μια συνάρτηση που θα παίρνει ένα δελτίο ΠροΠο και τη νικήτρια στήλη και θα μας επιστρέφει, ως τιμή, το πλήθος των στηλών του δελτίου που κερδίζουν (συμφωνούν με τη νικήτρια σε 11 σημεία τουλάχιστον).

Μπορούμε να γράψουμε το εξής σχέδιο:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
      μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      if ( ne >= 11 )  ++es;
   } // for
   return es;
} // winners
```

Η μέτρηση του πλήθους *ne* σωστών σημείων της στήλης deltio[*k*] γίνεται ως εξής:

```
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
```

Και να ολόκληρη η συνάρτηση:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
      {
         if ( deltio[k][j] == nik[j] )  ++ne;
      } // for ( int j . . .
      if ( ne >= 11 )  ++es;
   } // for ( int k . . .
   return es;
```

```
} // winners
```

Πώς την καλούμε; Έστω ότι έχουμε δηλώσει στη **main**:

```
   ProSthlh c, d[138];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners(d, 138, c) << endl;
```

Παρά την προσπάθεια που κάναμε (;) για να κρύψουμε την αλήθεια, αυτή δεν κρύβεται: στη μέτρηση των σωστών σημείων βλέπουμε το δελτίο ως *δισδιάστατο πίνακα*. Να αφήσουμε κατά μέρος τον τύπο ProSthlh και να πούμε τα πράγματα με το όνομά τους;

```
int winners2( char deltio[][13], int N, char nik[] )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
      if (ne >= 11)  ++es;
   } // for ( int k...
   return es;
} // winners2
```

Εδώ βλέπεις πώς περνούμε μια παράμετρο-δισδιάστατο πίνακα: το πλήθος των στηλών (13) γράφεται, το πλήθος των γραμμών όχι απαραίτητα. Αν θέλεις να περάσεις έναν πολυδιάστατο πίνακα θα πρέπει να καθορίζεις όλες τις διαστάσεις εκτός από την πρώτη.

Αν έχουμε δηλώσει στη **main**:

```
   char c[13], d[138][13];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners2(d, 138, c) << endl;
```

Μέχρι εδώ καλά. Δες τώρα ένα άλλο πρόβλημα.

Ας πούμε τώρα ότι θέλουμε να περάσουμε έναν δισδιάστατο πίνακα χωρίς να έχουμε καθορισμένο πλήθος στηλών· θέλουμε να περάσουμε τα πλήθη γραμμών και στηλών ως παραμέτρους. Πώς μπορεί να γίνει αυτό; Με βάση αυτά που είπαμε στην προηγούμενη παράγραφο. Δηλαδή *περνούμε τον πίνακα ως μονοδιάστατο*. Ας το δούμε με ένα παράδειγμα.

Παράδειγμα

Θέλουμε μια συνάρτηση που θα παίρνει έναν τετραγωνικό πίνακα a με στοιχεία τύπου `double` και θα μας επιστρέφει τιμή `true` αν ο a είναι συμμετρικός και `false` αν δεν είναι.

Η ιδέα είναι η εξής: Όταν καλούμε τη συνάρτηση θα της δίνουμε τη διεύθυνση του πρώτου στοιχείου του πίνακα και πλήθος *N* γραμμών και στηλών. Μέσα στη συνάρτηση δεν θα γράφουμε `a[r][c]` αλλά `a[r*n+c]`.

```
bool isSymmetric( const double* a, int n ) // double a[n][n]
{
   bool symmet( true );

   for ( int r(0); r < n; ++r )
   {
      for ( int c(r+1); c < n; ++c )
      {
//       if ( a[r][c] != a[c][r] ) {  symmet = false;  break;  }
         if ( a[r*n+c] != a[c*n+r] ) {  symmet = false;  break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
   return symmet;
} // isSymmetric
```

Ας δούμε τώρα πώς την καλούμε. Έστω ότι έχουμε:

```
double p1[2][2] = { {1,2}, {3,4} },
       p2[3][3] = { {1, 0, 1.5}, {0, 1.8, 2}, {1.5, 2, 4.1} };
```

Αν θέλουμε να ελέγξουμε τη συμμετρικότητα των p1 και p2 με τη συνάρτησή μας γράφουμε:

```
if ( isSymmetric(&p1[0][0], 2) )  cout << " ο p1 ναι" << endl;
                            else  cout << " ο p1 όχι" << endl;
if ( isSymmetric(&p2[0][0], 3) )  cout << " ο p2 ναι" << endl;
                            else  cout << " ο p2 όχι" << endl;
```

€€€

Όπως βλέπεις –και στην περίπτωση αυτήν– χρησιμοποιούμε αυτά που ξέρουμε για την εσωτερική παράσταση για να περάσουμε ως παράμετρο και να χειριστούμε έναν δισδιάστατο πίνακα. Αυτό είναι μάλλον ένα τέχνασμα αλλά είναι ο μόνος τρόπος που έχουμε.[4]

12.6.1 Και Άλλα Τεχνάσματα

Στην Άσκ. 9-6 ζητούμε μια «*συνάρτηση rawSum() που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο r1 και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής r1.*»

Όποιος έχει διαβάσει με προσοχή αυτά που είδαμε μέχρι τώρα θα πεί: «Δεν χρειάζεται, έχουμε τη *vectorSum*!» Πράγματι, αν

```
double a[5][7], z;
```

και θέλουμε το άθροισμα των στοιχείων της γραμμής 2 μπορούμε να γράψουμε:

```
z = vectorSum( a[2], 7, 0, 6 );
```

ή

```
z = vectorSum( &a[2][0], 7, 0, 6 );
```

Εδώ στηριζόμαστε στο πώς αποθηκεύει η C++/C τους δισδιάστατους πίνακες και θεωρείται τέχνασμα. Λύσε λοιπόν την Άσκ. 9-6 με πιο «ορθόδοξο» τρόπο.

Ας δούμε όμως και μια περίπτωση που τα τεχνάσματα δεν δουλεύουν. Έχουμε τον πίνακα:

```
double p3[6][9] = { {1, 2, 4, 0, 5, 2, 7, 9, 0},
                    {2, 2, 0, 3, 4, 5, 9, 7, 8},
                    {4, 0, 2, 6, 3, 4, 0, 5, 7},
                    {3, 4, 8, 3, 5, 7, 0, 4, 1},
                    {4, 5, 1, 4, 7, 9, 0, 5, 7},
                    {1, 1, 5, 0, 0, 0, 8, 8, 6} };
```

και θέλουμε να ελέγξουμε αν οι (τετραγωγικοί) υποπίνακες από `p3[0][0]` μέχρι `p3[2][2]` και από `p3[2][3]` μέχρι `p3[5][6]` είναι συμμετρικοί. Θα μπορούσαμε να ρωτήσουμε *isSymmetric*`(&p3[0][0], 3)` και *isSymmetric*`(&p3[2][3], 4)`; Ούτε για αστείο!

Αν μας ενδιαφέρουν τέτοιες επεξεργασίες ξαναγράφουμε την *isSymmetric* ώστε να τηρούμε αυτό που είπαμε και πιο πριν:

- ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Στο παράδειγμά μας η *isSymmetric* θα πρέπει να ξαναγραφεί ως εξής:

```
bool isPartSymmetric( const double* a, int nRow, int nCol,
                      int rUL, int cUL, int rLR, int cLR )
```

Οι παράμετροι της πρώτης γραμμής περνούν στη συνάρτηση τον πίνακα: αρχή, γραμμές, στήλες. Οι παράμετροι της δεύτερης γραμμής περνούν την περιοχή επεξεργασίας που τη θεωρούμε ως ορθογώνιο και περνούμε: τη γραμμή (*rUL*) και τη στήλη (*cUL*) πάνω αριστε-

[4] Αργότερα θα μάθουμε τους δυναμικούς πίνακες που έχουν πιο «ορθόδοξο» χειρισμό.

ρής κορυφής και τη γραμμή (*rLR*) και τη στήλη (*cLR*) κάτω δεξιάς κορυφής. Για τα παραδείγματα που είπαμε πιο πάνω θα καλούμε ως εξής:

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          0, 0, 2, 2 ) . . .
```

και

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          2, 3, 5, 6 ) . . .
```

12.7 Οι Παράμετροι της `main`

Ας πούμε ότι έχουμε ένα πρόγραμμα φυλαγμένο στο αρχείο mainargs.exe. Για να ζητησουμε την εκτέλεσή του από γραμμή εντολών δίνουμε:

```
D:\>mainargs<enter>
```

Όπως θα έχεις δει όμως, μερικές φορές εκτός από το όνομα του αρχείου γράφουμε και διάφορες παραμέτρους, οι οποίες περνούν στο πρόγραμμα και επηρεάζουν την εκτέλεσή του. Μπορούμε να γράψουμε τέτοια προγράμματα; Ναι! Ας πούμε ότι το αρχικό πρόγραμμα αυτού που έχουμε στο mainargs.exe είναι το παρακάτω:

```
#include <iostream>
using namespace std;
int main( int argc, char* argv[] )
{
   cout << "Η τιμή της argc είναι " << argc << endl;
   cout << "Αυτά είναι τα ορίσματα που πέρασαν στην int main:"
        << endl;

   for ( int k(0); k <= argc; ++k )
      cout << "   argv[" << k << "]: " << argv[k] << endl;
} // main
```

Η διαφορά του από αυτά που είδαμε μέχρι τώρα βρίσκεται στις παραμέτρους που υπάρχουν στην επικεφαλίδα της **main**. Η δεύτερη παράμετρος είναι ένας πίνακας που κάθε του στοιχείο είναι ένα «C-style string»· μπορεί να τη δεις και ως `char** argv`. Η πρώτη παράμετρος μας λέει πόσα στοιχεία έχει πίνακας (+1)· για την ακρίβεια μας λέει τον δείκτη του τελευταίου στοιχείου.

Τι μπορεί να είναι αυτό το "`char* argv[]`"; Όπως είναι γραμμένο, μας λέει ότι έχουμε πίνακα που το κάθε στοιχείο του είναι τύπου "`char*`". Δηλαδή, μπορεί να είναι δισδιάστατος πίνακας με στοιχεία τύπου `char`; Ναι, αλλά κάπως διαφορετικός από αυτούς που είδαμε. Παρ' όλα αυτά μπορούμε να τον αξιοποιήσουμε με αυτά που μάθαμε.

Όπως βλέπεις, αυτό που κάνει το πρόγραμμά μας είναι να τυπώνει τις τιμές των παραμέτρων. Να ένα παράδειγμα εκτέλεσης: Ζητούμε να εκτελεστεί ως εξής:

```
E:\cpp2bk\progs>mainargs όρισμα1 a\b "όρισμα 3" 4 5.5 c\d?'ef<enter>
```

και παίρνουμε:

```
Η τιμή της argc είναι 7
Αυτά είναι τα ορίσματα που πέρασαν στην int main:
   argv[0]: mainargs
   argv[1]: όρισμα1
   argv[2]: a\b
   argv[3]: όρισμα 3
   argv[4]: 4
   argv[5]: 5.5
   argv[6]: c\d?'ef
   argv[7]:
```

Ας δούμε λοιπόν τις τιμές των στοιχείων του argv:

- Το `argv[0]` είναι το όνομα του αρχείου που περιέχει το (εκτελέσιμο) πρόγραμμα (μπορεί να δεις και την πλήρη διαδρομή (path) προς αυτό).

- Το `argv[1]` δείχνει τον πρώτο ορμαθό χαρακτήρων μετά το όνομα του αρχείου, δηλαδή το πρώτο όρισμα (στην περίπτωσή μας: `όρισμα1`).
- Το `argv[`*k*`]` δείχνει τον *k*-στό ορμαθό χαρακτήρων μετά το όνομα του αρχείου.
- Το `argv[argc-1]` δείχνει το τελευταίο όρισμα (ορμαθό χαρακτήρων).
- Το `argv[argc]` είναι φρουρός: περιέχει το (βέλος) 0.

Όπως βλέπεις, τα ορίσματα-ορμαθοί ξεχωρίζουν μεταξύ τους με ένα διάστημα. Στο τρίτο όρισμα θέλαμε να περάσουμε την τιμή `όρισμα 3`, που περιέχει διάστημα. Γι' αυτό υποχρεωθήκαμε να το γράψουμε: `"όρισμα 3"`.

12.8 Τελικώς ...

Τα βέλη είναι εργαλεία με τα οποία μπορείς να χειρίζεσαι πίνακες. Η χρήση βελών (υποτίθεται ότι) κάνει τα προγράμματά σου πιο γρήγορα αλλά η απλή κωδικοποίησή τους με δείκτες τα κάνει πιο ευανάγνωστα (και πιο σίγουρα). Προτίμησε λοιπόν τους δείκτες και χρησιμοποίησε βέλη μόνον όταν δεν μπορείς να τα αποφύγεις.

Δεν μπορείς να τα αποφύγεις (προς το παρόν) όταν περνάς πολυδιάστατο πίνακα σε συνάρτηση.

Όταν περνάς πίνακα σε συνάρτηση μην προσπαθείς να ξεγελάσεις τον μεταγλωττιστή με τεχνάσματα. Επαναλαμβάνουμε για τρίτη φορά τη συμβουλή που δώσαμε:

♦ ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Ασκήσεις

Α Ομάδα

12-1 Τι δίνει το παρακάτω πρόγραμμα; (εκτέλεσέ το *εσύ, όχι ο υπολογιστής*)

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

int qaw( int* p )
{
   *p = 5;
   return *p;
} // qaw

int main()
{
   int x( 0 );
   cout << " the value of x is " << x << endl;
   int y( qaw(&x) );
   cout << " the new value of x is " << x << endl;
   cout << " the value of y is " << y << endl;
} // main
```

12-2 Τα ίδια για το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   char a[] = "abcdefgh";
   for ( char* p(a); *p != 0; ++p )  cout << p << endl;
} // main
```

12-3 Έστω ότι έχουμε έναν τετραγωνικό πίνακα πραγματικών αριθμών. Το άθροισμα των στοιχείων της (κύριας) διαγωνίου ονομάζεται **ίχνος** (trace) του πίνακα. Γράψε συνάρτηση *trace* που θα τροφοδοτείται με τον πίνακα και θα υπολογίζει και θα επιστρέφει το ίχνος.

12-4 Γράψε συνάρτηση *trace2* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της *δεύτερης* διαγωνίου.

12-5 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων δύο στηλών *c1, c2* ενός δισδιάστατου πίνακα. Το ίδιο για τις τιμές των στοιχείων δύο γραμμών *r1, r2*.

12-6 Γράψε συνάρτηση *rawSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο *r1* και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής *r1*.

Γράψε συνάρτηση *colSum()* που θα κάνει το ίδιο για τα στοιχεία της στήλης *c1*.

12-7 Γράψε συνάρτηση *isDiagonal()* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα επιστρέφει τιμή `true` αν και μόνον αν ο πίνακας είναι διαγώνιος (όλα τα στοιχεία εκτος της *πρώτης* διαγωνίου είναι μηδέν).

Β Ομάδα

12-8 Τροποποίησε το πρόγραμμα της Άσκ. 9-2 ώστε να δίνει:

```
h
gh
fgh
efgh
defgh
cdefgh
bcdefgh
abcdefgh
```

Υπόδ.: Αν δεν μπορείς να το κάνεις με βέλη, κάνε το πρώτα όπως μπορείς και μετά μετάτρεψέ το ώστε να χρησιμοποιεί μόνο βέλη.

12-9 Γράψε συνάρτηση *periphSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών με *nR* γραμμές και *nC* στήλες και θα υπολογίζει και θα επιστρέφει το άθροισμα των περιφερειακών στοιχείων (των γραμμών 0 και *nR*-1 και των στηλών 0 και *nC*-1.)

12-10 Γράψε συνάρτηση *matrix2Sum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε συνάρτηση *matrix3Sum()* που θα τροφοδοτείται με τρισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε πρόγραμμα που θα δείχνει τον τρόπο χρήσης των δύο συναρτήσεων.

12-11 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο τις (πραγματικές) τιμές των στοιχείων ενός δισδιάστατου πίνακα και θα υπολογίζει και θα μας δίνει:

- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *γραμμών* του,
- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *στηλών* του.

Γ Ομάδα

12-12 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης ενός τετραγωνικού πίνακα. Δοκίμασε να κάνεις το ίδιο για τα στοιχεία της *r* γραμμής και της *c* στήλης όταν $r \neq c$.

12-13 Γράψε την *isPartSymmetric()* που προδιαγράψαμε στην§12.6.1.

κεφάλαιο

13

Συναρτήσεις ΙΙ – Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:

Αφού έμαθες –πολύ νωρίς– να γράφεις συναρτήσεις που σου χρειάζονται και δεν υπάρχουν στις βιβλιοθήκες της C++ τώρα *θα μάθεις να γράφεις δικές σου εντολές* ή αλλιώς συναρτήσεις `void`. Θα μάθεις ακόμη *να περνάς τιμές από μια συνάρτηση προς μια άλλη που την κάλεσε* μέσω παραμέτρων.

Προσδοκώμενα αποτελέσματα:

Με τα εργαλεία που σου δίνουμε μπορείς πια να γράψεις πρόγραμμα για να λύσεις μη τετριμμένα προβλήματα.

Έννοιες κλειδιά:

- *συναρτήσεις* `void`
- *παράμετρος-αναφοράς*
- *παράμετρος-βέλος* (*pointer*)
- *τύπος-αναφοράς*
- *παράμετρος-ρεύμα από/προς αρχείο*
- *δομημένος προγραμματισμός*
- *βήμα-προς-βήμα ανάλυση*
- *καθολικές μεταβλητές*
- *στατικές μεταβλητές*

Περιεχόμενα:

13.1 Ένα Παλιό Πρόβλημα Ξανά 344
13.2 Επιστροφή Τιμών από τη Συνάρτηση I 346
13.3 Επιστροφή Τιμών από τη Συνάρτηση II 348
13.3.1 Παράμετρος `unsigned`; (ξανά) 350
13.4 Τύποι Αναφοράς 350
13.5 Η Εντολή `return` (ξανά) 353
13.6 Εμβέλεια και Χρόνος Ζωής Μεταβλητών 353
13.6.1 * Στατικές Μεταβλητές 357
13.6.2 Καθολικά Αντικείμενα και Τεκμηρίωση 358
13.7 * Οι Συναρτήσεις στις Αποδείξεις (ξανά) 359
13.8 Ορμαθοί C και Αριθμοί (ξανά) 360
13.9 Πώς Επιλέγουμε το Είδος της Συνάρτησης 362
13.9.1 Περί Παραμέτρων 363
13.9.2 Παράμετρος – Ρεύμα 364
13.9.3 Παραδείγματα 365
13.10 Υποδείγματα Συναρτήσεων 370
13.11 Ένα Δύσκολο Πρόβλημα! 371
13.11.1 «Άνοιξε τα ρεύματα των αρχείων» 374
13.11.2 «Επεξεργασία» 376

13.11.3 «Κλείσε τα Ρεύματα» 382
13.11.4 Ολόκληρο το Πρόγραμμα 382
13.12 Δυο Λόγια για το Παράδειγμά μας 383
Ασκήσεις 385
Α Ομάδα 385
Β Ομάδα 385
Γ Ομάδα 386

Εισαγωγικές Παρατηρήσεις:

Έλυσες την Άσκ. 9-1; Ας τη δούμε μαζί:

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

int qaw( int* p )
{
   *p = 5;
   return *p;
} // qaw

int main()
{
   int x( 0 );
   cout << " the value of x is " << x << endl;
   int y( qaw(&x) );
   cout << " the new value of x is " << x << endl;
   cout << " the value of y is " << y << endl;
} // main
```

Η *qaw* έχει παράμετρο βέλος προς αντικείμενο τύπου `int`. Στη μοναδική κλήση της *qaw* –"`int y( qaw(&x) )`"– περνούμε ως όρισμα βέλος προς τη *x* και αυτό γίνεται τιμή του *p*. Άρα το `*`*p* –στο οποίο δίνουμε την τιμή 5– είναι ακριβώς ή *x*.

Έτσι, το πρόγραμμα θα μας δώσει:

```
the value of x is 0
the new value of x is 5
the value of y is 5
```

Ενώ μέχρι τώρα ξέραμε ότι ο μόνος τρόπος να πάρουμε τιμή από μια συνάρτηση ήταν το όνομά της με τα ορίσματα, τώρα βλέπουμε ότι μπορούμε να παίρνουμε τιμές και από τις παραμέτρους, αν αυτές είναι βέλη.

Αυτός είναι ο πάγιος τρόπος της C. Η C++ τον έχει κληρονομήσει αλλά μας δίνει και μια σαφώς πιο εύχρηστη παραλλαγή του. Αυτά και άλλα παρεμφερή θα δούμε σε αυτό το κεφάλαιο.

Ακόμη, στο κεφάλαιο αυτό θα γράψουμε και πρόγραμμα διαφορετικό από τα προηγούμενα. Μέχρι τώρα γράφαμε προγράμματα για να δείχνουμε μια νέα έννοια ή μια νέα τεχνική. Τώρα θα γράψουμε πρόγραμμα που θα λύνει ένα πρόβλημα.

13.1 Ένα Παλιό Πρόβλημα Ξανά

Στην §2.7 είχαμε λύσει το εξής πρόβλημα:

> *Από ύψος h αφήνεται να πέσει προς τη γή ένα σώμα. Να γραφεί πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του h και θα υπολογίζει και θα γράφει:*
> *α) τον χρόνο που θα κάνει το σώμα μέχρι να φτάσει στην επιφάνεια της γής*
> *β) η ταχύτητά του τη στιγμή της πρόσκρουσης.*
> *Να αγνοηθεί η αντίσταση του αέρα. Επιτάχυνση βαρύτητας: g = 9.81 m/sec².*

Είχαμε κάνει ένα σχέδιο για τη λύση:

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

και γράψαμε το πρόγραμμα.

Τώρα θέλουμε κάτι άλλο: Να διαχωρίσουμε τα δύο τελευταία βήματα, να τα βγάλουμε από τη **main** και να τα «κρύψουμε» σε ξεχωριστές συναρτήσεις, Γιατί; Διότι θέλουμε να έχουμε τη δυνατότητα να αλλάζουμε την υλοποίησή τους όποτε μας χρειαστεί.

Ας ξεκινήσουμε από το τελευταίο. Θέλουμε να γράψουμε μια συνάρτηση που θα τροφοδοτείται με τα *tP* και *vP* και θα γράφει στην οθόνη τις τιμές τους (μαζί και την αρχική τιμή του ύψους). Εύκολο φαίνεται, αλλά υπάρχει ένα ερώτημα: Τι τιμή θα επιστρέφει αυτή η συνάρτηση, αφού δεν υπάρχει κάτι που να υπολογίζει;

Θα μπορούσαμε να βάλουμε μια άχρηστη, επιστρεφόμενη τιμή, π.χ.:

```
int displayResults( double h, double t, double v )
{
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << t << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << v << " m/sec";
   return 0;
} // displayResults
```

που είναι μια χαρά. Η C++ όμως μας δίνει τη δυνατότητα να γράψουμε μια *συνάρτηση χωρίς τύπο*:

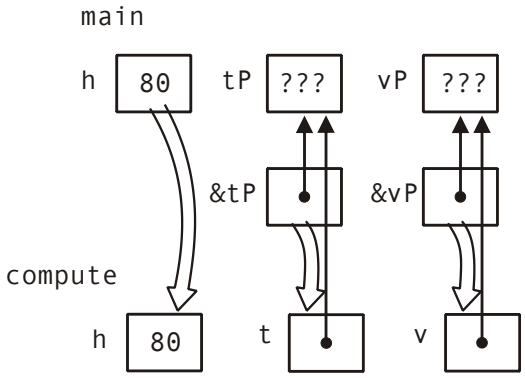
```
void displayResults( double h, double t, double v )
{
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << t << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << v << " m/sec" << endl;
} // displayResults
```

Το `void` στην αρχή δείχνει ότι *η συνάρτηση δεν επιστρέφει τιμή*.[1]

Πώς την καλούμε; Με το όνομά της και τα ορίσματά της, αλλά όχι μέσα σε παράσταση:

```
int main()
{
   const double  g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας

   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if (h >= 0)  {  // Υπολόγισε τα tP, vP
   // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
      tP = sqrt((2/g)*h);
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))

      displayResults( h, tP, vP );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

[1] Το «συνάρτηση χωρίς τύπο» δεν είναι και τόσο σωστό, τουλάχιστον κατ' αρχήν. Για τη C++ *ο* "`void`" *είναι τύπος με κενό σύνολο τιμών*.



Σχ. 13-1 Η κλήση της *compute* από τη **main**. Η τιμή της *h* της **main** αντιγράφεται ως τιμή της *h* της *compute()*. Στη μεταβλητή-βέλος *t* αντιγράφεται η τιμή-βέλος **&***tP*. Έτσι η *t* δείχνει τη μεταβλητή *tP* της **main**, δηλαδή ***t* είναι η *tP*, και αν δώσουμε με κάποιον τρόπο μια τιμή στην ***t* αυτή γίνεται τιμή της *tP*. Τα ίδια ισχύουν και για τις ***v* και *vP*.

13.2 Επιστροφή Τιμών από τη Συνάρτηση I

Τώρα, ας προσπαθήσουμε να γράψουμε μια συνάρτηση, ας την πούμε *compute*, που θα τροφοδοτείται με το ύψος (*h*) και θα υπολογίζει και θα επιστρέφει τα *tP*, *vP*.

Θα πεις: «αφού ξέρουμε ότι μια συνάρτηση επιστρέφει μια τιμή, ας γράψουμε *δύο* συναρτήσεις!» Όχι! Οι προδιαγραφές μας λένε *μία* συνάρτηση· μπορεί να γίνει κάτι;

Μήπως μπορούμε να παίρνουμε αποτελέσματα μέσω των παραμέτρων; Πώς; Στις «Εισαγωγικές Παρατηρήσεις» είδαμε τη λύση: θα βάλουμε παραμέτρους-βέλη! Δηλαδή, γράφουμε μια:

```
void compute( double h, double* t, double* v )
```

και την καλούμε ως εξής:

```
compute( h, &tP, &vP );
```

Και γιατί βάλαμε "`void`"; Διότι *η συνάρτηση δεν επιστρέφει τιμή* μέσα σε μια παράσταση που υπάρχει το όνομά της αλλά μόνο με τις παραμέτρους της. Τα λέμε πιο κάτω.

Ας δούμε, με τη βοήθεια του Σχ. 13-1, τι θα γίνει κατά την κλήση: η τιμή της *h* της **main** θα αντιγραφεί στην παράμετρο *h* της *compute()*. Η τιμή της παράστασης `&tP` θα αντιγραφεί στην παράμετρο *t* της *compute()*. Αλλά τι είναι η τιμή της `&tP`; Είναι ένα βέλος που δείχνει τη μεταβλητή *tP* της **main**. Άρα, μετά την αντιγραφή, η *t* της *compute()*, που είναι μεταβλητή βέλος (`double* t`), θα δείχνει επίσης τη μεταβλητή *tP* της **main**. Τα ίδια θα γίνουν και με τις *vP* και *v*. Αν λοιπόν, μέσα στην *compute()*, αλλάξω τις τιμές των ***t* και ***v* αλλάζω τις τιμές των *tP* και *vP* αντίστοιχα! Αυτό ήτανε λοιπόν:

```
void compute( double h, double* t, double* v )
{
   const double  g = 9.81; // m/sec², επιτάχυνση της βαρύτητας

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   *t = sqrt((2/g)*h);
   *v = -g*(*t);
// (*t ≈ √(2h/g)) && (*v ≈ -√(2hg))
} // compute
```

Και να πώς γίνεται τώρα πια η **main**:

```
int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
```

```
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   {
      compute( h, &tP, &vP );
      displayResults( h, tP, vP );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως η *displayResults()*, έτσι και η *compute* είναι μια συνάρτηση χωρίς τύπο (**void**). Δηλαδή δεν την καλούμε μέσα σε μια παράσταση, όπου θα μας επιστρέψει κάποια τιμή. Πώς παίρνουμε τα αποτελέσματα των υπολογισμών που κάνει; Μέσω των παραμέτρων: για την ακρίβεια, μέσω των παραμέτρων γνωστοποιούμε στη συνάρτηση τις θέσεις της μνήμης (**&tP** και **&vP**) όπου θέλουμε να αποθηκευτούν τα αποτελέσματα και αφού τελειώσει τη δουλειά της τα παίρνουμε και τα χρησιμοποιούμε.

Ας γράψουμε με τον ίδιο τρόπο και μια συνάρτηση, με όνομα *inputH()*, που θα διαβάζει από το πληκτρολόγιο το ύψος και θα το φέρνει στη **main**, όταν την καλεί.[2] Αν ο χρήστης δώσει αρνητικό αριθμό θα του δίνει *μία* ευκαιρία να διορθώσει το λάθος του.

```
   void inputH( double* h )
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> *h;
      if ( *h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> *h;  }
   } // inputH
```

Και να πώς γίνεται τελικώς το πρόγραμμά μας:

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

   void inputH( double* h )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   void compute( double h, double* t, double* v )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   void displayResults( double h, double tP, double vP )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   inputH( &h );
   if ( h >= 0 )
   {
      compute( h, &tP, &vP );
      displayResults( h, tP, vP );
```

[2] Βέβαια, αφού η *inputH()* επιστρέφει μια τιμή θα μπορούσαμε να γράψουμε συνάρτηση με τύπο:

```
   double inputH()
   {
      double locH;
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> locH;
      if ( locH < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> locH;  }
      return locH;
   } // inputH
```

Ναι, θα μπορούσαμε, αλλά... διάβασε παρακάτω.

```
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως βλέπεις, όλες οι δουλειές γίνονται στις συναρτήσεις και η **main** παίζει πια ρόλο «συντονιστικό». Και μπορεί αυτό να μη λέει και πολλά πράγματα σε ένα τόσο μικρό πρόγραμμα, αλλά τα μεγάλα προγράμματα δεν είναι δυνατό να γραφούν διαφορετικά.

Πόσο μεγάλες είναι αυτές οι συναρτήσεις; Από παλιά, οι προγραμματιστές έχουν έναν εμπειρικό κανόνα που λέει: *καμιά συνάρτηση του προγράμματος δεν πιάνει περισσότερο από μια σελίδα εκτύπωσης* (αυτό σημαίνει 60 γραμμές περίπου). Πάντως, ένας πιο καλός κανόνας είναι ο εξής: *κάθε συνάρτηση ανταποκρίνεται σε έναν συγκεκριμένο υπολογιστικό στόχο*. Φυσικά, κάθε υπολογιστικός στόχος μπορεί να διασπασθεί σε μικρότερους. Αυτό αντιστοιχεί σε μια συνάρτηση που καλεί άλλες συναρτήσεις. Στη συνέχεια θα δούμε αυτά τα πράγματα με παράδειγμα.

Όπως είπαμε, αυτός ο τρόπος για να βγάζουμε τιμές από μια συνάρτηση μέσω των παραμέτρων είναι κληρονομιά από τη C. Στην επόμενη παράγραφο θα μάθουμε έναν άλλον τρόπο που μας δίνει η C++. Πάντως, πολλοί προγραμματιστές προτιμούν τον τρόπο της C για λόγους που θα εξηγήσουμε αργότερα.

13.3 Επιστροφή Τιμών από τη Συνάρτηση II

Θα ξεκινήσουμε με έναν κανόνα, που θα εξηγηθεί αργότερα:

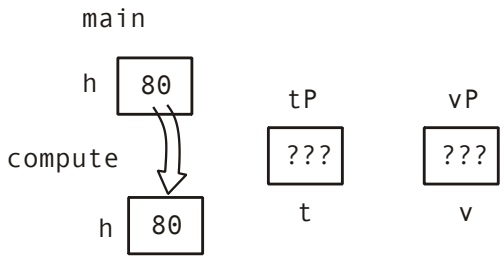
- ***Αν μετά τον τύπο της παραμέτρου μιας συνάρτησης βάλεις το σύμβολο "&" τότε η παράμετρος είναι «διπλής κατεύθυνσης»· δηλαδή οι αλλαγές της τιμής της τυπικής παραμέτρου, που γίνονται μέσα στη συνάρτηση, γίνονται αλλαγές τιμής της αντίστοιχης πραγματικής παραμέτρου.***

Αυτές οι παράμετροι λέγονται **παράμετροι αναφοράς** (reference parameters). Όπως καταλαβαίνεις,

- ***Η πραγματική παράμετρος που αντιστοιχεί σε μια παράμετρο αναφοράς δεν μπορεί να είναι σταθερά ή παράσταση, αλλά κάτι που να μπορεί να αλλάξει η τιμή του δηλαδή τροποποιήσιμη τιμή-l, π.χ. μια μεταβλητή ή ένα στοιχείο πίνακα.***

Ξαναγράφουμε ολόκληρο το πρόγραμμα της προηγούμενης παραγράφου χρησιμοποιώντας, αντί για βέλη, παραμέτρους αναφοράς:

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

   void inputH( double& h )
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
      if ( h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> h;  }
   } // inputH

   void compute( double h, double& t, double& v )
   {
      const double  g = 9.81; // m/sec2, επιτάχυνση της βαρύτητας

   // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
      t = sqrt( (2/g)*h );
      v = -g*t;
   // (t ≈ √(2h/g)) && (v ≈ -√(2hg))
   } // compute
```



Σχ. 13-2 Όταν καλείται η *compute()*, η πραγματική παράμετρος τιμής (*h* της **main**) αντιγράφεται στην τυπική (*h* της *compute*). Για τις παραμέτρους αναφοράς τα πράγματα είναι αλλιώς: οι δύο συναρτήσεις «κατέχουν από κοινού» τις ίδιες θέσεις της μνήμης, τις *tP*, *vP* της **main**. Η καλούμενη συνάρτηση (*compute*) δίνει δικά της ονόματα στις δύο θέσεις και τις χειρίζεται με αυτά.

```
   void displayResults(double h, double tP, double vP)
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   inputH( h );
   if ( h >= 0 )
   {
      compute( h, tP, vP );
      displayResults( h, tP, vP  );
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Πρόσεξε την επικεφαλίδα της *compute()*:

```
   void compute( double h, double& t, double& v )
```

Ο τύπος των *t* και *v* είναι **double&**. Δες και την κλήση της:

```
      compute( h, tP, vP );
```

Στις δύο τελευταίες παραμέτρους βάζουμε απλώς τα αναγνωριστικά των αντίστοιχων μεταβλητών: **tP**, **vP**.

Πρόσεξε τώρα το σώμα της *compute()*: Οι εκχωρήσεις γίνονται στις *t* και *v* και όχι στις ***t** και ***v** που είχαμε στην προηγούμενη παράγραφο. Παρόμοιες διαφορές βλέπεις και στην *inputH()*.

Όπως βλέπεις, οι παράμετροι αναφοράς σου δίνουν τη δυνατότητα να γράφεις το πρόγραμμά σου πιο απλά. Και, παρ' όλο που οι δύο τρόποι είναι κατά βάση ίδιοι, δες και ένα σενάριο κλήσης και για να τις σκέφτεσαι πιο απλά.

Όταν καλείται η συνάρτηση, στην περίπτωσή μας η *compute()*, της παραχωρείται μνήμη για τις παραμέτρους τιμής (όπως είναι η *h*) και τα τοπικά της αντικείμενα (όπως είναι η σταθερά *g*) αλλά όχι για τις παραμέτρους αναφοράς. Για αυτές γίνεται το εξής: της γνωστοποιούνται οι θέσεις τους στη μνήμη και η συνάρτηση τους δίνει δικά της ονόματα. Όσο εκτελείται η *compute()* μπορείς να σκέφτεσαι ότι η *tP* έχει δύο ονόματα: *tP* για τη **main** και *t* για την *compute()*. Παρομοίως, η *vP* της **main** έχει το όνομα *v* για την *compute()*. Δηλαδή, η συνάρτηση που καλεί και η συνάρτηση που καλείται «κατέχουν από κοινού» τις παραμέτρους αναφοράς. Αυτό προσπαθεί να δείξει και το Σχ. 13-2.

Το σενάριό μας δεν απέχει πολύ από την αλήθεια: Αν ζητήσεις να δεις τις διευθύνσεις των *tP* και *vP* της **main** και αυτές των *t* και *v* της *compute()* θα τις βρεις ίδιες.

Αλλά τώρα προσοχή:

Δηλαδή, αν έχεις τη

```
void inputH( double& h )
```

και τις:

```
int i;
double d;
```

μπορείς να βάλεις:

```
inputH( d );
```

αφού η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, *d*, είναι τύπου `double`.

Δεν μπορείς να βάλεις:

```
inputH( i );
```

διότι η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, *i*, είναι τύπου `int`.

Δεν μπορείς να βάλεις:

```
inputH( 1.0 + sqrt(d) );
```

διότι η τυπική παράμετρος, *h*, είναι τύπου `double&` και η πραγματική, `1.0 + sqrt(d)`, είναι παράσταση και όχι μεταβλητή.

Αν έχεις καταλάβει τον τρόπο αντιστοίχισης, που περιγράψαμε πιο πάνω, αυτά είναι αυτονόητα.

Στη συνέχεια θα προτιμήσουμε τις παραμέτρους αναφοράς από τις παραμέτρους-βέλη. Έτσι:

- όταν θέλουμε παράμετρο «μονής κατεύθυνσης», που θα μεταβιβάζει στοιχεία από την καλούσα συνάρτηση προς την καλούμενη θα χρησιμοποιούμε παράμετρο τιμής,
- όταν θέλουμε παράμετρο «διπλής κατεύθυνσης», που θα μεταβιβάζει στοιχεία και από την καλούσα προς την καλούμενη και αντίθετα, από την καλούμενη προς την καλούσα, θα χρησιμοποιούμε παράμετρο αναφοράς.

13.3.1 Παράμετρος `unsigned`; (ξανά)

Τώρα, θα ξαναδιατυπώσουμε, πιο προσεκτικά, τον κανόνα που δώσαμε στην §7.6:

♦ ***Μην βάζεις στις συναρτήσεις σου παραμέτρους τιμής τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

Για τις παραμέτρους αναφοράς αυτά δεν ισχύουν· αυτό που ισχύει είναι: *Ο τύπος της πραγματικής παραμέτρου θα πρέπει να είναι ίδιος με αυτόν της τυπικής παραμέτρου* (αλλιώς δεν περνάει από τον μεταγλωττιστή).

13.4 Τύποι Αναφοράς

Εκτός από τη λίστα παραμέτρων, μπορούμε να βάλουμε **τύπο αναφοράς** (reference type) και σε δηλώσεις μεταβλητών μέσα στη συνάρτηση (ή σε καθολικές). Ας πούμε ότι δηλώνουμε:

```
int x;
int& r( x );
```

Τι καταφέραμε; *Τα αναγνωριστικά x και r καθορίζουν την ίδια θέση της μνήμης*. Έτσι, αν δώσουμε τις εντολές:

```
r = 10;
cout << " x = " << x << "   r = " << r << endl;
x = 15;
```

```
   cout << " x = " << x << "   r = " << r << endl;
```

θα πάρουμε:

```
 x = 10   r = 10
 x = 15   r = 15
```

Δηλαδή, εκείνο το "`r(x)`" στη δήλωση της *r* της δίνει όχι την τιμή της *x* αλλά τη θέση.

Μα κάτι τέτοιο δεν κάνουμε και με τα βέλη; Περίπου! Να πώς γίνονται τα ίδια πράγματα με βέλη:

```
   int* p( &x );
   *p = 10;
   cout << " x = " << x << "   *p = " << *p << endl;
   x = 15;
   cout << " x = " << x << "   *p = " << *p << endl;
```

Αποτέλεσμα:

```
 x = 10   *p = 10
 x = 15   *p = 15
```

Τι διαφορές έχουμε;

- Ως αρχική τιμή στην *p* δίνουμε την `&x` και όχι τη `x`.
- Ακόμη, κάθε φορά που χρησιμοποιούμε τη μεταβλητή μέσω της *p* πρέπει να κάνουμε αποπαραπομπή (`*p`).
- Τέλος, *στη δήλωση μιας μεταβλητής αναφοράς πρέπει να δίνουμε οπωσδήποτε αρχική τιμή*, ενώ με τα βέλη αυτό δεν είναι υποχρεωτικό.

Μια συνάρτηση με τύπο μπορεί να έχει ως τύπο επιστροφής έναν τύπο αναφοράς. Ας δούμε ένα

Παράδειγμα

Αντιγράφουμε και αλλάζουμε λίγο την *maxNdx* από την §9.3:

```
   int& maxElmn( int x[], int n )
   {
      int mxP( 0 );

      for ( int m(1); m < n; ++m )
      {
         if ( x[m] > x[mxP] )  mxP = m;
      } // for (m …

      return x[mxP];
   } // maxElmn
```

Η *maxElmn* επιστρέφει τη μέγιστη τιμή από αυτές των στοιχείων του πίνακα και όχι τη θέση του.

Πρόσεξε ότι:

- Ως τύπο επιστροφής βάλαμε `int&` και όχι `int`. Στη `return` όμως επιστρέφουμε την x[*mxP*], δηλαδή μια τιμή τύπου `int`.
- Η πρώτη παράμετρος είναι `int x[]` και όχι `const int x[]`. Διάβασε παρακάτω τα σχετικά.

Πώς καλούμε τη συνάρτηση; Όπως ξέρουμε. Αν έχουμε δηλώσει:

```
   int zm, z[] = { 12, 4, -3, 24, 33, 2, 4, 7 };
```

μπορούμε να γράψουμε:

```
   zm = maxElmn( z, 8 );
   cout << zm << "   " << maxElmnD(z, 8) << endl;
```

που θα μας δώσουν:

```
33   33
```

Δηλαδή τα πάντα δουλεύουν σαν να είχαμε τύπο επιστροφής `int` και όχι `int&`.

Γιατί βγάλαμε το "`const`"; Διότι δεν το δέχεται ο μεταγλωττιστής και να το πρόβλημα που έχει: Οι εντολές

```
for ( int k(0); k < 8; ++k )  cout << z[k] << "  ";
cout << endl;
maxElmn( z, 8 ) = 17;
for ( int k(0); k < 8; ++k )  cout << z[k] << "  ";
cout << endl;
```

θα δώσουν:

```
12  4  -3  24  33  2  4  7
12  4  -3  24  17  2  4  7
```

Δηλαδή, η "`maxElmn(z, 8) = 17`" όχι μόνον είναι δεκτή αλλά μας επιτρέπει να αλλάξουμε την τιμή του στοιχείου `z[4]`! Βάζοντας ως τύπο επιστροφής της συνάρτησης "`int&`" η συνάρτηση επιστρέφει *το στοιχείο* με τη μέγιστη τιμή (τιμή-*l*) και όχι απλώς την τιμή του. Παρακάτω θα διαβάσεις πώς γίνεται αυτό. Πώς διορθώνεται όμως; Έτσι:

```
const int& maxElmn( const int x[], int n )
```

και η παράξενη εκχώριση απαγορεύεται!

Και πού θα μας χρειασθούν αυτά τα μπερδεμένα πράγματα; Ας δούμε τι ακριβώς γίνεται και μετά θα απαντήσουμε. Όταν εκτελείται η κλήση της συνάρτησης, στην εντολή "`zm = maxElmn(z, 8)`", γίνονται τα εξής:

- υπολογίζεται η *mxP* και
- η `return x[mxP]` *αντιγράφει*, σε κάποια θέση της μνήμης, *ένα βέλος* προς το στοιχείο x[*mxP*].

Όταν στη συνέχεια εκτελείται η εκχώρηση, *στην zm αντιγράφεται η τιμή που δείχνει το βέλος*.

Αν είχαμε βάλει ως τύπο επιστροφής `int` αντί για `int&` τι θα γινόταν; Θα αντιγραφόταν, από τη `return`, η τιμή του x[*mxP*] και στη συνέχεια θα είχαμε άλλη μια αντιγραφή αυτής της τιμής κατά την εκχώρηση. Δηλαδή:

- αν έχουμε τύπο επιστροφής τον *T*`&` έχουμε μια αντιγραφή βέλους και μια αντιγραφή τιμής τύπου *T* ενώ
- αν έχουμε τύπο επιστροφής τον *T* έχουμε δύο αντιγραφές τιμών τύπου *T*.

«Σιγά τη διαφορά!» θα πεις και θα έχεις δίκιο αν *T* είναι ο `int`. Αν όμως οι τιμές τύπου *T* πιάνουν μερικά *kB* η κάθε μια τότε τα πράγματα αλλάζουν: βάζοντας ως τύπο της συνάρτησης τον *T*`&` έχουμε πιο γρήγορη εκτέλεση! Θα δούμε τέτοιους «μεγάλους» τύπους στη συνέχεια.

Τώρα όμως προσοχή! Θα μπορούσαμε να αλλάξουμε την *maxNdx* σε:

```
int& maxPosD( const int x[], int n )
{
   int mxP( 0 );

   for ( int m(1); m < n; ++m )
   {
      if ( x[m] > x[mxP] )  mxP = m;
   } // for (m …

   return mxP;
} // maxNdx
```

Όχι! Και να γιατί: Ας πούμε ότι είχαμε την εντολή:

```
p = maxNdx( z, 8 );
```

Η εκτέλεση της `return mxP` θα αντέγραφε ένα βέλος προς την *mxP*, *που είναι μια μεταβλητή τοπική στη συνάρτηση*. Όταν θα έφθανε η ώρα να εκτελεσθεί η αντιγραφή της τιμής στην *p*, η εκτέλεση της συνάρτησης θα είχε τελειώσει και *η mxP δεν θα υπήρχε*! Δίδαγμα:

- ***Αν ο τύπος επιστροφής μιας συνάρτησης είναι τύπος αναφοράς η συνάρτηση δεν θα πρέπει να επιστρέφει ως τιμή κάποιο τοπικό αντικείμενο.***

13.5 Η Εντολή `return` (ξανά)

Είπαμε στην §7.2 ότι: η **`return`** Π «*επιστρέφει την τιμή της Π αφού τη μετατρέψει στον τύπο της συνάρτησης*. Αλλά η **`return`** κάνει και κάτι άλλο: *τελειώνει την εκτέλεση της συνάρτησης στην οποία υπάρχει* και η εκτέλεση συνεχίζεται στο σημείο που κλήθηκε· οι εντολές που τυχόν την ακολουθούν δεν θα εκτελεσθούν.»

Δηλαδή, σε μια συνάρτηση χωρίς τύπο, που οι επιστροφές τιμών γίνονται μέσω των παραμέτρων, δεν μας χρειάζεται **`return`**; Ναι, και γι' αυτό δεν τη χρησιμοποιήσαμε. Πάντως υπάρχει και μορφή –χωρίς παράσταση– που απλώς τελειώνει την εκτέλεση της συνάρτησης· αυτή μπορεί να χρησιμοποιηθεί.

Παράδειγμα

Η παρακάτω συνάρτηση μας επιστρέφει την ελάχιστη (*mnxy*) και τη μέγιστη (*mxxy*) από δύο ακέραιες τιμές:

```
void minmax(int x, int y, int& mnxy, int& mxxy)
{
   if (x < y)  {  mnxy = x;  mxxy = y;  return;  }
   mnxy = y;  mxxy = x;
} // minmax
```

Αν ισχύει η συνθήκη $x < y$ θα εκτελεσθούν οι "**`mnxy = x; mxxy = y;`**" και στη συνέχεια, με την εκτέλεση της **`return`** τελειώνει η εκτέλεση της συνάρτησης· οι "**`mnxy = y; mxxy = x;`**", που ακολουθούν, δεν εκτελούνται. Αν δεν ισχύει η συνθήκη θα εκτελεσθούν μόνον οι "**`mnxy = y; mxxy = x;`**" και θα τελειώσει η εκτέλεση της συνάρτησης.

Όπως βλέπεις όμως, η χρήση της **`return`**, στις συναρτήσεις χωρίς τύπο, σημαίνει παράβαση του κανόνα «*μία είσοδος και μία έξοδος*». Για τον λόγο αυτόν θα αποφεύγουμε τη χρήση της.

13.6 Εμβέλεια και Χρόνος Ζωής Μεταβλητών

Όπως λέγαμε και στο Κεφ. 7,

> *«Ένα πρόγραμμα της C++ είναι ένα σύνολο από συναρτήσεις. Μια από αυτές τις συναρτήσεις έχει το όνομα* **`main`** *και από αυτήν αρχίζει η εκτέλεση του προγράμματος.*
>
> *Κάθε συνάρτηση είναι μια σύνθετη εντολή (σώμα της συνάρτησης) με μια επικεφαλίδα. Στην επικεφαλίδα, δηλώνεται και ένα όνομα που χαρακτηρίζει τη συνάρτηση. Η σύνθετη εντολή αποτελείται από τις δηλώσεις των σταθερών, των τύπων και των μεταβλητών και τις άλλες εντολές. Στις εντολές αυτές μπορεί να περιλαμβάνονται άλλες σύνθετες εντολές με το περιεχόμενο που περιγράφουμε (δηλ. δηλώσεις κλπ).»*

Τώρα θα σταθούμε σε δύο σημεία:

- Δηλώσεις μπορεί να υπάρχουν και έξω από οποιαδήποτε συνάρτηση (καθολικές).
- Δηλώσεις μπορεί να υπάρχουν και μέσα σε οποιαδήποτε σύνθετη εντολή (τοπικές στη σύνθετη εντολή).

Ξεκινούμε με την πρώτη περίπτωση ξαναγράφοντας το παράδειγμα που είδαμε πιο πριν κάπως διαφορετικά.

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

   const double  g = 9.81; // m/sec2, επιτάχυνση της βαρύτητας

   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

   void inputH()
   {
      cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
      if ( h < 0 )
      {  cout << " Πρέπει να είναι θετικός! ";  cin >> h;  }
   } // inputH

   void compute()
   {
   // (g == 9.81) && (0 <= h <= DBL_MAX)
      tP = sqrt( (2/g)*h );
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   } // compute

   void displayResults()
   {
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   } // displayResults

int main()
{
   inputH();
   if ( h >= 0 )
   {
      compute();
      displayResults();
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
} // main
```

Όπως βλέπεις:

- οι συναρτήσεις δεν έχουν παραμέτρους και
- οι μεταβλητές έχουν δηλωθεί έξω από τις συναρτήσεις.

Λέμε ότι, στην περίπτωση αυτή, οι μεταβλητές είναι **καθολικές** (global)· *μπορεί να τις χρησιμοποιεί και να αλλάζει τις τιμές τους οποιαδήποτε συνάρτηση.*

Οποιαδήποτε; Και αν έχω, ας πούμε, μια καθολική μεταβλητή *x* και μια μεταβλητή *x* τοπική σε κάποια συνάρτηση τότε τι γίνεται; Όσο εκτελείται η συνάρτηση, με το όνομα *x*, ξέρει την τοπική μεταβλητή. Μπορεί όμως να δει και την καθολική *x* ως "`::x`". Αν έχουμε για παράδειγμα:

```
   int x;

   void q( double x )
   {
      cout << x << "  " << ::x << endl;
//       . . .
   } // q

int main()
{
// . . .
```

```
   x = 4;  q( 2.5 );
// . . .
}
```

Στη **main**, η *καθολική* μεταβλητή *x* παίρνει τιμή 4 και στη συνέχεια καλείται η *q* με πραγματική παράμετρο 2.5. Αυτή γίνεται τιμή της τυπικής παραμέτρου *x*, που είναι *τοπική* στην *q*. Όταν εκτελεσθεί η εντολή `cout << x << "  " << ::x << endl` θα καταλάβει ως *x* την τοπική παράμετρο, ενώ θα μας δώσει και την τιμή της καθολικής από την `::x`. Έτσι, θα πάρουμε:

```
2.5  4
```

Εκτός από αυτό το χαρακτηριστικό, της «καθολικής ορατότητας», οι καθολικές μεταβλητές έχουν και ένα άλλο ενδιαφέρον χαρακτηριστικό: είναι **στατικές**, δηλαδή ζουν –άρα κρατούν την τιμή τους– από την αρχή μέχρι το τέλος της εκτέλεσης του προγράμματος.

Τώρα θα ξαναδούμε τα περί τοπικών μεταβλητών. Αλλά πριν προχωρήσεις ξαναδιάβασε την §11.1.

Για τη C++, μια **ομάδα** (block) είναι μια **σύνθετη εντολή** (compound statement), δηλαδή ό,τι υπάρχει ανάμεσα σε ένα ζευγάρι "`{`" και "`}`". Όπως έχουμε δει, μπορεί να υπάρχει *ομάδα μέσα σε ομάδα*.

Σε ένα πρόγραμμα C++:

- Μπορείς να δηλώσεις μια μεταβλητή σε οποιοδήποτε σημείο, αλλά πριν τη χρησιμοποιήσεις.
- Η εμβέλειά της ξεκινάει από το σημείο που έγινε η δήλωση και τελειώνει στο τέλος της πιο εσωτερικής ομάδας που περιλαμβάνει τη δήλωση.

Ας δούμε τι ακριβώς συμβαίνει με ένα παράδειγμα. Δες το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   { // ομάδα 1.1
      int x( 5 );
      cout << "p2: x = " << x << endl;
   }
   cout << "p3: x = " << x << endl;
// . . .
```

Αν προσπαθήσεις να το μεταγλωττίσεις θα πάρεις μήνυμα:

```
Error … line 9: Undefined symbol 'x' in function main()
```

Τι συμβαίνει; Η γραμμή 9 (`cout << "p3: x = " << x << endl`) βρίσκεται στην ομάδα 1, όπου δεν έχει δηλωθεί μεταβλητή. Η *x* έχει δηλωθεί στην ομάδα 1.1, είναι γνωστή μέσα σ' αυτήν –έτσι δεν υπάρχει πρόβλημα με την εντολή "`cout << "p2: x = " << x << endl`"– αλλά δεν είναι γνωστή στη γραμμή 9 που βρίσκεται έξω από την ομάδα (που τελειώνει στη γραμμή 8).

Οι εντολές μιας ομάδας μπορεί να βλέπουν αντικείμενα που δηλώνονται σε ομάδες που την περιβάλλουν. Έστω π.χ. το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   int y;

   { // ομάδα 1.1
      y = 19;
      cout << "p1: y = " << y << endl;
   }
   cout << "p2: y = " << y << endl;
   . . .
```

που δίνει:

Πλαίσιο 13.1

Κανόνας Εμβέλειας

Κάθε δήλωση ισχύει στη σύνθετη εντολή όπου έγινε και στις σύνθετες εντολές που είναι εσωτερικές σε αυτήν. Δεν ισχύει στις εσωτερικές σύνθετες εντολές που υπάρχει άλλος ορισμός (άλλη δήλωση) για το ίδιο όνομα. Οι παράμετροι μιας συνάρτησης είναι σαν να δηλώνονται στην ομάδα που ακολουθεί την επικεφαλίδα.

Το ίδιο πράγμα με άλλα λόγια:

Σε κάθε σύνθετη εντολή είναι «ορατά» όλα τα αντικείμενα που δηλώνονται σε αυτήν –τοπικά (local)– και ακόμη όλα αυτά που είναι «ορατά» στην ομάδα που την περιβάλλει –καθολικά (global). Από αυτά που έρχονται από το περιβάλλον δεν είναι «ορατά» αυτά που έχουν το ίδιο όνομα με κάποιο τοπικό αντικείμενο.

Κανόνας Διάρκειας Ζωής

Κάθε αντικείμενο του προγράμματός μας υπάρχει όσο εκτελείται η σύνθετη εντολή όπου έχει δηλωθεί. Τα καθολικά και αυτά που έχουν δηλωθεί με προδιαγραφή `static` ***υπάρχουν από την αρχή μέχρι το τέλος της εκτέλεσης του προγράμματος.***

```
p1: y = 19
p2: y = 19
```

Και οι δύο εντολές της ομάδας 1.1 διαχειρίζονται μια μεταβλητή, την *y*, που δηλώνεται στην ομάδα 1. Η δεύτερη γραμμή των αποτελεσμάτων μας (`p2: y = 19`) δείχνει ότι η τιμή που εκχωρήθηκε στην *y* στην ομάδα 1.1 διατηρείται και όταν τελειώσει η εκτέλεση αυτής της ομάδας.

Τι γίνεται όμως όταν ένα όνομα, ας πούμε *x*, χρησιμοποιείται σε κάποια ομάδα και έξω από αυτήν; Οι εντολές που υπάρχουν στην ομάδα, με το *x* καταλαβαίνουν το αντικείμενο που έχει δηλωθεί *μέσα* στην ομάδα. Για παράδειγμα, το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{ // ομάδα 1
   int x( 1 );

   cout << "p1: x = " << x << endl;
   { // ομάδα 1.1
      const int x( 5 );
      cout << "p2: x = " << x << endl;
   }
   cout << "p3: x = " << x << endl;
}
```

μας δίνει:

```
p1: x = 1
p2: x = 5
p3: x = 1
```

Η γραμμή "`p1: x = 1`" προέρχεται από την εντολή εκτύπωσης: "`cout << "p1: x = " << x << endl`", που «βλέπει» την αρχική δήλωση "`int x = 1`", της ομάδας 1.

Η "`p2: x = 5`" προέρχεται από τη "`cout << "p1: x = " << x << endl`" που υπάρχει στην ομάδα 1.1. Η σταθερά *x* αυτής της ομάδας δεν έχει σχέση με τη μεταβλητή *x* που δηλώσαμε στην ομάδα 1. Οι εντολές της ομάδας 1.1 «βλέπουν» μόνον τη «δική» τους *x* ενώ δεν έχουν δυνατότητα να «δουν» τη *x* της ομάδας 1.

Η γραμμή "`p3: x = 1`" προέρχεται από την εντολή εκτύπωσης: "`cout << "p3: x = " << x << endl`", που ανήκει στην ομάδα 1 και «βλέπει» την αρχική δήλωση "`int x = 1`", της ομάδας 1.

Και τι συμβαίνει με τη «ζωή» των αντικειμένων: Τα αντικείμενα που δηλώνονται σε μια ομάδα «ζουν», υπάρχουν, όσο εκτελούνται οι εντολές της ομάδας αυτής. Π.χ. η μεταβλητή *x* που δηλώνεται στην ομάδα 1 ζει όσο εκτελείται η ομάδα της **main**. Η σταθερά *x* που δηλώνεται στην ομάδα 1.1 ζει όσο εκτελείται η ομάδα 1.1 ή –αφού πρόκειται για έναν ορισμό– αυτός ο ορισμός ισχύει μόνον όσο εκτελείται η ομάδα 1.1. Όσο εκτελείται η ομάδα 1.1 το πρόγραμμά μας έχει στη διάθεσή του μια θέση μνήμης για τη μεταβλητή *x* της ομάδας 1 και μια θέση για τη σταθερά της ομάδας 1.1. Όταν έρχεται η ώρα να εκτελεστεί η εντολή "`cout << "p3: x = " << x << endl`", που βρίσκεται μετά το τέλος της ομάδας 1.1, η σταθερά *x* δεν υπάρχει· έχουμε μείνει μόνον με τη μεταβλητή που είχαμε αρχικά.

Όλα αυτά υπάρχουν μαζεμένα στο Πλ. 13.1.

Υπάρχουν και δύο ειδικές περιπτώσεις:

1. Οι παράμετροι μιας συνάρτησης έχουν εμβέλεια όλο το σώμα της συνάρτησης και χρόνο ζωής το χρόνο εκτέλεσης της συνάρτησης.
2. Όπως είδαμε, μπορούμε να δηλώσουμε μεταβλητές στην αρχική εντολή της **for**. Στην περίπτωση αυτή η εμβέλεια των μεταβλητών εκτείνεται μέχρι και την τελευταία επαναλαμβανόμενη εντολή και χρόνος ζωής μέχρι το τέλος της εκτέλεσης της επαναλαμβανόμενης εντολής.

Μπορούμε να έχουμε τοπικές μεταβλητές που ζούν σε όλη τη διάρκεια της εκτέλεσης του προγράμματος; Ναι! Διάβασε την επόμενη παράγραφο.

13.6.1 * Στατικές Μεταβλητές

Αν θέλεις «να κρατάς πάντοτε στη ζωή» κάποιο αντικείμενο μιας συνάρτησης μπορείς να το δηλώσεις ως `static`, όπως δηλώνουμε την τοπική μεταβλητή *a* στην παρακάτω συνάρτηση[3]:

```
double rnd01()
{
   static double a( 0.13579246801357924680 );

   cout << " rnd01 arxh: a = " << a << endl;
   a = 37*a;
   a = a - static_cast<int>(a);
   cout << " rnd01 telos: a = " << a << endl;
   return a;
} // rnd01
```

Ας πούμε ότι έχουμε και την:

```
double f( double x )
{
   double z;

   cout << " f arxh: z = " << z << endl;
   z = pow( x, 0.25 );
   cout << " f telos: z = " << z << endl;
   return x + z;
} // f
```

που κι αυτή έχει μια τοπική μεταβλητή, τη *z*, που όμως δεν δηλώνεται `static`. Και στις δύο συναρτήσεις έχουμε βάλει εντολές που τυπώνουν τις τιμές των τοπικών μεταβλητών στην αρχή και στο τέλος της εκτέλεσής τους. Ζητούμε το εξής:

```
   for ( int k(1); k <= 3; ++k )
```

[3] Πρόκειται για μια απλοϊκή μέθοδο για την παραγωγή ψευδοτυχαίων αριθμών στο διάστημα (0,1).

```
        cout << k << "  " << f(rnd01()) << endl;
```

και να τι παίρνουμε:

```
 rnd01 arxh: a = 0.135792
 rnd01 telos: a = 0.0243213
 f arxh: z = 4.97035e-219
 f telos: z = 0.394909
1  0.41923
 rnd01 arxh: a = 0.0243213
 rnd01 telos: a = 0.899889
 f arxh: z = 4.97035e-219
 f telos: z = 0.973974
2  1.87386
 rnd01 arxh: a = 0.899889
 rnd01 telos: a = 0.295882
 f arxh: z = 4.97035e-219
 f telos: z = 0.73753
3  1.03341
```

Βλέπουμε λοιπόν ότι, κατά την πρώτη κλήση της *rnd*01, η *a* έχει στο τέλος τιμή 0.0243213 και αυτή ακριβώς είναι η τιμή της στην αρχή της δεύτερης κλήσης. Κατά την πρώτη κλήση της *f*, στο τέλος, η *z* έχει τιμή 0.394909 αλλά στην αρχή της δεύτερης κλήσης έχει τιμή 4.97035*e*–219.

Πώς εξηγούνται αυτά; Η *z* είναι μια **αυτόματη** (automatic) μεταβλητή που δημιουργείται όταν ενεργοποιείται η *f* και εξαφανίζεται όταν τελειώνει τη δουλειά της. Η *a* ως στατική συνεχίζει να υπάρχει και μετά το τέλος της εκτέλεσης της *rnd*01 και έτσι δεν χάνει την τιμή της.

Οι στατικές μεταβλητές έχουν και μια άλλη ιδιότητα: αν δεν τους δοθεί αρχική τιμή με τη δήλωση παίρνουν την *ερήμην καθορισμένη* (default) τιμή του τύπου τους. Για όλους τους αριθμητικούς τύπους της C++ και για τα βέλη είναι η τιμή 0 (μηδέν).

13.6.2 Καθολικά Αντικείμενα και Τεκμηρίωση

Απόφευγε όσο μπορείς τις καθολικές μεταβλητές. Αν τις χρησιμοποιείς θα γράφεις προγράμματα που δεν είναι ευκολο να

- Επαληθευθούν
- Τροποποιηθούν.

Πάντως αυτό δεν θα είναι πάντοτε εύκολο. Διάβασε λοιπόν αυτά που λέμε παρακάτω για τις συναρτήσεις και τις μεταβλητές.

Οι συναρτήσεις είναι *καθολικά* αντικείμενα. Έτσι, κάθε συνάρτηση μπορεί να καλεί άλλες συναρτήσεις και να καλείται από άλλες συναρτήσεις. Σκέψου λοιπόν το πρόβλημα που έχεις όταν χρειάζεται να τροποποιήσεις κάποια συνάρτηση. Θα πρέπει να πας σε όλες τις συναρτήσεις που την καλούν και να δεις τι πρόβλημα θα δημιουργηθεί.

Καταλαβαίνεις λοιπόν ότι: ουσιώδες τμήμα της τεκμηρίωσης ενός προγράμματος είναι η καταγραφή των αλληλεξαρτήσεων των συναρτήσεων.

♦ ***Για κάθε συνάρτηση θα πρέπει να είναι καταγραμμένο: από ποιες καλείται και ποιες καλεί.***

Αν έλθουμε τώρα στις καθολικές μεταβλητές, ο αντίστοιχος κανόνας τεκμηρίωσης είναι ο εξής:

♦ ***Για κάθε καθολική μεταβλητή θα πρέπει να είναι καταγραμμένο: ποιες συναρτήσεις τη χρησιμοποιούν και ποιες συναρτήσεις αλλάζουν την τιμή της.***

13.7 * Οι Συναρτήσεις στις Αποδείξεις (ξανά)

Στην §7.9 είχαμε δει το τι και πώς πρέπει να αποδείξουμε όταν έχουμε μια συνάρτηση με τύπο με παραμέτρους τιμής μόνον. Τώρα θα δούμε τα ίδια πράγματα για την περίπτωση που έχουμε συνάρτηση χωρίς τύπο (`void`) με παραμέτρους τιμής και αναφοράς.

Ξεκινούμε με δύο παραδείγματα:

Παράδειγμα 1

Ας πάρουμε τη *minmax()* που τροφοδοτείται με δύο ακέραιους *x*, *y* και μας επιστρέφει τον ελάχιστο και τον μέγιστο από τους δύο:

```
   void minmax( int x, int y, int& mnxy, int& mxxy )
   {
      if ( x < y )
      {
         mnxy = x;  mxxy = y;
      }
      else
      {
         mnxy = y;  mxxy = x;
      }
   } // minmax
```

Αν τώρα βάλουμε σε μια συνάρτηση την εντολή:

```
   minmax( a, b, p, q );
```

τι περιμένουμε; Όποιες και να είναι οι *a*, *b* αρχικά, θα πρέπει μετά από αυτήν να έχουμε την τιμή της μικρότερης από αυτές στην *p* και την τιμή της μεγαλύτερης στην *q*. Να λοιπόν οι προδιαγραφές της *minmax()*:

```
// true
   minmax( a, b, p, q );
// ((p == a && q == b) || (p == b && q == a)) &&
// (p <= a && p <= b) && (q >= a && q >= b)
```

Όπως ξέρουμε, στις τυπικές παραμέτρους τιμής, *x*, *y*, θα αντιγραφούν οι τιμές των πραγματικών παραμέτρων *a*, *b*, ενώ οι *mnxy*, *mxxy* αναφέρονται στις ίδιες θέσεις της μνήμης με τις *p*, *q*. Άρα, μέσα στη συνάρτηση θα πρέπει να αποδείξουμε ότι:

```
// true
   if (x < y)
   {
      mnxy = x;  mxxy = y;
   }
   else
   {
      mnxy = y;  mxxy = x;
   }
// ((mnxy == x && mxxy == y) || (mnxy == y && mxxy == x)) &&
// (mnxy <= x && mnxy <= y) && (mxxy >= x && mxxy >= y)
```

Αυτό δεν διαφέρει και πολύ από αυτό που είχαμε να αποδείξουμε όταν είχαμε συνάρτηση με τύπο. Απλώς εδώ υπολογίζουμε δύο τιμές, αντί για μία.

Παράδειγμα 2

Ας δούμε τώρα τη *swap()*, που δέχεται δύο ορίσματα (τη γράφουμε για τον τύπο `int`) και αντιμεταθέτει τις τιμές τους:

```
   void swap( int& x, int& y )
   {
      int s( x );
      x = y;  y = s;
   } // swap
```

Ποιες είναι οι προδιαγραφές της *swap()*; Μπορούμε να τις διατυπώσουμε εύκολα (ρίξε μια ματιά και στο Κεφ. 3):

```
// a == a0 && b == b0
   swap( a, b );
// a == b0 && b == a0
```

Επομένως στη συνάρτηση θα πρέπει να αποδείξουμε:

```
// x == a0 && y == b0
   int s( x );  x = y;  y = s;
// x == b0 && y == a0
```

☜☜☜

Συγκρίνοντας τα δύο παραδείγματα βλέπουμε ότι όταν έχουμε εισερχόμενες και εξερχόμενες τιμές μπορούμε να γράψουμε προδιαγραφές με τα ονόματα των παραμέτρων μόνον. Όταν όμως έχουμε μεταβαλόμενες τιμές θα πρέπει να χρησιμοποιήσουμε στις προδιαγραφές μας και τις αρχικές και τελικές τιμές των παραμέτρων αναφοράς.

Ας υποθέσουμε ότι έχουμε μια συνάρτηση, `void p`, με *n* παραμέτρους τιμής, *v1*, *v2*,... *vn* και *m* παραμέτρους αναφοράς *r1*, *r2*,... *rm*. Αν ονομάσουμε *r1α*, *r2α*,... *rmα* τις αρχικές τιμές των παραμέτρων αναφοράς και *r1τ*, *r2τ*,... *rmτ* τις τελικές τιμές τους, οι προδιαγραφές της *p* μπορεί να διατυπωθούν ως εξής:

```
// Pd(p1, p2,... pn, q1, q2,... qm) &&
// (q1 == r1α) && (q2 == r2α) && ... (qm == rmα)
   p(v1, v2,... vn, r1, r2,... rm);
// Qd(p1, p2,... pn, r1α, r2α,... rmα, q1τ, q2τ,... qmτ)
```

Όταν την καλούμε, την τροφοδοτούμε με τις τιμές που έχουν οι παράμετροι τιμής και πιθανότατα κάποιες από τις παραμέτρους αναφοράς. Αν υποθέσουμε ότι οι τιμές των παραμέτρων τιμής δεν αλλάζουν, μπορούμε να γράψουμε:

```
   void p(T1 v1, T2 v2,... Tn vn, Tr1& r1, Tr2& r2,... Trm& rm)
   {
   // Pd(v1, v2,... vn, r1, r2,... rm) &&
   // (r1 == r1α) && (r2 == r2α) && ... (rm == rmα)
         :
   // Qd(v1, v2,... vn, r1α, r2α,... rmα, r1, r2,... rm)
   } // p
```

13.8 Ορμαθοί C και Αριθμοί (ξανά)

Πριν προχωρήσουμε παρακάτω, θα κάνουμε μια παρένθεση για να «εξοφλήσουμε κάποια παλιά χρέη». Στην §10.13.1 είχαμε πει ότι για μετατροπές ορμαθών σε αριθμούς «*πιο πλήρη δουλειά, σε περίπτωση λάθους, κάνουν οι strtod() και strtol() που θα δούμε αργότερα.*» Τώρα μπορούμε να τις δούμε.

Ας πούμε ότι δίνουμε:

```
   d = strtod( s3, &p );
```

όπου *s3* αριθμητικός ορμαθός και "`char* p`". Αν ολόκληρος ο ορμαθός *s3* μετατραπεί σε ακέραιο, αυτός θα γίνει τιμή της *d* και η τιμή της *p* θα είναι (βέλος) 0. Αν βρεθούν απαράδεκτοι χαρακτήρες το βέλος *p* θα δείχνει έναν (υπο) ορμαθό που ξεκινάει με τον πρώτο από αυτούς.

Ας πούμε ότι έχουμε δηλώσει:

```
   double d;
   char  s1[] = "12345", s2[] = "1.23456789e10",
         s3[] = "1.23ab45678";
   char* p;
```

Οι παρακάτω εντολές:

```
   d = strtod( s2, &p );
   if ( p != 0 )  cout << d << "  " << p << endl;
            else  cout << d << endl;
   d = strtod( s3, &p );
   if ( p != 0 )  cout << d << "  " << p << endl;
```

```
            else  cout << d << endl;
```

θα δώσουν:

```
1.23457e+010
1.23  ab45678
```

Πώς είναι δηλωμένη η *strtod()* (στο **cstdlib**); Έτσι:

```
double strtod( const char* s, char** endptr );
```

Δεν έχεις πρόβλημα να καταλάβεις την πρώτη παράμετρο: Βέλος προς πίνακα χαρακτήρων. Η δεύτερη όμως; Μέσω της δεύτερης η συνάρτηση θα επιστρέψει ένα βέλος, είτε 0 (**NULL**) είτε προς τον πρώτο παράνομο χαρακτήρα. Στην §13.2 είδαμε πώς επιστρέφει η C τιμές μέσω παραμέτρων: αν η τιμή είναι τύπου *T*, βάζουμε μια παράμετρο τύπου *T** και εκεί –όταν καλούμε τη συνάρτηση– αντιστοιχίζουμε μια διεύθυνση της μνήμης όπου θα αποθηκευτεί η επιστρεφόμενη τιμή. Στην περίπτωσή μας θα επιστραφεί μια τιμή τύπου **char***· θα πρέπει λοιπόν να βάλουμε μια παράμετρο τύπου **(char*)***. Όταν καλούμε τη συνάρτηση θέλουμε να αποθηκεύσουμε θέλουμε η επιστρεφόμενη τιμή να αποθηκευτεί στην *p* (που είναι τύπου **char***). Δίνουμε λοιπόν την διεύθυνση της *p*, δηλαδή **&p**.

Αν κατάλαβες αυτό το σημείο, τα υπόλοιπα είναι απλά.

Σαν τη *strtod()* είναι και οι:

```
float strtof( const char* s, char** endptr );
long double strtold( const char* s, char** endptr );
```

Μπορείς να βλέπεις την *atof()* (§10.13.1) ως

```
double atof( const char* s )
{  return strtod( s, NULL );  }
```

Παρομοίως δουλεύει και η *strtol()*, που είναι δηλωμένη ως εξής:

```
long strtol( const char* s, char** endptr, int radix )
```

Όπως βλέπεις, αυτή έχει και μια τρίτη παράμετρο όπου πρέπει να βάλουμε τη βάση του αριθμητικού συστήματος στο οποίο είναι γραμμένος ο ακέραιος που παριστανει ο ορμαθός. Π.χ. οι:

```
   l = strtol( "11", &p,  2 );   cout << l << endl;
   l = strtol( "11", &p,  8 );   cout << l << endl;
   l = strtol( "11", &p, 10 );   cout << l << endl;
   l = strtol( "11", &p, 16 );   cout << l << endl;
```

δίνουν:

```
3
9
11
17
```

(3 = 2 + 1, 9 = 8 + 1, 11 = 10 + 1, 17 = 16 + 1)

Ακόμη, οι:

```
   long int l( strtol(s1, &p, 10) );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
   l = strtol( s2, &p, 10 );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
```

δίνουν:

```
12345
1  .23456789e10
```

Τα παραπάνω σου δείχνουν και τον τρόπο χρήσης των δύο συναρτήσεων: αφού τις καλέσεις ελέγχεις το *p*:

- Αν είναι 0 όλα πήγαν καλά, δηλαδή ολόκληρος ο ορμαθός μετατράπηκε στην αριθμητική τιμή που σου επίστρεψε τη συνάρτηση και μπορείς να τη χρησιμοποιήσεις.

- Αν δεν είναι 0 τότε δεν έγινε πλήρης μετατροπή διότι υπήρχε ένας τουλάχιστον παράνομος χαρακτήρας. Το *p* σου δείχνει τον πρώτο (`*p`).

Όλα καλά εκτός από την περίπτωση που έχουμε **υπερχείλιση** (overflow)! Τι γίνεται στην περίπτωση αυτή; Σου έρχεται ένα μήνυμα από την τιμή που επιστρέφει η συνάρτηση και την τιμή μιας καθολικής μεταβλητής με το όνομα "`errno`". Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cerrno>`".

- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μεγαλύτερη από `DBL_MAX` αυτή επιστρέφει τιμή `HUGE_VAL` (ορίζεται στο `cmath`) και βάζει στην *errno* τιμή `ERANGE` (ορίζεται στο `cerrno`).
- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μικρότερη από `-DBL_MAX` αυτή επιστρέφει τιμή `-HUGE_VAL` και βάζει στην *errno* τιμή `ERANGE`.[4]

Αν λοιπόν μετά τις

```
errno = 0;
d = strtod( s2, &p );
```

ισχύει η συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

δεν έχει νόημα να χρησιμοποιήσεις την τιμή της *d*.

Παρομοίως, αν μετά τις

```
errno = 0;
l = strtol(s2, &p, 10)
```

η συνθήκη

```
(l == LONG_MAX || l == LONG_MIN) && errno == ERANGE
```

έχει τιμή `true` μην χρησιμοποιήσεις την τιμής *l*.

Για την *errno* θα τα ξαναπούμε.

Παρόμοιες με τη *strtol()* είναι και οι

```
long long int strtoll( const char* s, char** endptr, int base );
unsigned long int strtoul( const char* s, char** endptr,
                                                    int base );
unsigned long long int strtoull( const char* s, char** endptr,
                                                    int base );
```

ενώ οι *atol()* (§10.13.1), *atoi()*, *atoll()* είναι ισοδύναμες με

```
long int atol( const char* s )
{  return strtol( s, NULL, 10 );  }
int atoi( const char* s )
{  return int( strtol(s, NULL, 10) );  }
long long int atoll( const char* s )
{  return strtoll( s, NULL, 10 );  }
```

13.9 Πώς Επιλέγουμε το Είδος της Συνάρτησης

Μάθαμε λοιπόν για συναρτήσεις με τύπο και χωρίς τύπο, για παραμέτρους βέλη, τιμής και αναφοράς, για καθολικές μεταβλητές... Πώς τα χρησιμοποιούμε όλα αυτά;

Παρακάτω δίνουμε μερικούς κανόνες για το πώς επιλέγουμε το είδος της συνάρτησης. Θα πρέπει να τηρείς αυτούς τους κανόνες μέχρι να γίνεις μεγάλος(-η) προγραμματιστής(-στρια)· τότε θα ξέρεις πότε και πώς να τους παραβιάζεις.

Πριν από όλα λοιπόν θα βάλουμε τον εξής κανόνα:

♦ ***Όποτε μπορούμε να γράψουμε συνάρτηση με τύπο θα γράφουμε συνάρτηση με τύπο.***

[4] Σε περίπτωση υπερχείλισης η *strtof* επιστρέφει *HUGE_VALF* ή *–HUGE_VALF* ενώ η *strtold* θα δώσει *HUGE_VALL* ή *–HUGE_VALL*.

Αυτός ο κανόνας μπορεί να θεωρηθεί ως ειδική περίπτωση ενός γενικότερου κανόνα που λέει:

- ***Κάθε συνάρτηση θα πρέπει να έχει τον ελάχιστο καλά καθορισμένο υπολογιστικό στόχο***

Δηλαδή, όχι «αφού γράφουμε μια συνάρτηση να τα βάλουμε όλα μέσα!» Στον κανόνα αυτόν θα επανέλθουμε αργότερα με παραδείγματα.

1ος Κανόνας:

- ***Κάθε συνάρτηση επικοινωνεί –με τη συνάρτηση που την καλεί– μόνο με το όνομά της και με τις παραμέτρους της.***

Με άλλα λόγια: *μη γράφεις συναρτήσεις που να επικοινωνούν μέσω καθολικών μεταβλητών*.

Όταν πρόκειται να γράψεις μια συνάρτηση θα πρέπει, πριν από όλα, να καταγράψεις:

- τις τιμές που θα εισάγονται σε αυτήν (εισερχόμενες),
- τις τιμές που θα εξάγονται από αυτήν (εξερχόμενες),
- τις τιμές που θα μεταβάλλονται από αυτήν (διπλής κατεύθυνσης).

Με βάση αυτήν την καταγραφή αποφασίζεις το είδος της συνάρτησης που θα γράψεις:

2ος Κανόνας:

- ***Αν η συνάρτηση δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή τότε γράφεις συνάρτηση με τύπο.***

Στην περίπτωση αυτήν η εξερχόμενη τιμή θα αποδίδεται με την κλήση (όνομα) της συνάρτησης και τύπος της συνάρτησης είναι ο τύπος της εξερχόμενης τιμής. Είναι φανερό ότι οποιαδήποτε συνάρτηση με τύπο μπορείς να την υλοποιήσεις και ως συνάρτηση χωρίς τύπο με μια εξερχόμενη τιμή. Αλλά η συνάρτηση με τύπο είναι σαφώς πιο εύχρηστη και επομένως προτιμότερη.

3ος Κανόνας:

- ***Αν η συνάρτηση έχει μια ή περισσότερες μεταβαλλόμενες τιμές ή περισσότερες από μία εξερχόμενες τιμές τότε γράφεις συνάρτηση χωρίς τύπο*** `(void)`.

Στην περίπτωση αυτή οι μεταβαλλόμενες τιμές επιστρέφουν μέσω παραμέτρων. Ο κανόνας αυτός είναι συμπλήρωση του 2ου κανόνα· μια διαφορετική διατύπωση είναι η εξής:

- ***Μη γράφεις συνάρτηση με τύπο με παραμέτρους αναφοράς που να υλοποιούν επικοινωνία για μεταβαλλόμενες τιμές ή εξερχόμενες τιμές.***

13.9.1 Περί Παραμέτρων

Ας έλθουμε τώρα στις παραμέτρους. Άλλες γλώσσες προγραμματισμού επιτρέπουν προσδιορισμό των παραμέτρων σχετικό με τη χρήση τους. Η Ada, για παράδειγμα, έχει παραμέτρους, όπως τις δώσαμε πιο πάνω:

- **in** για εισαγωγή δεδομένων στη συνάρτηση,
- **out** για εξαγωγή τιμών από τη συνάρτηση και
- **in out** για τιμές που μεταβάλλονται από τη συνάρτηση.

Αξίζει να επισημάνουμε εδώ ότι η Ada στα υποπρογράμματα που αντιστοιχούν στις συναρτήσεις με τύπο της C++ επιτρέπει παραμέτρους **in** μόνον![5]

[5] Σου θυμίζει αυτό κάτι από τους κανόνες μας;...

Η C έχει ένα είδος παραμέτρων: *παραμέτρους τιμής*. Με αυτές περνάς δεδομένα προς τη συνάρτηση. Αν τώρα θέλεις να πάρεις τιμές από τη συνάρτηση μέσω παραμέτρων εισάγεις προς αυτήν ένα βέλος που δείχνει τη διεύθυνση όπου περιμένεις το «εξαγόμενο».

Η C++ έκανε ένα βήμα σε σχέση με τη C αλλά όχι μεγάλο: μετονόμασε το βέλος σε *αναφορά* και μας έδωσε την ευκολία να το χειριζόμαστε χωρίς να κάνουμε αποπαραπομπή, που γίνεται αυτομάτως. Παρ' όλα αυτά ο προσδιορισμός των παραμέτρων έχει σχέση με την υλοποίηση:

1. **Παράμετροι τιμής** (value parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση χωρίς να επιστρέφουν τις οποιεσδήποτε αλλαγές στη συνάρτηση που καλεί. Με αυτές υλοποιούμε παραμέτρους για εισερχόμενες τιμές (**in**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο τιμής μπορεί να είναι γενικά μια παράσταση.
2. **Παράμετροι αναφοράς** (reference parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση και γνωστοποιούν στη συνάρτηση που καλεί τις όποιες αλλαγές έγιναν στην καλούμενη συνάρτηση. Με αυτές υλοποιούμε παραμέτρους για εξερχόμενες (**out**) και μεταβαλλόμενες τιμές (**in out**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο αναφοράς μπορεί να είναι μεταβλητή ή στοιχείο πίνακα ή πίνακας (αν η τυπική παράμετρος είναι πίνακας).
3. **Παράμετροι-πίνακες** που είναι μεν παράμετροι τιμής αλλά είναι παράμετροι-βέλη. Επομένως, όπως έχουμε δει στην §9.4, είναι παράμετροι **in out**. Τις βάζουμε χωριστά διότι μπορείς να τις χειρίζεσαι μέσα στη συνάρτηση ως πίνακες χωρίς αποπαραπομπές. Αν θέλεις να έχεις έναν πίνακα ως παράμετρο **in** προτάσσεις τον περιορισμό `const`.

Προς το παρόν τα μόνα «μεγάλα» αντικείμενα που έχουμε δει είναι οι πίνακες και ξέρουμε πώς να τους χειριστούμε. Στη συνέχεια, όταν μάθουμε για κλάσεις, θα δούμε και άλλα. Αν θέλουμε να περάσουμε ένα μεγάλο αντικείμενο (ας πούμε τύπου *T*) ως παράμετρο **in**, η παράμετρος τιμής δεν είναι καλή λύση· η διαδικασία της αντιγραφής καθυστερεί την εκτέλεση του προγράμματος και φορτώνει μια περιοχή της μνήμης (τη στοίβα) που είναι μάλλον περιορισμένη. Στην περίπτωση αυτήν προτιμούμε να χρησιμοποιήσουμε παράμετρο αναφοράς (*T*`&`) με τον περιορισμό `const` (`const` *T*`&`).

Οι προγραμματιστές που έγραφαν παλιότερα C προτιμούν να χρησιμοποιούν αντί για παραμέτρους αναφοράς **παραμέτρους βέλη** (pointers). Ένας λόγος για την προτίμησή τους είναι ότι όταν διαβάζει κάποιος μια *κλήση συνάρτησης* καταλαβαίνει αμέσως ποιες παράμετροι είναι εισερχόμενες και ποιες διπλής κατεύθυνσης.

13.9.2 Παράμετρος – Ρεύμα

Είπαμε πριν (§11.6) ότι οι εντολές εισόδου/εξόδου είναι εντολές-παραστάσεις (πράξεις). Ας προσπαθήσουμε τώρα να το καταλάβουμε.

Κατ' αρχάς να υπενθυμίσουμε ότι, όπως μάθαμε στο Μέρος Α (§8.2), ένα ρεύμα έχει έναν **ενταμιευτή** (buffer), δηλαδή μια περιοχή μνήμης, όπου γίνεται ενδιάμεση αποθήκευση των δεδομένων του αρχείου που διαβάζουμε (ή γράφουμε). Φυσικά, χρειαζόμαστε και έναν δείκτη που δείχνει σε ποια θέση του ενταμιευτή διαβάζουμε, άλλους δείκτες που μας λένε ποια περιοχή του αρχείου έχουμε στον ενταμιευτή κλπ. Κάθε πράξη εισόδου/εξόδου μεταβάλλει την κατάσταση του ενταμιευτή και ολόκληρου του ρεύματος και η επόμενη πράξη εισόδου/εξόδου γίνεται πάνω σε αυτήν τη νέα κατάσταση.

Δες πώς αντανακλώνται αυτά τα πράγματα στο πρόγραμμα με ένα παράδειγμα σε γνωστά πράγματα. Ας πούμε ότι έχουμε την εντολή (δήλωσε τις *x*, *r* όπως θέλεις και δώσε όποιες τιμές θέλεις):

```
cout << " x = " << x << "   r = " << r << endl;
```

Δοκιμάζουμε να τη γράψουμε με διαφορετικό τρόπο:

```
(((((cout << " x = ") << x) << "   r = ") << r) << endl);
```

Βλέπουμε ότι γίνεται δεκτή από τον μεταγλωττιστή και βγάζει το ίδιο αποτέλεσμα με την αρχική.

Για να δούμε όμως τι γίνεται εδώ: Ξέρουμε ότι η "`cout << " x = "`" σημαίνει «*στείλε στο ρεύμα cout την τιμή της παράστασης* `" x = "`.» Ωραία! Τότε όμως τι σημαίνει η:

`(cout << " x = ") << x`

Στείλε στο ρεύμα `(cout << " x = ")` *την τιμή της x;*

Ναι, ακριβώς αυτό σημαίνει. *Η "*`cout << " x = "`*" είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.

Όπως καταλαβαίνεις αυτό είναι απαραίτητο για να μπορούμε να γράφουμε σε μια εντολή εξόδου πολλές εξερχόμενες τιμές (παραστάσεις).

Και κάτι ακόμη: Το «*Στείλε στο ρεύμα* `(cout << " x = ")` *την τιμή της x*» πρέπει να διατυπωθεί ακριβέστερα: *Στείλε στο ρεύμα "*`cout << " x = "`*" όπως έχει γίνει μετά την έξοδο της* `" x = "` *, την τιμή της x.*

Παίρνοντας υπόψη μας τα παραπάνω ας σκεφτούμε την εξής περίπτωση: Έχουμε δηλώσει στη **main**

```
ofstream tout( "afile.txt" );
```

και στη συνέχεια καλούμε μια συνάρτηση με παράμετρο *tout*:

```
afunction( tout, . . . );
```

Αν η *tout* περάσει ως παράμετρος τιμής, στο εσωτερικό της συνάρτησης θα δουλεύουμε με ένα αντίγραφο του ρεύματος ενώ η κατάσταση του *tout* της **main** δεν μεταβάλλεται. Όταν ο έλεγχος της εκτέλεσης επιστρέψει στη **main** το *tout* δεν θα μάθει αυτά που έγιναν στην *afunction*!

Επομένως; Η λύση είναι μία:

- ***Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* in out *(αναφοράς).***

  Στην περίπτωσή μας θα πρέπει να έχουμε:

```
void afunction( ofstream& tout, . . . )  { . . . }
```

  Και κάτι ακόμη:

- Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο *cout* βάζε παράμετρο τύπου `ostream` και όχι `ofstream`.
- Παρομοίως, αν θέλεις να περνάς σε μια συνάρτηση είτε το *cin* είτε κάποιο ρεύμα κλάσης `ifstream` βάζε παράμετρο κλάσης *istream*.

Γιατί; Ξαναδιάβασε την §8.1 και πάρε υπόψη σου ότι –όπως θα μάθουμε αργότερα– όπου μπορεί να εμφανιστεί αντικείμενο μιας κλάσης μπορεί να εμφανιστεί και αντικείμενο των κληρονόμων της.

Με βάση αυτά που είπαμε, μια συνάρτηση για είσοδο ή έξοδο στοιχείων έχει πάντοτε μια παράμετρο για μεταβαλλόμενη τιμή: το ρεύμα. Θα πρέπει λοιπόν να είναι συνάρτηση `void`.

13.9.3 Παραδείγματα

Ας δούμε μερικά παραδείγματα εφαρμογής των παραπάνω κανόνων ξεκινώντας από αυτά της προηγούμενης παραγράφου.

Παράδειγμα 1

Θέλουμε μια συνάρτηση *minmax* που θα τροφοδοτείται με δύο ακέραιους *x*, *y* και θα μας επιστρέφει τον ελάχιστο και τον μέγιστο από τους δύο.

Η συνάρτησή μας:

- θα παίρνει (**in**) δύο τιμές, τις *x*, *y* και

- θα επιστρέφει (**out**) δύο τιμές, τις *mnxy*, *mxxy*.

Αφού η συνάρτησή μας έχει δύο εξερχόμενες τιμές, σύμφωνα με τον Κανόνα 3 θα πρέπει να είναι συνάρτηση χωρίς τύπο.

Τι παραμέτρους θα έχει;

- δύο παραμέτρους τιμής, τις *x*, *y* και
- δύο παραμέτρους αναφοράς, τις *mnxy*, *mxxy*.

Να λοιπόν η συνάρτηση της:

```
void minmax( int x, int y, int& mnxy, int& mxxy )
{
   if ( x < y )
   {  mnxy = x;  mxxy = y;  }
   else // x >= y
   {  mnxy = y;  mxxy = x;  }
} // minmax
```

Ας πούμε τώρα ότι δεν μας ζητούν να γράψουμε τη συγκεκριμένη συνάρτηση αλλά να επιλέξουμε τι είδους συνάρτηση ή συναρτήσεις θα γραφούν.

Πρέπει να επιλέξουμε μεταξύ της *minmax()* και των:

```
int min( int x, int y )
{
   int fv( x );
   if ( x > y )  fv = y;
   return fv;
} // min
int max( int x, int y )
{
   int fv( x );
   if ( y > x )  fv = y;
   return fv;
} // max
```

Η *minmax()* προφανώς παραβιάζει τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά αν το πρόγραμμά μας χρειάζεται να υπολογιζει κατ' επανάληψη και το ελάχιστο και το μέγιστο τότε κάθε φορά έχουμε:

- Η *minmax()* θα καλείται ως εξής:

  ```
  minmax( a, b, mnab, mxab );
  ```

  και θα έχουμε:
  - μια κλήση συνάρτησης,
  - μια σύγκριση,
  - δύο εκχωρήσεις.
- Οι *min()* και *max()* θα καλούνται ως εξής:

  ```
  mnab = min( a, b );  mxab = max( a, b );
  ```

  και θα έχουμε:
  - δύο κλήσεις συναρτήσεων,
  - δύο συγκρίσεις,
  - δύο δηλώσεις μεταβλητών με αρχική τιμή (ορισμούς),
  - μια εκχώρηση,
  - δύο επιστροφές τιμών.

Προφανώς, για την περίπτωση αυτή, η *minmax()* είναι προτιμότερη.

Παράδειγμα 2

Θέλουμε μια συνάρτηση (με όνομα *swap*) που θα παίρνει δύο μεταβλητές τύπου `int` και θα αντιμεταθέτει τις τιμές τους.

Στην περίπτωση αυτή θέλουμε μια συνάρτηση που θα *μεταβάλλει* τις τιμές δύο μεταβλητών (**in out**). Κατά τον 3ο Κανόνα θα γράψουμε μια συνάρτηση χωρίς τύπο.

```
void swap( int& x, int& y )
{
   int s( x );
   x = y;  y = s;
} // swap
```

Παράδειγμα 3

Θέλουμε μια συνάρτηση, με όνομα *cntInt()*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου **double** και το πλήθος *n* των στοιχείων του και θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή.

Η *cntInt()*:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και
- με «*το πλήθος n των στοιχείων του*»,
- «*θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή*».

Δηλαδή, τροφοδοτείται με έναν πίνακα –βέλος και πλήθος στοιχείων– (**in**) και επιστρέφει μία ακέραιη τιμή.

Αφού η συνάρτησή μας «*δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή*» θα γράψουμε *συνάρτηση με τύπο*. Γράφουμε λοιπόν:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
```

Η εξερχόμενη τιμή θα επιστρέφει μέσω του ονόματος της συνάρτησης στην κλήση της.

Ολόκληρη η συνάρτηση[6]:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
{
   int m( 0 );

   for ( int k(0); k < n; ++k )
      if (a[k] == static_cast<long int>(a[k]))  ++m;
   return m;
} // cntInt
```

Παράδειγμα 4

Θέλουμε μια συνάρτηση, με όνομα *to1Dgt*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου **double**, και το πλήθος *n* των στοιχείων του και θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους.

Η συνάρτηση:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και «*θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους*»· δηλαδή οι τιμές των στοιχείων του πίνακα θα *μεταβάλλονται* από τη συνάρτηση.

  Ακόμη, η συνάρτηση

- «[*θα τροφοδοτείται*] *με το πλήθος n των στοιχείων του* [*πίνακα*]». Η τιμή της *n* είναι εισερχόμενη στη συνάρτηση.

Αφού η συνάρτησή μας «*έχει μεταβαλλόμενες τιμές*» (πίνακας a) θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο (**void**). Γράφουμε λοιπόν:

```
void to1Dgt( double a[],     // μεταβαλλόμενη παράμετρος
```

[6] Προφανώς, ο έλεγχος `a[k] == static_cast<long int>(a[k])` για ακέραιη τιμή δεν δουλεύει όταν η a[*k*] έχει πολύ μεγάλη τιμή.

```
                int n )          // εισερχόμενη παράμετρος
```

Τα στοιχεία του a έχουν τις μεταβαλλόμενες τιμές.

Ολόκληρη η συνάρτηση:

```
void to1Dgt( double a[], int n )
{
   for (int k(0); k < n; ++k )
      a[k] = (static_cast<long int>(a[k]*10))/10.0;
} // to1Dgt
```

&

Παράδειγμα 5

Στο Παράδ. 2 της §7.7 γράψαμε δύο συναρτήσεις την *gcd()* –που υπολογίζει τον ΜΚΔ δύο ακεραίων– και την *lcm()* που υπολογίζει το ΕΚΠ. Η *lcm()* καλεί την *gcd()*.

Αν έχουμε να επιλέξουμε ποιες και τι είδους συναρτήσεις θα γράψουμε:

- Σε πλήρη συμφωνία με όλους τους κανόνες μας θα γράψουμε την

```
unsigned int gcd( int x, int y )
```

- Η *lcm()* που γράψαμε μπορεί να γραφεί ως εξής:

```
void gcdLcm( int x, int y, int& gcdP, int& lcmP )
{                                 // ΕΚΠ(x,y)*ΜΚΔ(x,y) = x*y
   if ( (x == 0 && y == 0) || x < 0 || y < 0 )
   {
      cout << " η gcdLcm κλήθηκε με "
           << x << ", " << y << endl;
      exit( EXIT_FAILURE );
   }
// (x != 0 || y != 0) && x >= 0 && y >= 0
   gcdP = gcd(x, y);
   lcmP = x * y / gcdP;
} // gcdLcm
```

Εδώ, προφανώς, παραβιάζουμε τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά δεν κάνουμε τους ίδιους υπολογισμούς δύο φορές.

Πρόσεξε ότι ο ΜΚΔ υπολογίζεται σε ένα μέρος μόνον: στη *gcd()*. Για βελτιώσεις ή προβλήματα που μπορεί να παρουσιασθούν παρεμβαίνουμε μόνον εκεί.

- Και την *lcm()*, θα την πετάξουμε; Ε, μη την πετάξεις. Αν έχεις κάποια εφαρμογή που θέλει μόνον το ΕΚΠ χρησιμοποίησέ την.

Μια παρόμοια περίπτωση είναι και αυτή του Παραδ. 2 της §9.4: Οι συναρτήσεις

```
double vectorAvrg( const double x[], int n, int from, int upto)
double stdDev( const double x[], int n, int from, int upto )
```

υπολογίζουν τη μέση τιμή και την τυπική απόκλιση των `x[`*from*`]..x[`*upto*`]` αντιστοίχως. Η *stdDev()* καλεί τη *vectorAvrg()*.

Εκεί όμως έχουμε και κάτι άλλο: μπορούμε να γράψουμε μια:

```
void simpleStat ( const double x[], int n, int from, int upto,
                  double& avrgP, double& stdDevP )
```

που υπολογίζει και τα δύο μεγέθη με πιο αποδοτικό τρόπο.

Το τι γράφουμε, τι όχι και πότε τα χρησιμοποιούμε τα αφήνουμε ως άσκηση για σένα.

&

Παράδειγμα 6

Στο Παράδ. 4 της §12.4 (πολλαπλασιασμός πινάκων) γράφουμε δυο φορές τις ίδιες εντολές για να διαβάσουμε τους δυο πίνακες και τρεις φορές τις ίδιες εντολές για να γράψουμε τους τρεις πίνακες. Δεν θα μπορούσαμε να ευκολύνουμε τη ζωή μας με δύο συναρτήσεις; Φυσικά!

Κατ' αρχάς, αφού οι δύο συναρτήσεις κάνουν είσοδο/έξοδο στοιχείων θα πρέπει να είναι `void`:

```
void input2DAr( istream& tin, . . .
```

```
void output2DAr( ostream& tout, . . .
```

Και οι δύο συναρτήσεις έχουν ως παράμετρο τον πίνακα: η πρώτη ως παράμετρο **out** και η δεύτερη ως παράμετρο **in**. Για τη δεύτερη τα πράγματα είναι απλά:

```
void output2DAr( ostream& tout,
                 const int* a, int nRow, int nCol )
```

Για την πρώτη πώς θα είναι οι παράμετροι; Έτσι:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
```

Οι διαστάσεις των πινάκων δεν διαβάζονται από το αρχείο· είναι γνωστές στο πρόγραμμα που καλεί τη συνάρτηση γι' αυτό και δεν βάζουμε "`int& nRow, int& nCol`". Αργότερα θα δούμε και περιπτώσεις όπου τα πάντα είναι **out**.

Ολόκληρη η *input2DAr*:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
         tin >> a[r*nCol+c];
//          tin >> a[r][c];
   }
} // input2DAr
```

Για τούς πίνακες:

```
int  a[ l ][ m ],  b[ m ][ n ];
```

την καλούμε ως εξής:

```
ifstream atx( "arr.txt" );
input2DAr( atx, &a[0][0], l, m );
input2DAr( atx, &b[0][0], m, n );
atx.close();
```

Αφήνουμε ως άσκηση την *output2DAr*.

Παρατήρηση: ▶

Ωραίες οι συνταγές, αλλά εδώ βλέπουμε συναρτήσεις της πάγιας βιβλιοθήκης της γλώσσας που δεν συμμορφώνονται με αυτές.

- Στην προηγούμενη παράγραφο οι *strtod()* και *strtol()* επιστρέφουν τιμή (`double` ή `long int` αντιστοίχως) αλλά βγάζουν και ένα βέλος μέσω της παραμέτρου *endptr*.
- Στο προηγούμενο κεφάλαιο προσπαθώντας να μιμηθούμε τη λειτουργία των *strlen()* και *strcat()* είδαμε ότι η δεύτερη:
  - αλλάζει την πρώτη παράμετρο,
  - επιστρέφει και τιμή-βέλος.

Τι συμβαίνει με αυτές τις παρανομίες;

Οι απαντήσεις στα ερωτήματα αυτά είναι οι εξής:

- Οι συναρτήσεις αυτές έχουν σχεδιασθεί πολύ παλιά (έρχονται από τη C). Ο τρόπος που σκέφτονταν τότε οι προγραμματιστές ήταν αρκετά διαφορετικός και το βασικό κριτήριο για να πάρουν τις αποφάσεις τους είχε να κάνει με τους περιορισμούς που έβαζαν οι ΗΥ εκείνου του καιρού. Τώρα τα πράγματα είναι αρκετά διαφορετικά.
- Και εσυ, στο μέλλον, πιθανότατα, θα παραβείς αυτούς τους κανόνες. Αλλά να το κάνεις όταν θα έχεις γίνει προγραμματιστής(-στρια) σαν τον D.M. Ritchie ή τον B.W. Kernighan, γιατί τότε θα ξέρεις πότε και πώς να τους παραβείς. Μέχρι τότε, το καλύτερο που έχεις να κάνεις είναι να τους τηρείς.◀

13.10 Υποδείγματα Συναρτήσεων

Έστω ότι θέλουμε να γράψουμε ένα πρόγραμμα που i) θα διαβάζει από το πληκτρολόγιο τις τιμές των στοιχείων ενός μονοδιάστατου πίνακα (με 5 στοιχεία), ii) θα βρίσκει το πλήθος των στοιχείων του με ακέραιη τιμή, iii) θα αποκόπτει τις τιμές τους στο 1ο δεκαδικό ψηφίο και iv) θα τις γράφει στην οθόνη.

Μπορούμε να το γράψουμε χρησιμοποιώντας τις συναρτήσεις που είδαμε στα παραδ. 3 και 4 της προηγούμενης παραγράφου:

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

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main
```

Οι συναρτήσεις έχουν γραφεί έτσι ώστε να μην προκαλούν προβλήματα κατά τη μεταγλώττιση. Δηλαδή, γράψαμε τελευταία τη **main** που έχει τις αναφορές στις δύο συναρτήσεις, ώστε όταν ο μεταγλωττιστής βρει τα ονόματα των συναρτήσεων να τις ξέρει ήδη.

Όταν γράφεις ένα μεγάλο πρόγραμμα κάτι τέτοιο δεν είναι πάντοτε εφικτό ούτε βολικό. Πολύ συχνά το πρόγραμμά μας χρησιμοποιεί συναρτήσεις που δεν γράφονται μαζί με αυτό αλλά υπάρχουν σε διάφορες βιβλιοθήκες και συνδέονται στο πρόγραμμα μετά τη μεταγλώττισή του. Όπως λέγαμε και στην §7.4, η C++ μας επιτρέπει να γράφουμε τις συναρτήσεις σε οποιαδήποτε σειρά (ή και να μην τις γράφουμε καθόλου) αρκεί να βάζουμε πιο πριν **υποδείγματα** (prototypes) των συναρτήσεων.

Το υπόδειγμα μιας συνάρτησης περιέχει τον τύπο του αποτελέσματος, το όνομα της συνάρτησης και –μέσα σε παρενθέσεις– τους τύπους των τυπικών παραμέτρων. Πάντως μπορείς να χρησιμοποιήσεις και την επικεφαλίδα της τερματιζόμενη με ένα ";". Για τις συναρτήσεις του παραπάνω παραδείγματος θα μπορούσαμε να έχουμε:

```
   int cntInt( const double[], int );
   void to1Dgt( double[], int );
```

ή

```
   int cntInt( const double*, int );
   void to1Dgt( double*, int );
```

ή

```
   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );
```

Έτσι, το πρόγραμμά μας θα μπορούσε να γραφεί ως εξής:

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

   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt
```

Πολύ συχνά, κυρίως όταν έχουμε μεγάλα προγράμματα, οι επικεφαλίδες μαζί με διάφορες δηλώσεις μπαίνουν σε ένα ξεχωριστό αρχείο που περιλαμβάνεται (**#include**) στην αρχή του αρχείου του προγράμματος. Π.χ. αν το πρόγραμμά μας βρίσκεται στο αρχείο **truncarr.cpp** θα βάλουμε, κατά τη συνήθεια που επικρατεί, τις επικεφαλίδες σε ένα αρχείο με όνομα: **truncarr.h** και η αρχή του προγράμματός μας θα είναι:

```
#include <iostream>
#include <cmath>

#include "truncarr.h"

using namespace std;

int main()
   . . .
```

13.11 Ένα Δύσκολο Πρόβλημα!

Και τώρα θα γράψουμε πρόγραμμα! Μάθαμε αρκετά ώστε να μπορούμε να αντιμετωπίσουμε ένα πρόβλημα που θα μπορούσε να υπάρχει στον πραγματικό κόσμο.

Ένα συνεργείο έκανε μέτρηση ροής οχημάτων σε κάποιο δρόμο. Σε αρχείο, text – με το όνομα στο δίσκο **autoflow.txt***– καταγράφηκαν οι τιμές ροής σε οχήματα/min κάθε λεπτό. Το πλήθος των τιμών στο αρχείο είναι άγνωστο, αλλά σίγουρα θετικό.*

Οι πρώτες τρεις γραμμές του αρχείου έχουν την «ταυτότητα» της μέτρησης ως εξής:

```
Υπεύθυνος:\t<όνομα>\t<επώνυμο>
Σημείο Μετρήσεων:\t<οδός-αριθμός>\t<περιοχή>\t<δήμος>
Αρχή:\tdd.mm.yyyy\thh:mm
```

Στις υπόλοιπες γραμμές δίνονται οι τιμές από τη μέτρηση, μια σε κάθε γραμμή. Κάθε τιμή είναι γραμμένη στις πέντε πρώτες θέσεις. Πέρα από τη δέκατη θέση μπορεί να υπάρχουν σχόλια που περιγράφουν συμβάντα κατά τη διάρκεια της μέτρησης. Να ένα παράδειγμα:

```
Υπεύθυνος:   Ανδρέας      Νικολόπουλος
Σημείο Μετρήσεων: Τζαβέλλα 48    Νεάπολις     Αθήνα
Αρχή:  15.06.2009   05:00
   26
   30
    8
   28
. . .
   17
   19
    4
   12     / Αρχή λειτουργίας σηματοδότησης
   28
   17
. . .
   17
   31     / Βλάβη σηματοδότη Τζαβέλλα & Γιωργάκου
   23
   24
. . .
```

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει:

1. Το πλήθος τιμών του αρχείου καθώς και τη διάρκεια των μετρήσεων σε h και min.

2. Το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων.

3. Τη μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min.

Όλα αυτά θα γράφονται στην τελική έκθεση, που θα γραφεί σε αρχείο με το όνομα `report.txt`*. Στις πρώτες τρεις γραμμές του αρχείου θα αντιγραφούν οι τρεις πρώτες γραμμές του* `autoflow.txt`*.*

Ένα από τα ζητούμενα της δουλειάς είναι και η μελέτη της μεταβολής της ροής οχημάτων. Ως πρώτο βήμα μας ζητείται να δημιουργήσουμε ένα άλλο αρχείο με τις μεταβολές ροής ανά min.

Σαν πολλά δε ζητάει; Μπα, τα περισσότερα τα ξέρουμε! Ας ξεκινήσουμε με τα αρχεία. Οι προδιαγραφές λένε ότι θα διαβάζουμε ένα αρχείο και θα γράφουμε δύο. Όλα τα αρχεία θα είναι text.

Το πρώτο που θα κάνουμε είναι να διαμορφώσουμε τις προδιαγραφές. Αν $numberPerMin_k$, k: $1..n$ είναι η ακολουθία τιμών που διαβάζουμε από το `autoflow.txt` τότε στο ένα από τα αρχεία θα γράψουμε τις τιμές μιας νέας ακολουθίας, της

$$difference_k = numberPerMin_k - numberPerMin_{k-1},\ k: 2..n$$

«*Το πλήθος τιμών του αρχείου*» είναι το *n*. Αφού οι μετρήσεις λαμβάνονται ανά *min*, η «*διάρκεια των μετρήσεων*» σε *min* είναι *n*. Για να τη μετατρέψουμε «*σε h και min*» παίρνουμε το πηλίκο και το υπόλοιπο της *n* δια 60:

$$hours = n \ / \ 60, \quad minutes = n \ \% \ 60$$

Αφού κάθε $numberPerMin_k$ που διαβάζουμε είναι το πλήθος των αυτοκινήτων που περνούν σε ένα *min*, «*το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων*» είναι προφανώς το άθροισμα των τιμών που διαβάζονται:

$$numberOfCars = \sum_{k=1}^{n} numberPerMin_k .$$

Η «*μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min*» είναι η

$$averageFlow = \sum_{k=1}^{n} numberPerMin_k / n$$

Αυτή μπορεί να υπολογισθεί μόνον αν $n > 0$, αν δηλαδή το αρχείο που μας δίνεται έχει μια τουλάχιστον μέτρηση.

Να λοιπόν οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Απαίτηση: Στο αρχείο `differences.txt` γράφονται τα μέλη της ακολουθίας $difference_k$, k: `2..`n

Στο αρχείο `report.txt` γράφονται:

- οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των *n*, *hours* (`==` n `/ 60`), *minutes* (`==` n `%` 60), *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (`==` *numberOfCars* / n).

Ας κάνουμε τώρα ένα σχέδιο για το πρόγραμμά μας.

Στην αρχή θα πρέπει να ανοιχθούν οπωσδήποτε τα αρχεία `autoflow.txt`, για διάβασμα και το `report.txt` για να γράψουμε τις τρεις πρώτες γραμμές. Στο `differences.txt` θα γράψουμε αν από το `autoflow.txt` διαβάσουμε τουλάχιστον δύο τιμές ροής. Θα πρέπει λοιπόν να περιμένουμε και να το ανοίξουμε μόνον αν βρούμε δεύτερη τιμή; Όχι! Αφού μας ζητείται αρχείο διαφορών θα το δημιουργήσουμε ακόμη και αν το αφήσουμε κενό.

Στη συνέχεια θα αντιγράψουμε στο `report.txt` τις τρεις πρώτες γραμμές.

Μετά θα διαβάζουμε από το `autoflow.txt`, θα υπολογίζουμε και θα γράφουμε στο `differences.txt` μέχρι να τελειώσει το αρχείο. Αλλά εδώ χρειάζεται προσοχή: για την πρώτη τιμή δεν μπορούμε να υπολογίσουμε διαφορά· επομένως χρειάζεται ειδικό χειρισμό.

Όσο διαβάζουμε και υπολογίζουμε και γράφουμε διαφορές θα πρέπει να υπολογίζουμε το *n* (να μετρούμε) και το *numberOfCars* (να αθροίζουμε) ώστε να υπολογίσουμε και τη *averageFlow*.

Τέλος, υπολογίζουμε τη μέση τιμή, γράφουμε όσα χρειάζεται στο `report.txt` και κλείνουμε τα αρχεία.

Να λοιπόν ένα πρώτο σχέδιο του προγράμματός μας:

```
Άνοιξε τα ρεύματα των αρχείων
Επεξεργασία
Κλείσε τα ρεύματα
```

Αλλά, σαν να ξεχάσαμε κάτι εδώ: Θα προχωρήσουμε στις επεξεργασίες αν ανοίξουν τα ρεύματα, αλλιώς τι να επεξεργαστούμε; Το σχέδιο πρέπει να αλλάξει λίγο:

```
Άνοιξε τα ρεύματα των αρχείων
if ( άνοιξαν τα ρεύματα )
{
   Επεξεργασία
   Κλείσε τα ρεύματα
}
```

Στην §0.5 λέγαμε ότι «*το αρχικό πρόβλημα διασπάται σε δύο ή περισσότερα υποπροβλήματα*» από τα οποία το καθένα «*έχει προϋπόθεση την απαίτηση του προηγούμενου*». Ας βάλουμε λοιπόν και εδώ τις προδιαγραφές στα προβλήματα που προκύπτουν από τη διάσπαση:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

```
Άνοιξε τα ρεύματα των αρχείων
```

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής n και το πλήθος των οχημάτων *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$).

Υπολογίστηκαν τα *hours* (== n / 60), *minutes* (== n % 60).

Αν $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση) υπολογίστηκε η *averageFlow* (== *numberOfCars* / n).

Γράφηκαν στο `report.txt` τα n, *hours*, *minutes*, *numberOfCars*, *averageFlow*.

`Κλείσε τα ρεύματα`

Απαίτηση: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n

Στο αρχείο `report.txt` έχουν γραφεί:

- οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των n, *hours* (== n / 60), *minutes* (== n % 60), *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (== *numberOfCars* / n).

Κάπως έτσι γίνεται η ανάλυση των προδιαγραφών μέσα στο σχέδιό μας. Πρόσεξε το "$n > 0$" που μεταφέρεται από βήμα σε βήμα μέχρι το «`Τελικοί υπολογισμοί`» όπου και χρειάζεται για να μπορέσουμε να υπολογίσουμε τη μέση ροή.

13.11.1 «Άνοιξε τα ρεύματα των αρχείων»

Ξεκινούμε από την "`Άνοιξε τα ρεύματα των αρχείων`" και το πρώτο που πρέπει να δούμε είναι το τι μπορεί να συμβεί και να μην ανοίξουμε τα ρεύματα. Θα πεις «Για ποια ρεύματα συζητούμε; Το μόνο πρόβλημα που μπορεί να έχουμε είναι: το ρεύμα από το `autoflow.txt`, να μη βρει το αρχείο!». Ας δούμε μιαν άλλη λεπτομέρεια. Μέχρι τώρα ξέρουμε ότι αν απόπειραθούμε να δημιουργήσουμε ένα αρχείο που ήδη υπάρχει στον δίσκο, αυτομάτως σβήνεται το παλιό και δημιουργείται το καινούριο. Ε, λοιπόν: δεν θέλουμε –το πρόγραμμα που θα γράψουμε– να σβήσει, χωρίς προειδοποίηση, κάποιο αρχείο που ήδη υπάρχει με όνομα `report.txt` ή `differences.txt`.

Θα ανοίξουμε αυτά τα αρχεία με την παρακάτω συνάρτηση:

```
void openWrNoReplace( ofstream& newStream, string flNm, bool& ok )
{
   ifstream test( flNm.c_str() );           // άνοιξε για διάβασμα
   if ( test.fail() )                 // ok, δεν υπάρχει το αρχείο
   {
      newStream.open( flNm.c_str() );        // άνοιξε για γράψιμο
      ok = !newStream.fail();
   }
   else // υπάρχει το αρχείο, κλείσε το και μην το πειράξεις
   {
      test.close();
      ok = false;
   }
// ok = (δεν υπήρχε το αρχείο) && (άνοιξε για γράψιμο)
} // openWrNoReplace
```

Ας πούμε ότι έχουμε δηλώσει:

```
ofstream report;
bool     ok;
```

και στη συνέχεια το ανοίγουμε ως εξής:

```
openWrNoReplace( "report.txt", report, bool& ok );
```

Τι θα γίνει; Η συνάρτηση θα προσπαθήσει να δημιουργήσει ένα τοπικό ρεύμα (*test*) *για διάβασμα* από το αρχείο `report.txt`.

- Αν το *test* δημιουργηθεί τότε το κλείνουμε και δεν το πειράζουμε.
- Αποτυχία της απόπειρας (`test.fail()`) σημαίνει ότι το αρχείο δεν υπάρχει και έτσι προσπαθούμε να το ανοίξουμε για γράψιμο.

Με αυτόν τον τρόπο θα ανοίξουμε τα δύο εξερχόμενα ρεύματα.

Τώρα μπορούμε να προχωρήσουμε στο επόμενο επίπεδο ανάλυσης. Η "`Άνοιξε τα ρεύματα των αρχείων`" αναλύεται ως εξής:

```
Άνοιξε το autoflow για διάβασμα
if (δεν άνοιξε το autoflow)
{
   Μην προχωρείς
}
else    // το autoflow ανοικτό
{
   Άνοιξε το differences για γράψιμο
   if (δεν άνοιξε το differences)
   {
      κλείσε το autoflow
      Μην προχωρείς
   }
   else    // τα autoflow, differences ανοικτά
   {
      Άνοιξε το report για γράψιμο
      if (δεν άνοιξε το report)
      {
         κλείσε τα autoflow, differences
         Μην προχωρείς
      }
      else  //τα autoflow, differences, report ανοικτά
      {
         Προχώρησε στην επεξεργασία
      } // if (δεν άνοιξε το report)...
   } // if (δεν άνοιξε το differences)...
} // if (δεν άνοιξε το autoflow)...
```

Όλα αυτά θα τα βάλουμε σε μια συνάρτηση, ας την πούμε *openFiles*. Να δούμε τι είδους θα είναι αυτή η συνάρτηση και τι παραμέτρους θα έχει.

Η *openFiles* θα ανοίγει τρία ρεύματα από και προς αρχεία ή –με άλλα λόγια– θα δίνει τιμές σε τρία ρεύματα. Θα έχει λοιπόν τρεις *εξερχόμενες* παραμέτρους· θα είναι λοιπόν συνάρτηση *χωρίς τύπο* (`void`). Ναι, αλλά εκείνα τα «`Μην προχωρείς`» και «`Προχώρησε στην επεξεργασία`» πώς θα τα βγάζει προς τα έξω; Χρειαζόμαστε άλλη μια εξερχόμενη παράμετρο ας την πούμε *ok* που θα περνάει προς τα έξω αυτά τα μηνύματα. Αν πάρουμε υπόψη μας ότι τα δύο μηνύματα είναι αμοιβαίως αποκλειόμενα μπορούμε να δηλώσουμε την *ok* τύπου `bool`:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm,
                bool& ok )
```

Παίρνουμε ολόκληρη την *openFiles* μεταφράζοντας σε C++ το σχέδιο που δώσαμε πιο πάνω:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
```

```
                ofstream& report, string reprtFlNm,
                bool& ok )
{
   ok = true;
   autoflow.open( "autoflow.txt" );
   if ( autoflow.fail() )
      ok = false;
   else  // το autoflow ανοικτό
   {
      openWrNoReplace( "differences.txt", differences, ok );
      if ( !ok )
         autoflow.close();
      else    // τα autoflow, differences ανοικτά
      {
         openWrNoReplace( "report.txt", report, ok );
         if ( !ok )
         {
            autoflow.close();
            differences.close();
         } // if (δεν άνοιξε το report)...
      } // if (δεν άνοιξε το differences)...
   } // if (δεν άνοιξε το autoflow)...
} // openFiles
```

Εδώ, ας κάνουμε και μια σύγκριση με τις προδιαγραφές μας: Η τιμή της *ok* είναι η τιμή της «*Το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο.*»

Τώρα, ας αρχίσουμε να γράφουμε τη **main**, μεταφράζοντας σε C++ το σχέδιό μας:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      Επεξεργάσου
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.2 «Επεξεργασία»

Ας έρθουμε τώρα στην «`Επεξεργασία`» που έχει προδιαγραφές:

Προϋπόθεση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο.

Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής *n* και το πλήθος των οχημάτων *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMn_k$).

$n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Η συνθήκη «*το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο*» υπάρχει και στην προϋπόθεση και στην απαίτηση, παραμένει δηλαδή *αναλλοίωτη* από την "`Επεξεργασία`". Δηλαδή, δεν θα βάλουμε πουθενά εντολή "*close*". Αλλά, δεν ξέρεις τι άλλο μπορεί να συμβεί...

Η "$n > 0$" υπάρχει και στην προϋπόθεση και στην απαίτηση· και αυτή αναλλοίωτη; Για να κάνει το πρόγραμμα όλα όσα ζητούνται θα πρέπει να ισχύει αυτή η συνθήκη[7]. Αλλά αυτό δεν το ξέρουμε με βεβαιότητα παρά μόνον αφού προχωρήσουμε στο διάβασμα του αρχείου `autoflow.txt`. Να λοιπόν μια πιθανότητα να υπάρχει κάποιο πρόβλημα:

- Να βρούμε ένα αρχείο με λιγότερες από τέσσερις γραμμές· στην περίπτωση αυτή το *n* είναι 0. Μπορεί οι προδιαγραφές των στοιχείων εισόδου να λένε ότι έχει τουλάχιστον μια τιμή, αλλά ένα καλό πρόγραμμα θα ελέγξει αν πληρούνται οι προδιαγραφές των στοιχείων εισόδου! Θα πρέπει λοιπόν να είμαστε προσεκτικοί όταν έλθει η ώρα να υπολογίσουμε τη μέση τιμή της ροής.

Υπάρχει πιθανότητα να βρούμε άλλο λάθος κατά την επεξεργασία; Φυσικά υπάρχει:

- Αφού το `autoflow.txt` είναι text υπάρχει μεγαλύτερη πιθανότητα να υπάρχουν λάθη μέσα στο αρχείο. Πιθανές προσπάθειες παρεμβάσεων με κάποιον κειμενογράφο μπορεί να έχουν αφήσει διάφορα σφάλματα. Κάποιος έλεγχος εγκυρότητας των στοιχείων είναι αναγκαίος· θέλοντας όμως να κρατήσουμε το παράδειγμά μας απλό δεν θα τον κάνουμε προς το παρόν.

Οι τέσσερις πρώτες γραμμές, ο «τίτλος», θα πρέπει να διαβαστούν –και να αντιγραφούν– ως κείμενο. Από εκεί και πέρα τα ξέρουμε (σχεδόν): Μηδενίζουμε την *n* και τη *numberOfCars* κλπ. Για να είναι τα πράγματα εντάξει θα πρέπει να φτάσουμε να διαβάσουμε μέχρι και μια, τουλάχιστον, τιμή ροής.

```
Αντίγραψε τίτλο
if (εντάξει)
{
   Μηδένισε τους μετρητές
   Διάβασε και επεξεργάσου τις τιμές
   Τελικοί υπολογισμοί
}
```

Όπως κάναμε και πιο πριν, θα πρέπει να διατυπώσουμε προδιαγραφές για τη νέα διάσπαση. Αυτό σου το αφήνουμε ως άσκηση. Στη συνέχεια θα κάνουμε το ίδιο αλλά με αρκετά «ελεύθερο» τρόπο.

Η αντιγραφή του «τίτλου» μπορεί να γίνει όπως ξέρουμε (αντιγραφή αρχείου text), αλλά θα πρέπει να αντιγράψουμε τέσσερις γραμμές μόνον.

Θα γράψουμε μια συνάρτηση *copyTitle* που θα της περνούμε τα ρεύματα *autoflow* και *report* και αυτή θα αντιγράφει από το πρώτο στο δεύτερο τρεις γραμμές. Μέσω μιας παραμέτρου (`bool ok`) θα γνωστοποιεί αν τα κατάφερε.

Τι είδους θα είναι η συνάρτηση; Αφού η συνάρτηση θα εκτελεί αντιγραφή από το ένα αρχείο στο άλλο (είσοδο και έξοδο στοιχείων) η συνάρτηση θα είναι χωρίς τύπο:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
```

Πρόσεξε ότι τα ρεύματα περνούν (πάντοτε) ως παράμετροι αναφοράς.

---

[7] Και για να υπάρχει μια τουλάχιστον διαφορά θα πρέπει να έχουμε "$n \geq 2$".

Θα μπορούσαμε, όπως είπαμε, να γράψουμε τη συνάρτηση όπως μάθαμε στην §8.11, αντιγράφοντας χαρακτήρα προς χαρακτήρα αλλά καλύτερα να χρησιμοποιήσουμε τη *getline()* που μάθαμε στην §10.4. Αφού δηλώσουμε:

```
   int  lineCount;   // μετρητής γραμμών
```

αντιγράφουμε και μετρούμε ως εξής:

```
   lineCount = 0;
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
```

Στο τέλος, αν αντιγράψαμε τέσσερις γραμμές όλα πήγαν εντάξει:

```
   ok = ( lineCount == 3 );
```

Να ολόκληρη η συνάρτηση:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
{
   int lineCount( 0 );   // μετρητής γραμμών
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
   ok = (lineCount == 3);
} // copyTitle
```

Ας έρθουμε τώρα στη "**Μηδένισε τους μετρητές**". Τι σημαίνει; Δύο εντολές:

```
   n = 0;  numberOfCars = 0;
```

Θα μπορούσαμε να τις βάλουμε έτσι και ούτε γάτα ούτε ζημιά.

Εμείς όμως θα τις βάλουμε σε μια συνάρτηση με όνομα *initialize*. Γιατί; Σε κάθε πρόγραμμα, συνήθως, υπάρχει μια συνάρτηση (ή και περισσότερες) που διαμορφώνει μια αρχική κατάσταση (κάνει κάποια αναλλοίωτη να ισχύει): δίνει αρχικές τιμές σε μεταβλητές, ανοίγει αρχεία κλπ. Θα τη δεις συνήθως με το όνομα *initialize* ή κάτι παρόμοιο[8]. Καλό είναι να αρχίσεις να συνηθίζεις κάτι τέτοιες πάγιες πρακτικές.

Τι τύπο θα δηλώσουμε για τις *n* και *numberOfCars*; Θα πεις «θέλει και ρώτημα; `int` βέβαια!» Για την *n* δεν υπάρχει πρόβλημα. Για την *numberOfCars* όμως, αν έχουμε `INT_MAX == 32767` και μετρήσεις από έναν πολυσύχναστο δρόμο για μεγάλο χρονικό διάστημα, ο `int` δεν είναι αρκετός και καλύτερα να βάλουμε `long`.

```
void initialize( int& n, long& numberOfCars )
{
   n = 0;
   numberOfCars = 0;
} // initialize
```

Και τώρα ερχόμαστε στο «ψητό»: "**Διάβασε και επεξεργάσου τις τιμές**". «Σιγά το ψητό» θα πεις «αθροίσματα και μέσες τιμές υπολογίζουμε συνέχεια.» Σωστό! Αρκεί να προσέξουμε την πρώτη τιμή.

Ας δούμε τώρα πώς θα γίνεται η επεξεργασία. Αφού θέλουμε να υπολογίσουμε μέση τιμή αντιγράφουμε ένα γνωστό «πατρόν» επεξεργασίας από το *Μέση Τιμή 5* (§8.5):

```
   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
```

[8] Στην πραγματικότητα η *openFiles* είναι μια τέτοια συνάρτηση για ολόκληρο το πρόγραμμα. Η *initialize()* είναι η αντίστοιχη για την επεξεργασία.

```
a.open( "exp4.txt" );
a >> x;
while ( !a.eof() )
{
   n = n + 1;               // Αυτά γίνονται για
   sum = sum + x;           // όλους τους αριθμούς
   if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
   {
      selSum = selSum + x;       // Αυτά γίνονται μόνο για
      selN = selN + 1;           // τους επιλεγόμενους αριθμούς
   } // if
   a >> x;
} // while
```

Εδώ:

- το ρεύμα που διαβάζουμε (*autoflow*) είναι ήδη ανοικτό,
- αντί για την *sum* έχουμε την *numberOfCars*,
- αντί για την *x* έχουμε την *numberPerMin*,
- μετά την ανάγνωση μιας τιμής *numberPerMin*, θα πρέπει να πηγαίνουμε στην αρχή της επόμενης γραμμής,
- δεν έχουμε επιλεκτική επεξεργασία.

Άρα παίρνουμε:

```
initialize( n, numberOfCars );
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

έχοντας δηλώσει:

```
string aLine;
char*  p;
```

Πρόσεξε πώς γίνεται η ανάγνωση μιας τιμής από το *autoflow*: αφού σε κάθε γραμμή υπάρχει μια τιμή στην αρχή της, διαβάζουμε ολόκληρη τη γραμμή:

```
getline( autoflow, aLine, '\n' );
```

και μετά, με τη *strtol*, μετατρέπουμε σε **long int** τα ψηφία που υπάρχουν στην αρχή της. Διαστήματα που μπορεί να υπάρχουν πριν από τα ψηφία αγνοούνται από τη *strtol*. Αυτή θα σταματήσει τη μετατροπή όταν βρει τον πρώτο χαρακτήρα που δεν είναι ψηφίο. Την *p* δεν τη χρησιμοποιούμε; Όχι, αλλά θα σημειώσουμε ότι εκεί βρίσκεται ένα από τα μεγαλύτερο πλεονεκτήματα αυτού του τρόπου ανάγνωσης· μπορούμε να χρησιμοποιήσουμε την πληροφορία που μας δίνει για να κάνουμε έλεγχο εγκυρότητας των στοιχείων.

Χρειαζόμαστε όμως ακόμη κάτι: τον υπολογισμό των διαφορών. Ας πούμε ότι κρατούμε την προηγούμενη τιμή της *numberPerMin* στην *previous*. Τότε θα έχουμε:

```
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   numberOfCars = numberOfCars + numberPerMin;
   difference = numberPerMin - previous;
   differences << difference << endl;
   previous = numberPerMin;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

Γιατί βγάλαμε την κλήση προς την *initialize*; Διότι έχουμε και την χωριστή επεξεργασία της πρώτης τιμής που πρέπει να μεσολαβήσει.

Την πρώτη τιμή:

- θα τη μετρήσουμε,
- θα την προσθέσουμε,
- *δεν θα υπολογίσουμε διαφορά*, αφού δεν έχουμε προηγούμενη,
- *ούτε θα γράψουμε κάτι στο αρχείο διαφορών*, αλλά
- θα την βάλουμε στην *previous*, για να χρησιμοποιηθεί από τη δεύτερη τιμή.

Να λοιπόν τι πρέπει να γίνει πριν από τη **while**:

```
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );    // 1η τιμή
   if ( !autoflow.eof() )
   {
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      previous = numberPerMin;
      getline( autoflow, aLine, '\n' );
      numberPerMin = strtol( aLine.c_str(), &p, 10 ); // 2η τιμή
      while ( !autoflow.eof() )
      {
         . . .
      } // while
   } // if (!autoflow.eof())
```

Και τώρα μπορούμε να γράψουμε την *processData* αφού αποφασίσουμε για το είδος και τις παραμέτρους της συνάρτησης. Η συνάρτηση διαβάζει από αρχείο και γράφει σε αρχείο. Θα είναι λοιπόν χωρίς τύπο. Οι παράμετροί του θα είναι τα ρεύματα των δύο αρχείων και οι *n* και *numberOfCars* (εξερχόμενες):

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
```

Ολόκληρη η *processData*:

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
{
   int numberPerMin;          // μια τιμή ροής από το autoflow.txt
   string aLine;
   char*  p;
// επεξεργασία 1ης τιμής
   getline( autoflow, aLine, '\n' );
   if ( !autoflow.eof() )
   {
      numberPerMin = strtol( aLine.c_str(), &p, 10 );   // 1η τιμή
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      int previous( numberPerMin );     // η προηγούμενη τιμή ροής
      getline( autoflow, aLine, '\n' );
      while ( !autoflow.eof() )
      {
         numberPerMin = strtol( aLine.c_str(), &p, 10 );// 2η τιμή
         ++n;
         numberOfCars = numberOfCars + numberPerMin;
         int difference( numberPerMin - previous );
                                 // η διαφορά των δύο προγούμενων
         differences << difference << endl;
         previous = numberPerMin;
         getline( autoflow, aLine, '\n' );
      } // while
   } // if (!autoflow.eof())
} // processData
```

Εδώ πρόσεξε: Οι προδιαγραφές μας λένε ότι τα αρχεία θα είναι τελικώς ανοικτά. Τι γίνεται όμως με το ρεύμα *autoflow*; Σταματάμε να ασχολούμαστε με αυτό όταν βρούμε `autoflow.eof()`. Όπως ξέρουμε όμως, στην περίπτωση αυτή, το ρεύμα δεν κλείνει βέβαια αλλά έχουμε αναστολή της λειτουργίας του.

Μετά την επεξεργασία των τιμών έχουμε τους τελικούς υπολογισμούς: από τις *n* και *numberOfCars* υπολογίζουμε

- τις ώρες και τα λεπτά καθώς και
- τη μέση τιμή ροής αν $n > 0$. Εδώ ακριβώς γίνεται ο έλεγχος προϋπόθεσης.

Όλα αυτά γράφονται στο `report.txt`:

```
void finish( ofstream& report, int n, long numberOfCars )
{
   int    hours( n / 60 ),    // διάρκεια μέτρησης σε h . . .
          minutes( n % 60 );  // . . . και min
// υπολογισμοί και γράψιμο στο report.txt
   report << endl;
   report << " Η μέτρηση κράτησε ";
   report.width(2);  report << hours << " h και ";
   report.width(2);  report << minutes << " min." << endl;
   report << " Διάβασα " << n << " τιμές." << endl;
   report << " Μέτρησα " << numberOfCars << " οχήματα συνολικά"
          << endl;
   if ( n > 0 )
   {
      double averageFlow( static_cast<double>(numberOfCars) / n );
                                        // μέση τιμή ροής οχημάτων
      report << " Μέση Τιμή Ροής: ";
      report.setf( ios::fixed, ios::floatfield );
      report.precision(1);
      report.width(5);
      report << averageFlow << " οχήματα/min" << endl;
   }
} // finish
```

Πρόσεξε ότι υπολογίζουμε και γράφουμε τη μέση τιμή ροής αφού προηγουμένως εξασφαλίσουμε ότι $n > 0$.

Τώρα, μπορούμε να γράψουμε την *processing()*. Θα είναι και αυτή `void` και θα μας δίνει μέσω μιας παραμέτρου (`bool ok`) την πληροφορία για το τι κατάφερε να κάνει:

```
void processing( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   int n;             // πλήθος τιμών ροής στο autoflow.txt,
   long numberOfCars; // το άθροισμα των τιμών ροής.
   bool ok;

   copyTitle( autoflow, report, ok );
   if ( ok )
   {
      initialize( n, numberOfCars );
      processData( autoflow, differences, n, numberOfCars );
      finish( report, n, numberOfCars );
   } // if (ok)...
} // processing
```

Προχωρώντας στο γράψιμο της **main**, έχουμε:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
```

```
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.3 «Κλείσε τα Ρεύματα»

Η *closeFiles* κλείνει τα τρία ρεύματα με τα οποία δουλέψαμε:

```
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   autoflow.close();

   differences.close();
   if ( differences.fail() )
   {
      cout << "Τις χάσαμε τις διαφορές!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = true;

   report.close();
   if ( report.fail() )
   {
      cout << "Γράψε την έκθεση με το χέρι!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = ok && true; // άχρηστη
} // closeFiles
```

Μπορεί να αποτύχει το κλείσιμο των αρχείων; Βεβαιότατα! Αν πάρουμε υπόψη μας ότι με το κλείσιμο αντιγράφεται στο αρχείο το όποιο περιεχόμενο του ενταμιευτή, σκέψου την εξής περίπτωση: ας πούμε ότι γράφεις σε αφαιρούμενο δίσκο –π.χ. σε δισκέτα– και βιάζεσαι να την αφαιρέσεις πριν εκτελεσθεί η *close*.

Παρατήρηση: ▶
Βεβαίως εδώ τελειώνουμε όλες τις γραμμές με "`endl`"· έτσι ο ενταμιευτής είναι σχεδόν μονίμως άδειος.◀

13.11.4 Ολόκληρο το Πρόγραμμα

Ολόκληρο το πρόγραμμα θα είναι κάπως έτσι:

```
#include <iostream>
#include <fstream>

using namespace std;

void openWrNoReplace( string flNm, ofstream& newStream,
                      bool& ok );
```

```
void openFiles( ifstream& autoflow,
                ofstream& differences, ofstream& report,
                bool& ok );
void process( ifstream& autoflow,
              ofstream& differences, ofstream& report,
              bool& ok );
void copyTitle( ifstream& autoflow, ofstream& report,
                bool& ok );
void initialize( int& n, long& numberOfVehicles );
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfVehicles );
void finish( ofstream& report,
             int n, long numberOfVehicles );
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok );

int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      if ( !ok )
         cout << "Δεν βρήκα ούτε μια τιμή ροής" << endl;
      closeFiles( autoflow, differences, report, ok );
      if ( ok ) // γράψιμο και κλείσιμο επιτυχώς
         cout << "Τέλος καλό, όλα καλά..." << endl;
   }
   else
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
} // main
```

13.12 Δυο Λόγια για το Παράδειγμά μας

Έχοντας δει αρκετά στοιχεία προγραμματισμού, θελήσαμε να σου δώσουμε ένα παράδειγμα ανάπτυξης ενός μη-τετριμμένου προγράμματος. Η ανάπτυξη βασίστηκε στην αρχή «διαίρει και βασίλευε» (divide and conquer). Δηλαδή, ενώ το αρχικό πρόβλημα έμοιαζε κάπως δύσκολο, το διασπάσαμε σε μικρότερα και μικρότερα, μέχρι που φτάσαμε σε προβλήματα με σχετικά απλή λύση. Όπως είχαμε πει στην §0.5, αυτή η διαδικασία λέγεται και **βήμα προς βήμα ανάλυση** (step by step refinement). Τώρα την είδαμε στην πράξη.

Ξεκινήσαμε από το αρχικό πρόβλημα και πήγαμε προς τις λεπτομέρειες, **από πάνω προς τα κάτω** (top - down). Σε αυτήν την πορεία, δημιουργούσαμε δικές μας εντολές, π.χ.: *openFiles()*, *initialize()*, *finish()* κλπ, που στη συνέχεια τις υλοποιούσαμε ως συναρτήσεις. Ένας τρόπος απεικόνισης αυτής της διαδικασίας είναι το **ιεραρχικό διάγραμμα** που βλέπεις στο Σχ. 13-3. Σε αυτό βλέπεις τα διαδοχικά επίπεδα ανάλυσης, ενώ οι συναρτήσεις που καλούνται από κάθε συνάρτηση φαίνονται σαν κλαδιά της. Το διάγραμμα αυτό είναι ένας τρόπος καταγραφής των αλλεξαρτήσεων των συναρτήσεων (για τις οποίες μιλούσαμε στην §13.6.2.)

Στο διάγραμμα αυτό δεν παρατίθενται συνήθως συναρτήσεις όπως η *sqrt()*, η *strcpy()* και άλλες τέτοιες από τις βιβλιοθήκες της C++ ή άλλες βιβλιοθήκες συναρτήσεων. Έτσι, πρέπει να βλέπεις και την *openWrNoReplace()*. Αυτή θα τη βάλουμε αργότερα σε μια δική μας βιβλιοθήκη.

Ένα άλλο σημείο που πρέπει να προσέξεις, είναι το πώς αποφασίσαμε το ποιες συναρτήσεις θα γράψουμε. Ούτε για μια στιγμή δεν είχαμε κάποια αμφιβολία. Οι συναρτήσεις «ξεπήδησαν», μπορούμε να πούμε, αυθόρμητα. Αυτό *δεν σημαίνει ότι η ανάλυση είναι μοναδική*.

Ενδιαφέρον έχει και το λογικό δέσιμο των συναρτήσεων. Κάθε μια προετοιμάζει την κατάσταση για την επόμενη (ή τις επόμενες) –όπως π.χ. η *openFiles()* για όλες τις επόμενες, η *initialize()* για την *processData()*– και κάθε μια συνεχίζει τη δουλειά από εκεί που την άφησε η προηγούμενη –όπως π.χ. οι *copyTitle()*, *processData()*.

Αυτή είναι η μέθοδος του **δομημένου προγραμματισμού** (structured programming) και είναι η μόνη μέθοδος που μπορείς να χρησιμοποιήσεις προς το παρόν. Αργότερα θα δεις και άλλες μεθόδους σχεδίασης λογισμικού και (γενικότερα) συστημάτων. Αλλά, παντού θα βλέπεις αυτές εδώ τις αρχές. Θα δεις ακόμη ότι έχει γίνει πολλή δουλειά στο δέσιμο αυτής της μεθόδου με τη μαθηματική λογική.

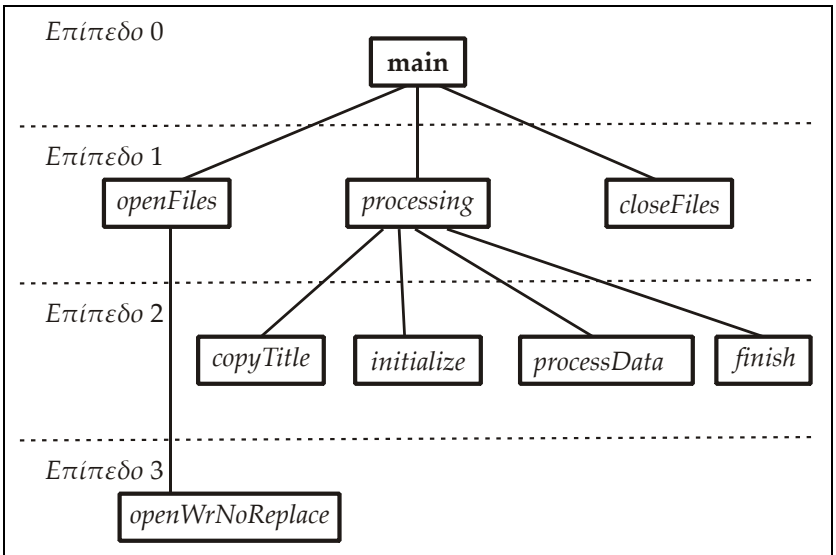
Πρόσεξε ακόμη τη σειρά υλοποίησης των συναρτήσεων. Αρχίσαμε από το τελευταίο επίπεδο, υλοποιώντας συναρτήσεις που δεν καλούν άλλες και προχωρήσαμε προς τα πάνω. Τελευταία υλοποιήθηκε η **main**. Αυτή η διαδικασία λέγεται **υλοποίηση από κάτω προς τα πάνω** (bottom-up implementation) και συνήθως πάει μαζί με τη **σχεδίαση από πάνω προς τα κάτω** (top-down design, bottom-up implementation).

Παρατηρήσεις: ▶

Κατά τ' άλλα, το πρόγραμμα που γράψαμε θέλει κι άλλη δουλειά, κυρίως στον τομέα της ασφάλειας.

1. Ένα πράγμα που δεν μας απασχόλησε καθόλου είναι ο **έλεγχος εγκυρότητας** των στοιχείων (data validation) που διαβάσαμε. Για παράδειγμα, δεν είναι δυνατόν να έχουμε *αρνητικές* τιμές ροής αλλά και μια τιμή 5000 *οχήματα/min* είναι απαράδεκτη. Ας πούμε ότι ο έλεγχος εγκυρότητας των στοιχείων έχει γίνει πιο πριν, από άλλο πρόγραμμα.

2. Ας έλθουμε στο άνοιγμα των αρχείων: Το πρόγραμμά μας είναι εξαιρετικώς ανελαστικό· ένα καλύτερο πρόγραμμα, κάθε φορά που θα εύρισκε πρόβλημα με κάποιο αρχείο, θα έπρεπε να ζητάει νέο όνομα αρχείου. ◀



Σχ. 13-3 Ιεραρχικό διάγραμμα της εφαρμογής. Η *openWrNoReplace* (επομένως και το Επίπεδο 3) θα ήταν καλό να μην υπάρχει στο διάγραμμα.

Ασκήσεις

Α Ομάδα

Για κάθε συνάρτηση που ζητείται στις Ασκ. 13-1..13-11 θα πρέπει να δικαιολογήσεις το είδος της συνάρτησης (με τύπο ή `void`*) και το είδος και τον τύπο κάθε παραμέτρου. Γράψε πρόγραμμα που θα δοκιμάζει τη συνάρτηση.*

13-1 Γράψε τη συνάρτηση *output2Dar()*, όπως την προδιαγράψαμε στο Παράδ. 6 της §13.9.3.

13-2 (συνέχεια της Άσκ. 12-5) Γράψε συνάρτηση που θα τροφοδοτείται με έναν δισδιάστατο πίνακα και δύο φυσικούς *c*1, *c*2 και θα αντιμεταθέτει τις τιμές των στοιχείων των στηλών *c*1, *c*2.

13-3 (συνέχεια της Άσκ. 12-12) Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και έναν φυσικό *k* και θα αντιμεταθέτει τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης του πίνακα.

13-4 Γράψε συνάρτηση με όνομα *h3*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με τρεις πραγματικές τιμές, ας τις πούμε χ_1, χ_2, χ_3, και έναν ακέραιο *n* και θα επιστρέφει τη διαφορά $\chi_1 - \chi_2$ αν *n* == 0, τη $\chi_2 - \chi_3$ αν *n* == 1 και τη $\chi_3 - \chi_1$ αν *n* == 2.

13-5 Μια συνάρτηση ορίζεται στο {1,2,3}×ℝ×ℝ ως εξής:

$$g(n, x, y) = \begin{cases} x + y, & n = 1 \\ x\eta\mu y, & n = 2 \\ x - 1/y, (y \neq 0) & n = 3 \end{cases}$$

Πώς θα υλοποιήσουμε τη *g()* στη C++;

13-6 Θέλουμε μια συνάρτηση, με όνομα *q1*, που θα τροφοδοτείται μέσω των παραμέτρων της, με τρεις πραγματικούς αριθμούς, ας τους πούμε *x*, *y*, *z*, και θα υπολογίζει και θα μας επιστρέφει δύο τιμές, των παραστάσεων p_1 και p_2, όπου:

αν $z > 0$ τότε $p_1 = \dfrac{x^z + y^z}{z}$ και $p_2 = z^{y+x}$,

αν $z < 0$ τότε $p_1 = \dfrac{x^{-z} - y^{-z}}{z}$ και $p_2 = (-z)^{y-x}$.

Δεν επιτρέπεται να κληθεί η *q1()* με τιμή της παραμέτρου *z* ίση με 0 (μηδέν).

13-7 Γράψε συνάρτηση που θα τροφοδοτείται, μέσω των παραμέτρων της, με δύο μονοδιάστατους πίνακες x, y –με το ίδιο πλήθος *n* στοιχείων τύπου `double`–και με έναν πραγματικό *w*. Η συνάρτηση θα κάνει το εξής: θα αλλάζει τις τιμές των στοιχείων των x και y ως εξής: αν αρχικώς x_k == *a* και y_k == *b* τότε τελικά θα πρέπει να έχουμε: x_k == *a* – *wb* και y_k == *a* + *wb*, για όλα τα *k* από 0 μέχρι *n* – 1.

Β Ομάδα

13-8 Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει τον **ανάστροφό** του (τον πίνακα που έχει ως γραμμές τις στήλες του αρχικού). Γράψε συνάρτηση που θα κάνει το ίδιο για τυχαίο δισδιάστατο πίνακα (όχι κατ' ανάγκη τετραγωνικό).

13-9 Κάθε τετραγωνικός πίνακας M (*n*×*n*) μπορεί να γραφεί ως άθροισμα δύο τετραγωνικών πινάκων *n*×*n*· ενός συμμετρικού S και ενός αντισυμμετρικού A. Τα στοιχεία τους:

$$s_{rc} = \frac{m_{rc} + m_{cr}}{2} (= s_{cr}) \text{ και } a_{rc} = \frac{m_{rc} - m_{cr}}{2} (= -a_{cr})$$

Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει το συμμετρικό και το αντισυμμετρικό μέρος του.

13-10 Με βάση το προγράμματα που έγραψες για τις Άσκ. 5-7, 5-8 γράψε μια συνάρτηση, με όνομα *trinomial()*, που:

- Θα τροφοδοτείται με τους (πραγματικούς) συντελεστές *a*, *b*, *c* ενός τριωνύμου $ax^2 + bx + c$.
- Θα υπολογίζει και θα επιστρέφει τέσσερις πραγματικές τιμές *reX1*, *imX1*, *reX2*, *imX2*, τα πραγματικά και φανταστικά μέρη των ριζών της εξίσωσης $ax^2 + bx + c = 0$.
- Θα επιστρέφει και μια ακέραιη τιμή, *n*, που θα είναι:
  - −1, αν η εξίσωση είναι αδύνατη,
  - 0 αν δεν έχει πραγματικές ρίζες,
  - 1, αν είναι πρώτου βαθμού με μία ρίζα (στη *reX1*),
  - 2, αν έχει δύο πραγματικές ρίζες (οι *imX1*, *imX2* θα είναι 0),
  - *INT_MAX*, αν είναι αόριστη.

Γράψε πρόγραμμα που θα διαβάζει τρεις πραγματικούς *a*, *b*, *c* από το πληκτρολόγιο και, καλώντας την *trinomial()*, θα λύνει την $ax^2 + bx + c = 0$.

13-11 Γράψε μια συνάρτηση, με όνομα *addPrev*, που θα τροφοδοτείται με ένα μονοδιάστατο πίνακα a με στοιχεία τύπου `double` και θα αντικαθιστά την τιμή του κάθε στοιχείου (εκτός από το a[0]) με το άθροισμά της παλιάς και της τιμής του προηγούμενου στοιχείου, δηλαδή: $a[k]_{\text{νέα}} = a[k]_{\text{παλιά}} + a[k-1]_{\text{παλιά}}$, k: 1..n−1. (Προσοχή! Θα πας από την αρχή προς το τέλος ή από το τέλος προς την αρχή;).

13-12 Κάνε αυτό που σου αφήσαμε ως άσκηση στην §13.11.2: διατύπωσε προδιαγραφές για τα επί μέρους βήματα της διάσπασης του βήματος "`Επεξεργασία`" του αρχικού προγράμματος. Μετά έλεγξε τις συναρτήσεις που γράψαμε (δηλαδή: κάνε μια άτυπη επαλήθευση). Συμμορφώνονται με τις προδιαγραφές;

13-13 α) Γράψε συνάρτηση, ας την πούμε *toSec*, που θα παίρνει *h* (0..23), *min*, *sec* μιας χρονικής στιγμής και θα τα μετατρέπει σε *sec* από τα τελευταία μεσάνυκτα.

β) Γράψε συνάρτηση, ας την πούμε *toHms*, αντίστροφη της προηγούμενης: θα παίρνει δευτερόλεπτα από τα τελευταία μεσάνυκτα και θα μας δίνει *h*, *min*, *sec*.

γ) Γράψε συνάρτηση *readHms()* που θα διαβάζει από το πληκτρολόγιο, θα ελέγχει και θα περνάει στο πρόγραμμά μας μια χρονική στιγμή που δίνεται στη μορφή *h*, *min*, *sec*.

Γράψε πρόγραμμα που θα χρησιμοποιεί τις παραπάνω συναρτήσεις για να

- διαβάσει από το πληκτρολόγιο δύο τριάδες ακεραίων h_1, m_1, s_1 και h_2, m_2, s_2 που είναι ώρα, πρώτα λεπτά και δευτερόλεπτα δύο χρονικών στιγμών,
- να ελέγξει αν είναι σωστές: $0 \le h_1, h_2 < 24$, $0 \le m_1, m_2 < 60$, $0 \le s_1, s_2 < 60$,
- να υπολογίσει και θα μας δώσει τη χρονική διαφορά μεταξύ των δύο χρονικών στιγμών σε ώρες, λεπτά και δευτερόλεπτα στη μορφή: *hh*:*mm*:*ss*. Π.χ.: Αν δοθούν: ***15 59 0*** και ***16 1 12*** γράφει: `00:02:12`.

Γ Ομάδα

13-14 Στο παιχνίδι "Φιδάκι" (snakes and ladders) ο κάθε παίκτης έχει να διανύσει έναν δρόμο με 100 βήματα αριθμημένα από 1 μέχρι 100. Ο κάθε παίκτης προχωράει όσα βήματα δείξει το ζάρι, που ρίχνει κάθε φορά που έρχεται η σειρά του. Για να τερματίσει ένας παίκτης, θα πρέπει να φέρει ζαριά που να τον φέρει ακριβώς στο 100. Αν φέρει μεγαλύτερη επιστρέφει πίσω όσα βήματα περισσεύουν (π.χ. αν είναι στο 96 και φέρει 6 θα προχωρήσει μέχρι το 100 με τα τέσσερα και θα επιστρέψει στο 98 με τα δύο που περίσσεψαν). Μια παρτίδα του παιχνιδιού τελειώνει όταν τερματίσει ένας παίκτης.

- Σε όρισμένα βήματα υπάρχουν φίδια· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα το φίδι τον «καταπίνει» και τον βγάζει στο βήμα που είναι η ουρά του (μερικά βήματα πιο πίσω).
- Σε άλλα βήματα υπάρχουν σκάλες· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα «ανεβαίνει» τη σκάλα που τον βγάζει στο βήμα που είναι η άλλη άκρη της (μερικά βήματα πιο μπροστά).

Στους παρακάτω πίνακες βλέπεις πού αρχίζουν και πού καταλήγουν οι σκάλες και τα φίδια:

Σκάλες		**Φίδια**	
αρχή	τέλος	αρχή	τέλος
5	12	10	2
17	30	25	15
40	66	37	20
60	78	50	25
75	96	88	64

Γράψε πρόγραμμα που θα προσομοιώνει μια παρτίδα του παιχνιδιού με τέσσερις παίκτες.

13-15 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο έναν φυσικό αριθμό n και θα μας δίνει όλους τους πρώτους φυσικούς που είναι ≤ n. Ο αλγόριθμος που θα χρησιμοποιήσεις είναι το *Κόσκινο του Ερατοσθένη*, που δουλεύει ως εξής:

1. Γράψε όλους τους φυσικούς από 1 μέχρι n
2. Βάλε $p = 2$
3. while ($p^2 \le n$) {

3.1 Διάγραψε όλα τα πολλαπλάσια του p

3.2 Βάλε p = επόμενος αριθμός που δεν έχεις διαγράψει

}

Παράδειγμα: (n = 22)

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
p = 2
2 3   5   7   9    11    13    15    17    19    21
p = 3
2 3   5   7        11    13          17    19
```

κεφάλαιο

14

Συναρτήσεις III

Ο στόχος μας σε αυτό το κεφάλαιο:

Να καλύψουμε μερικά υπόλοιπα σχετικά με συναρτήσεις. Μερικά από αυτά είναι πολύ σημαντικά:

- Οι συναρτήσεις *ανάκλησης* (callback).
- Η *επιφόρτωση* συναρτήσεων και τελεστών.
- Τα *περιγράμματα* (templates) συναρτήσεων.
- Οι *εξαιρέσεις* (exceptions) και η διαχείρισή τους.

Προσδοκώμενα αποτελέσματα:

Η ενσωμάτωση της χρήσης των παραπάνω τεχνικών στην ανάπτυξη λογισμικού που κάνεις θα πάρει καιρό· η πλήρης αξιοποίησή τους έρχεται με τη σχετική ωριμότητα. Είναι όμως ένα σημαντικό βήμα προς την επαγγελματική διαδικασία ανάπτυξης λογισμικού. Ακόμη, σε προετοιμάζουν για το επόμενο σχετικό βήμα που θα δούμε στο τελευταίο κεφάλαιο του *Μέρους Β*.

Έννοιες κλειδιά:

- *συναρτήσεις* `inline`
- *προκαθορισμένες τιμές παραμέτρων*
- *παράμετρος-συνάρτηση*
- *βέλος προς συνάρτηση*
- *συνάρτηση ανάκλησης* (*callback*)
- *επιφόοτωση συναρτήσεων*
- *επιφόοτωση τελεστών*
- *γενικές συναρτήσεις*
- *περιγράμματα συναρτήσεων*
- *παράμετροι της* ***main***
- *στοίβα* (*stack*)
- *εξαιρέσεις*
- *αναδρομή*

Περιεχόμενα:

14.1 Συναρτήσεις "`inline`" 390
14.2 Προκαθορισμένες Τιμές Παραμέτρων 390
14.3 Παράμετρος – Συνάρτηση 392
14.4 * Συναρτήσεις και Βέλη – Συναρτήσεις Ανάκλησης 394
14.5 Επιφόρτωση Συναρτήσεων 397
14.6 Επιφόρτωση Τελεστών 400
14.6.1 Τελεστής για Έξοδο Στοιχείων Τύπου *WeekDay* 400

14.6.2 Ο Τελεστής "++" για τον Τύπο *WeekDay* 401
14.6.3 Η Πράξη της Εκχώρησης (ξανά) 403
14.6.4 Γενικώς 404
14.7 Γενικές Συναρτήσεις 405
14.7.1 Περιγράμματα Συναρτήσεων 405
14.7.1.1 Η «Μικροδιαφορά» στο "using" 409
14.7.2 Επιφόρτωση στο Περίγραμμα 409
14.7.2.1 Επιφόρτωση, Εξειδίκευση και Άλλα Ψιλά Γράμματα... 410
14.8 Η Στοίβα 411
14.8.1 Η Συνάρτηση *stackavail* 413
14.9 Διαχείριση Εξαιρέσεων με Δυο Λόγια 414
14.9.1 Μια Ιστορία με Εξαιρέσεις 419
14.10 Αναδρομή (ξανά) 420
14.11 * Ακαθόριστο Πλήθος Παραμέτρων 424
14.12 Συνοψίζοντας 426
Ασκήσεις 427
Α Ομάδα 427
Β Ομάδα 427
Γ Ομάδα 428

14.1 Συναρτήσεις "inline"

Η κλήση μιας συνάρτησης χρειάζεται κάποιο χρόνο εκτέλεσης (πέρα από το χρόνο εκτέλεσης των εντολών που έχει στο σώμα της). Μια παλιά (και παλιομοδίτικη) προγραμματιστική συνταγή λέει: αν μια συνάρτηση καλείται πολύ συχνά μη τη γράφεις ως χωριστή συνάρτηση αλλά όπου υπάρχει η κλήση της αντίγραψε όλες τις εντολές της. Κάτι τέτοιο βέβαια «φουσκώνει» πολύ το πρόγραμμα και δεν είναι εύκολα διαχειρίσιμο. Το πρόβλημα λύθηκε, από πολύ παλιά, με τις **μακροσυναρτήσεις** (macros): πρόκειται για συναρτήσεις που γράφονται μεν χωριστά αλλά ο μεταγλωττιστής αντικαθιστά τις κλήσεις τους με τις (μεταγλωττισμένες) εντολές τους.

Αν και σήμερα το πρόβλημα με τις κλήσεις συναρτήσεων δεν είναι και τόσο σοβαρό, η C++[1] μας δίνει τη δυνατότητα να ζητήσουμε από τον μεταγλωττιστή να χειριστεί κάποια συνάρτηση ως μακροσυνάρτηση. Αν γράψεις, π.χ.:

```
inline int max( int x, int y )
{
   int fvx;

   if ( x > y )  fvx = x;
          else  fvx = y;
   return fvx;
} // max int
```

ζητάς από τον μεταγλωττιστή να χειριστεί τη συνάρτηση ως μακροσυνάρτηση. Δεν είναι καθόλου σίγουρο ότι ο μεταγλωττιστής θα σε υπακούσει: υπακούει όταν η συνάρτηση είναι αρκετά απλή. Στην περίπτωσή μας η **if** μπορεί να τον αποτρέψει. Αν όμως γράψεις:

```
inline int max( int x, int y )  {  return (x > y) ? x : y;  }
```

είναι σίγουρο ότι θα σε υπακούσει.

14.2 Προκαθορισμένες Τιμές Παραμέτρων

Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση, ας την πούμε *inc*, με τις εξής προδιαγραφές:

```
void inc( int& x, int s )
```

[1] Η προδιαγραφή "`inline`" έχει περιληφθεί και στην τυποποίηση της C του 1999 (ISO/ IEC 1999).

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }
{ x == x_0 } `inc( x );` { x == $x_0 + 1$ }
Δηλαδή, η *inc* θα μπορεί να καλείται άλλοτε με ένα όρισμα και άλλοτε με δύο![2]

Ας πάρουμε μόνον την:

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }

Αυτήν μπορούμε να τη γράψουμε; Εύκολα:

```
void inc( int& x, int s )
{
   x += s;
} // inc
```

Με μια μικρή μετατροπή μπορούμε να έχουμε αυτό που μας ενδιαφέρει:

```
void inc( int& x, int s = 1 )
{
   x += s;
} // inc
```

Τι λέει η νέα επικεφαλίδα; Αν δεν δοθεί δεύτερο όρισμα –αντίστοιχο της *s*– η *s* να θεωρηθεί ότι είναι 1.

Έτσι, αν στο πρόγραμμά μας δώσουμε:

```
   int p;

   p = 5;   inc( p, 2 );   cout << p << endl;
   p = 5;   p += 2;   cout << p << endl;
   p = 5;   inc( p );   cout << p << endl;
   p = 5;   ++p;   cout << p << endl;
```

θα πάρουμε:

```
7
7
6
6
```

Αν λοιπόν γράψουμε καταλλήλως μια συνάρτηση μπορούμε να την καλούμε παραλείποντας μερικά ή όλα τα ορίσματα.

Είναι φανερό ότι,

- ***Στην κλήση μιας συνάρτησης, δεν μπορείς να παραλείψεις ορίσματα που αντιστοιχούν σε παραμέτρους αναφοράς.***

Ακόμη:

- ***Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.***

Φυσικά, η συνάρτηση θα πρέπει να είναι γραμμένη καταλλήλως. Ας δούμε τι σημαίνει αυτό με ένα παράδειγμα. *Δεν επιτρέπεται να γράψεις*:

```
int f( double x, int y = 0, float z )
```

ενώ *μπορείς να γράψεις*:

```
int f( double x, int y = 0, float z = 1.5 )
```

Η κλήση της *f* μπορεί να είναι:

```
f( 1.56 )    f( a+b, a, b/2 )   f( p+0.5, n+1 )
```

- Στην πρώτη περίπτωση το 1.56 θα γίνει τιμή της *x*. Οι *y* και *z* θα έχουν τις ερήμην καθορισμένες τιμές τους 0 και 1.5 αντιστοίχως.
- Στη δεύτερη περίπτωση η τιμή της *a+b* θα γίνει τιμή της *x*, η τιμή της *a* θα γίνει τιμή της *y* και η τιμή της *b*/2 θα γίνει τιμή της *z*.

[2] Δηλαδή θα δουλεύει άλλοτε ως "`x += a`" και άλλοτε ως "`++x`" ή "`x += 1`".

- Στην τρίτη περίπτωση οι τιμές των p+0.5 και n+1 θα γίνουν τιμές των x και y αντίστοιχα. Η z θα ξεκινήσει με την ερήμην καθορισμένη τιμή της 1.5.

14.3 Παράμετρος – Συνάρτηση

Θέλουμε να γράψουμε μια συνάρτηση που θα υπολογίζει, προσεγγιστικά, μια λύση της εξίσωσης $f(x) = 0$ στο διάστημα $[a_0,b_0]$. Μια πολύ απλή και σίγουρη μέθοδος είναι αυτή της **διχοτόμησης**[3] (bisection).

Η μέθοδος αυτή προϋποθέτει ότι η f είναι συνεχής στο διάστημα $[a_0,b_0]$ και ότι η $f(a_0)f(b_0) \le 0$. Αν m_0 είναι το μέσο του διαστήματος $[a_0,b_0]$, ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
αλλιώς
   ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

Τα παραπάνω γράφονται εύκολα σε C++ ως εξής:

```
a = a0;  b = b0;
m = (a + b) / 2;
if ( f(a)*f(m) <= 0.0 )  b = m;
                    else  a = m;
```

Το «*ψάχνουμε για τη λύση στο διάστημα* $[a_0,m_0]$» το μεταφράσαμε «`b = m`», που βέβαια είναι σωστό. Παρακάτω θα δεις ότι μπορεί να μεταφραστεί και αλλιώς (Άσκ. 14-3).

Στο νέο διάστημα ψάχνουμε με τον ίδιο τρόπο. Πρόσεξε ότι οι τιμές των a, b μεταβάλλονται έτσι που το εύρος του διαστήματος να ελαττώνεται. Πότε σταματούμε; Όταν η τιμή της m προσεγγίσει ικανοποιητικά την τιμή της λύσης. Αλλά πόσο είναι το μέγιστο σφάλμα που μπορεί να έχουμε εδώ; Η απόσταση κάθε σημείου του διαστήματος $[a,b]$ από το μέσο του m είναι $|b - a|/2$ το πολύ. Αν ρ είναι η ακριβής τιμή της λύσης, τότε θα έχουμε και $|m - \rho| \le |b - a|/2$. Αν λοιπόν θέλουμε να υπολογίσουμε τη λύση με ακρίβεια ε, θα μικρύνουμε το διάστημα τόσο[4] ώστε $|b - a|/2 < \varepsilon$.

Ο αλγόριθμός μας θα είναι:

```
a = a0;  b = b0;
while ( fabs(b - a)/2 >= epsilon )
{
   m = (a + b) / 2;
   if (f(a)*f(m) <= 0.0)  b = m;
                     else  a = m;
} // while
root = m;
```

Όλα αυτά θα γίνονται με την προϋπόθεση ότι $f(a_0)f(b_0) \le 0$. Αν δεν ισχύει αυτή η συνθήκη για το αρχικό διάστημα, καλύτερα να μην αποπειραθούμε να προχωρήσουμε, αλλά να στείλουμε στο πρόγραμμα που καλεί τη διαδικασία ένα σήμα λάθους. Αυτό συνήθως γίνεται με μια παράμετρο, ας την πούμε *errCode*, μέσω της οποίας επιστρέφεται τιμή 0 αν όλα πήγαν καλά και 1 αν το αρχικό διάστημα ήταν λάθος.

Τι είδους συνάρτηση θα γράψουμε: με τύπο ή χωρίς τύπο; Η συνάρτησή μας θα τροφοδοτείται με την f και τα a_0, b_0 και ε και θα επιστρέφει την προσέγγιση της ρίζας και την *errCode*. Αφού θα επιστρέφει δύο τιμές θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο:

```
void bisection( ...
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

---

[3] Περισσότερα για τη μέθοδο αυτή σε οποιοδήποτε βιβλίο Αριθμητικής Ανάλυσης αλλά και στο βιβλίο Ανάλυσης της Γ′ Λυκείου. (Κατσαργύρης et al. 1992).
Δες και στο `http://en.wikipedia.org/wiki/Bisection_method`.

[4] Η αλήθεια είναι ότι συνήθως έχουμε πετύχει την ακρίβεια που θέλουμε πιο πριν.

Οι *a*0, *b*0, *epsilon* είναι παράμετροι τιμής, μια και θα έχουν πληροφορίες που δίνονται *προς* τη διαδικασία. Η *root* θα μεταφέρει τη λύση της εξίσωσης *από* τη διαδικασία προς το πρόγραμμα που την κάλεσε. Το ίδιο και η *errCode*. Και οι τελείες στην αρχή; Μα εκεί θα πρέπει να μπει μια παράμετρος που θα μας δίνει την *f* ! Τι παράμετρος θα είναι αυτή; Η C++ λέει: εκεί θα πρέπει να βάλουμε μια παράμετρο *βέλος* προς μια συνάρτηση που επιστρέφει τιμή τύπου **double**. Δηλαδή κάτι σαν:

```
double* f(double)
```

Το "**(double)**" μετά το *f* δείχνει ότι η *f* είναι συνάρτηση με μια παράμετρο, τύπου **double**. Τώρα θα γυρίσεις στον πίνακα προτεραιοτήτων (Παράρ. Ε) και θα δεις ότι πρώτα αναγνωρίζεται το "**f(double)**" και μετά το "*". Αλλά αφού το *f* είναι βέλος, το "**f(double)**" δεν έχει νόημα. Το "***f**" είναι συνάρτηση και νόημα έχει το "**(*f)(double)**". Άρα, το σωστό είναι:

```
double (*f)(double)
```

και η επικεφαλίδα είναι:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
```

Αλλά και *μέσα στο σώμα της συνάρτησης* –κατ' αρχήν– *δεν έχει νόημα να γράφουμε*: "**f(a)**" *αλλά*: "**(*f)(a)**". Να λοιπόν η συνάρτηση:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
{
   double m;

   if ( (*f)(a)*(*f)(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b-a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( (*f)(a)*(*f)(m) <= 0.0 )  b = m;
                                   else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Η C++ σου κάνει ένα δώρο· *μπορείς να γράφεις απλώς*:

```
   if ( f(a)*f)(b) > 0 )
```

και

```
         if ( f(a)*f(m) <= 0.0 )  b = m;
```

αντιστοίχως.

Τώρα να δούμε πώς θα καλέσουμε τη *bisection()*. Ας πούμε ότι θέλουμε λύση της εξίσωσης $x - ln(x) - 2 = 0$ στο διάστημα [0.1, 1]. Κατ' αρχάς, γράφουμε μια συνάρτηση που υπολογίζει την τιμή της $x - ln(x) - 2$:

```
double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και –π.χ. μέσα στη **main**– βάζουμε την εντολή:

```
   bisection( &q, 0.1, 1.0, 1e-5, riza, errCode );
```

Τι σημαίνει **&q**; Ας πούμε ότι είναι είναι ένα βέλος προς τη θέση της μνήμης όπου είναι γραμμένες οι εντολές της *q()*.

Υπάρχει όμως και εδώ το δώρο της C++· σου επιτρέπει να παραλείψεις το **&** και να γράψεις απλούστερα:

```
   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
```

Αν θέλεις να βρείς τη λύση της εξίσωσης συνx = 0 στο διάστημα [0,π] δώσε την εντολή:

```
   bisection( cos, 0.0, 4*atan(1), 1e-5, riza, errCode );
```

Φυσικά, στην περίπτωση αυτή, δεν θα πρέπει να ξεχάσεις να βάλεις "**#include cmath**" όπου υπάρχει το υπόδειγμα των *cos()* και *atan()*.

Να ένα παράδειγμα χρήσης της *bisection()*:

```
#include <iostream>
#include <cmath>

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   const double pi( 4*atan(1.0) );
   double riza;
   int   errCode;

   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
   bisection( cos, 0.0, pi, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
double q( double x )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Στη συνέχεια βλέπεις το αποτέλεσμα του παραπάνω προγράμματος.

```
 Ρίζα = 0.1585983
 Ρίζα = 1.570784
```

Παρατηρήσεις:▶ σχετικώς με τη μέθοδο της διχοτόμησης

1. Πρόσεξε ότι μετά από n επαναλήψεις το σφάλμα είναι το πολύ $|b - a|/2^n$. Επομένως, θα μπορούσαμε ισοδυνάμως αντί για το ε να δίνουμε στη συνάρτηση κάποιο *nMax*.
2. Αν θελήσεις να χρησιμοποιήσεις τη *bisection()* πρόσεξε το εξής: το κριτήριο τερματισμού $|b - a|/2 < \varepsilon$ συχνά δεν είναι αρκετό. Ένα άλλο κριτήριο που μπορεί να χρησιμοποιείται (μαζί με αυτό που έχουμε ή –ισοδυνάμως– μαζί με μέγιστο αριθμό επαναλήψεων) είναι το $|f(x)| < \varepsilon'$.◀

14.4 * Συναρτήσεις και Βέλη – Συναρτήσεις Ανάκλησης

Το είδαμε λοιπόν και αυτό: βέλος προς συνάρτηση! Ας κάνουμε όμως μια αφαίρεση: να δούμε τι πρόβλημα είχαμε να λύσουμε και πώς το λύσαμε.

Πολύ συχνά έχουμε να γράψουμε πρόγραμμα που δεν έχει ορισμένες τις προδιαγραφές του μέχρι την τελευταία λεπτομέρεια. Στην περίπτωσή μας έπρεπε να γράψουμε συνάρτηση για την προσεγγιστική επίλυση της εξίσωσης $f(x) = 0$ χωρίς να ξέρουμε ποια ακριβώς είναι η f. Όταν γράφουμε τη *bisection* το μόνο που ξέρουμε για την f είναι ότι:

f : `double` → `double`

Τι κάνουμε λοιπόν; Δίνουμε λύση στο πρόβλημά μας –δηλαδή γράφουμε συνάρτηση– *με παράμετρο* την *f*. Και το εργαλείο που χρησιμοποιούμε για την υλοποίηση είναι το *βέλος προς τη συνάρτηση f*. Λέμε ότι η παράμετρος *f* περιμένει μια **συνάρτηση ανάκλησης** (callback function)[5]. Στο παράδειγμά μας, τέτοιες είναι η *q* και η *cos*.

Ας δούμε άλλο ένα παράδειγμα από την API των Windows[6]. Οι προγραμματιστές που την έγραψαν είχαν το εξής πρόβλημα: Όταν κάποιος προγραμματιστής (που χρησιμοποιεί την API για να γράψει μια εφαρμογή Windows) θέλει να βγάζει το πρόγραμμά του ένα **πλαίσιο διαλόγου** (dialog box) θα πρέπει να το σχεδιάσει και να γράψει μια συνάρτηση που θα καθορίζει όλη την αλληλεπίδρασή του με τον χρήστη. Αλλά ο προγραμματιστής της εφαρμογής δεν θα πρέπει να ασχολείται με το πώς θα εμφανιστεί το πλαίσιο· αυτό γίνεται με πάγια διαδικασία. Έγραψαν λοιπόν μια συνάρτηση:

```
int DialogBox( HINSTANCE hInstance,
               PCTSTR pTemplate,    // όνομα του πλαισίου διαλόγου
               HWND hWndParent,
               DLGPROC pDialogFunc )    // βέλος προς τη συνάρτηση
                           // διαχείρισης του πλαισίου διαλόγου
```

Στη δεύτερη παράμετρο περιμένει το όνομα του πλαισίου διαλόγου (από αυτό θα βρει το σχέδιο). Στην τέταρτη παράμετρο περιμένει βέλος προς τη συνάρτηση διαχείρισης του πλαισίου διαλόγου που είναι μια συνάρτηση ανάκλησης.

Η *DialogBox*, αφού κάνει αυτά που γίνονται για κάθε πλαίσιο διαλόγου, θα καλέσει τη συνάρτηση ανάκλησης για τη διαχείριση του συγκεκριμένου πλαισίου. Παράδειγμα επικεφαλίδας τέτοιας συνάρτησης:

```
bool CALLBACK about( HWND hDlg, UINT iMessage,
                     WPARAM wParam, LPARAM lParam )
```

ή, απλούστερα:

```
bool about( HWND hDlg, unsigned int iMessage,
            unsigned short int wParam, long int lParam )
```

Και η αντίστοιχη κλήση:

```
        DialogBox( hInstance, "AboutBox", hWnd, &about );
```

ή:

```
        DialogBox( hInstance, "AboutBox", hWnd, about );
```

Τα παραπάνω δείχνουν παραδείγματα που χρησιμοποιείς βέλη προς συναρτήσεις (και δεν μπορείς να αποφύγεις.) Μπορεί να δεις και άλλες «εξωτικές» (ή, για άλλους, «τρομακτικές») περιπτώσεις αλλά να έχεις υπόψη σου ότι αναφέρονται κατά κύριο σε προγραμματισμό στη γλώσσα C. Στη C, με παρόμοιες τεχνικές, υλοποιούνται «γενικές» συναρτήσεις. Η C++ σου δίνει εργαλεία που (θα τα δούμε στη συνέχεια) που σου επιτρέπουν να αποφεύγεις τις «εξωτικές» περιπτώσεις.

Στο (Haendel 2001) θα βρεις πολλά πράγματα για βέλη προς συναρτήσεις, για τα προβλήματα που χρησιμοποιούνται και σχετικά εργαλεία και τεχνικές της C++. Στο –πιο απλό– (Hosey 2007) θα βρεις πολλά πράγματα για βέλη στη C και στη C++.

Εδώ θα δώσουμε μερικά παραδείγματα άλλων συνδυασμών συναρτήσεων και βελών που μπορεί να συναντήσεις.

Ξεκινούμε με την πιο απλή αλλά και πολύ συνηθισμένη περίπτωση: *συνάρτηση που επιστρέφει βέλος*. Τέτοιες συναρτήσεις έχουμε δει ήδη: *strcpy*, *strcat* και άλλες παρόμοιες. Στις περιπτώσεις αυτές όμως το αποτέλεσμα «έβγαινε» από δυο μεριές: ως αποτέλεσμα της συνάρτησης και ως τιμή της πρώτης παραμέτρου. Θα γράψουμε τώρα μια συνάρτηση:

```
const char* myStrLT( const char* s1, const char* s2 )
```

---

[5] Θα διαβάσεις αλλού ότι «συναρτήσεις ανάκλησης είναι αυτές που καλούμε με βέλος».

[6] Windows Application Program Interface (διεπαφή προγράμματος εφαρμογής).

που θα επιστρέφει ως τιμή βέλος προς τον ορμαθό –*s1* ή *s2*– που προηγείται λεξικογραφικώς. Προφανώς, είναι πολύ απλή:

```
const char* myStrLT( const char* s1, const char* s2 )
{
   const char* fv( s1 );
   if ( strcmp(s2, fv) < 0 ) fv = s2;
   return fv;
} // myStrLT
```

Συναρτήσεις που θα βγάζουν βέλη προς πίνακες διαφόρων τύπων –και όχι μόνον `char`– θα γράψουμε αρκετές.

Ας έλθουμε τώρα στο δεύτερο παράδειγμα. Τι είναι εκείνο το "`DLGPROC`"; Αν ψάξεις στο winuser.h θα δεις ότι είναι όνομα τύπου. Σε απλουστευμένη μορφή:

```
typedef int (*DLGPROC)( HWND, unsigned int,
                        unsigned short int, long int );
```

Δηλαδή έχουμε έναν τύπο βέλους προς συνάρτηση! Ο τύπος του `&`*about* είναι ακριβώς *DLGPROC*.

Παρομοίως, στην προηγούμενη παράγραφο, θα μπορούσαμε να ορίσουμε:

```
typedef double Real1( double );
```

Ο *Real1* είναι ο τύπος που περιλαμβάνει τις συναρτήσεις με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας τον θα μπορούσαμε να γράψουμε:

```
void bisection( Real1* f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Θα μπορούσαμε επίσης να ορίσουμε:

```
typedef double (*PReal1)( double );
```

Ο *PReal1* είναι ο τύπος που περιλαμβάνει *βέλη* προς συνάρτηση με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας αυτόν τον τύπο η επικεφαλίδα της *bisection* γράφεται:

```
void bisection( PReal1 f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Να δούμε τώρα *έναν πίνακα συναρτήσεων*! Ας πούμε ότι θέλουμε έναν πίνακα συναρτήσεων[7] τύπου *Real*1. Εδώ υπάρχει ένα πρόβλημα: αν δοκιμάσεις να δηλώσεις:

```
   Real1 funcArr[3];
```

ο μεταγλωττιστής δεν θα το επιτρέψει. Γιατί; Όλα τα στοιχεία του πίνακα πρέπει να έχουν το ίδιο μέγεθος· πόσο θα είναι αυτό; Αν δηλώσεις:

```
   PReal1 funcArr[3];
```

όλα πάνε καλά!

Γυρίζουμε στο παράδειγμα της προηγούμενης παραγράφου και το εμπλουτίζουμε με μια ακόμη συνάρτηση (για τη λύση της εξίσωσης $e^x = x+2$):

```
double em2( double x )
{
   return exp(x)-x-2;
} // em2
```

Έστω ότι θέλουμε να προσεγγίσουμε ρίζες των εξισώσεων:

$$x = lnx + 2 \text{ για } x \in [0.1,1], \quad συνx = 0 \text{ για } x \in [0,\pi], \quad e^x = x+2 \text{ για } x \in [0,2]$$

Μπορούμε να το κάνουμε ως εξής;

```
   const double pi( 4*atan(1.0) );
```

[7] «Τι να τον κάνουμε;» θα ρωτήσεις. Αν γράφεις πρόγραμμα C++ μάλλον δεν θα χρειαστείς τέτοιο πράγμα. Αν γράφεις C...

```
   PReal1 funcArr[3];
   double a[3], b[3];
   double riza;
   int   errCode;

   funcArr[0] = &q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = &cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = &em2;  a[2] = 0.0;  b[2] = 2.0;

   for ( int k(0); k < 3; ++k )
   {
      bisection( funcArr[k], a[k], b[k], 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
   }
```

Μην αμφιβάλλεις ότι μπορούμε να δώσουμε απλούστερα:

```
   funcArr[0] = q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = em2;  a[2] = 0.0;  b[2] = 2.0;
```

Ακόμη, μπορούμε να δηλώσουμε τον πίνακα χωρίς να ορίσουμε τον *PReal*1:

```
   double (*funcArr[3])( double );
```

Τέλος, ας δούμε και μια *συνάρτηση που επιστρέφει (βέλος προς) συνάρτηση*. Γράφουμε την:

```
PReal1 funcSel( PReal1 funcArr[], int n, int sel )
{
   return funcArr[sel];
}
```

Όπως βλέπεις αυτή επιστρέφει βέλος προς συνάρτηση και μπορούμε να γράψουμε (για λόγους επίδειξης και μόνον):

```
   bisection( funcSel(funcArr, 3, 1), a[1], b[1], 1e-5,
              riza, errCode );
```

Βεβαίως, θα μπορούσαμε να παραλείψουμε τους ορσμούς των ενδιάμεσων τύπων και να γράψουμε κατ' ευθείαν:

```
double (*(funcSel(double (*funcArr[])(double),
                  int n,int sel )))(double)
{
   return funcArr[sel];
}
```

Δεν σου αρέσει, ε;! Ας ελπίσουμε ότι δεν θα χρειαστεί να γράψεις τέτοιες συναρτήσεις.

Τελειώσαμε; Προς το παρόν: Ναι! Αλλά, υπάρχουν και χειρότερα που θα τα δεις αργότερα…[8]

14.5 Επιφόρτωση Συναρτήσεων

Στην §13.9.3 είδαμε τη συνάρτηση *swap()* που είναι πολύ χρήσιμη για όλους τους τύπους και όχι μόνον για τον `int`.

Να τη γράψουμε λοιπόν και για άλλους τύπους! Και για να μην αλλάζουμε πολύ το όνομα από τον έναν τύπο στον άλλο, θα κάνουμε το εξής: θα αλλάξουμε αυτήν που γράψαμε σε *swapI*, θα γράψουμε μια *swapD*, για μεταβλητές τύπου `double`, μια *swapC*, για μεταβλητές τύπου `char`… Ναι αυτά θα κάνουμε μόνο που δεν χρειάζεται να αλλάζουμε ονόματα.

Το πρόβλημα «*swap* για διάφορους τύπους» μπορεί να λυθεί έτσι:

```
#include <iostream>
```

[8] Έλα, αστείο ήταν!

```
using namespace std;

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

void swap( int& x, int& y );
void swap( double& x, double& y );
void swap( char& x, char& y );
void swap( WeekDay& x, WeekDay& y );

int main()
{
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

void swap( int& x, int& y )
{
   int s( x );
   x = y;  y = s;
} // swap int

void swap( double& x, double& y )
{
   double s( x );
   x = y;  y = s;
} // swap double

void swap( char& x, char& y )
{
   char s( x );
   x = y;  y = s;
} // swap char

void swap( WeekDay& x, WeekDay& y )
{
   WeekDay s( x );
   x = y;  y = s;
} // swap WeekDay
```

Εδώ, γράψαμε *τέσσερις διαφορετικές συναρτήσεις* –για τέσσερις τύπους (`int`, `double`, `char`, `WeekDay`)– *με το ίδιο όνομα*. Ο μεταγλωττιστής διαλέγει τη συνάρτηση που ταιριάζει στην κάθε κλήση ανάλογα με τους τύπους των ορισμάτων που βρίσκει σε αυτήν. Αυτό που κάναμε ονομάζεται **επιφόρτωση συναρτήσεων** (function overloading).

Το παράδειγμά μας δείχνει μια καλή χρήση επιφόρτωσης συναρτήσεων· *όλες οι συναρτήσεις που γράψαμε κάνουν την ίδια δουλειά* σε μεταβλητές διαφόρων τύπων.

Μπορείς όμως να γράφεις συναρτήσεις με το ίδιο όνομα που να κάνουν τελείως διαφορετικά πράγματα. Για παράδειγμα, μαζί με τις άλλες *swap()* να επιφορτώσεις και αυτήν:

```
double swap( int a, double b ) {  return (a+b)/2;  }
```

Ο μεταγλωττιστής δεν έχει αντίρρηση, αλλά κάτι τέτοιο δεν είναι και η πιο σοφή ιδέα για ένα πρόγραμμα.

Ας δούμε τώρα μερικά προβλήματα που μπορεί να συναντήσεις με τις επιφορτώσεις συναρτήσεων. Ας πούμε ότι ορίζεις:

```
void f( int& x, int& y ) { /*...*/ }
int f( int x, int y ) { /*...*/ return ...; }
```

```
void f( int& x, int y=0 ) { /*...*/ }
```

και μέσα στο πρόγραμμά σου δίνεις:

```
   int j1, j2;
// . . .
   f( 17, j2 );
   f( j1 );
   f( j1, j2 );
```

Βάλε τώρα τον εαυτό σου στη θέση του μεταγλωττιστή: Ποιαν από τις τρεις συναρτήσεις θα διάλεγες;

- Για την πρώτη περίπτωση τη δεύτερη συνάρτηση· είναι η μόνη που μπορεί να δεχθεί το "`17`" ως πρώτο όρισμα
- Για τη δεύτερη περίπτωση την τρίτη συνάρτηση· είναι η μόνη που μπορεί να κληθεί με ένα όρισμα.
- Για την τρίτη περίπτωση; Οποιαδήποτε! Και οι τρεις ταιριάζουν!

Αυτό το πρόβλημα μπορεί να σου παρουσιάζεται συχνά. Για να το λύνεις θα διαβάζεις προσεκτικά το διαγνωστικό που έβγαλε ο μεταγλωττιστής και θα σκέφτεσαι τα εξής:

- ***Ο μεταγλωττιστής επιλέγει –μεταξύ συναρτήσεων με το όνομα που δίνεται στην κλήση– την «πιο κοντινή» με βάση την υπογραφή (signature) της συνάρτησης, που για τη C++ είναι: το όνομα της συνάρτησης και οι τύποι των παραμέτρων.***[9]

Πιο συγεκριμένα:

- Πρώτη προτεραιότητα δίνεται σε συνάρτηση που οι τύποι των παραμέτρων της είναι ακριβώς ίδιοι με αυτούς των ορισμάτων της κλήσης. Έτσι, αν έχουμε τις συναρτήσεις:

```
void f( char x )                    // (a)
void f( int x )                     // (b)
void f( double x )                  // (c)
```

και δώσουμε:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (a).

- Δεύτερη προτεραιότητα δίνεται σε συναρτήσεις που οι τύποι των παραμέτρων τους είναι δυνατόν να προκύψουν από αυτούς των ορισμάτων της κλήσης με *ακέραιη προαγωγή*[10] ή κάποια από τις `float` σε `double` και `double` σε `long double`. Έτσι αν έχεις τις

```
void f( int x )                     // (b)
void f( double x )                  // (c)
```

για την:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (b).

- Τρίτη προτεραιότητα έχουν συναρτήσεις που οι τύποι των ορισμάτων τους προκύπτουν από αυτόν του ορίσματος με πάγιες μετατροπές τύπου όπως `int` σε `double`, `double` σε `int` και άλλες που θα μάθουμε αργότερα. Έτσι, αν δεν υπάρχουν οι επιλογές (a) και (b) ο μεταγλωττιστής θα δεχθεί τη (c).[11]

---

[9] Προσοχή! Ο επιστρεφόμενος τύπος δεν είναι μέρος της υπογραφής για τη C++· έτσι, στο παράδειγμά μας (τρίτη περίπτωση) ταιριάζει και η δεύτερη συνάρτηση.

[10] Δες το Παράρ. Ε.

[11] Υπάρχουν άλλα δύο σκαλοπάτια στην κλίματα προτεραιότητας που έχουν σχέση με πράγματα που θα μάθουμε αργότερα.

14.6 Επιφόρτωση Τελεστών

Οι τελεστές είναι συναρτήσεις. Ας πάρουμε τον "`+`". Μπορείς να τον δεις ως:

+: `int`×`int` → `int`

Ναι, αλλά μπορώ να γράψω και `1.2 + 3`. Φυσικά! Έχουμε επιφόρτωση των:

+: `double`×`int` → `double`

+: `int`×`double` → `double`

και βέβαια δεν τελειώσαμε εδώ. Ως άσκηση, σκέψου μερικές ακόμη συναρτήσεις «επιφορτωμένες» στον "`+`".

Η C++ μας επιτρέπει να επιφορτώνουμε όποιον (σχεδόν) θέλουμε από τους τελεστές της, όπως επιφορτώνουμε τις (άλλες) συναρτήσεις. Έχει όμως κάποιους περιορισμούς:

- *Δεν μπορούμε να πειράξουμε τους τελεστές που είναι ορισμένοι στους πρωτογενείς τύπους*. Π.χ. δεν μπορείς να επιφορτώσεις τον τελεστή "`*`" για πράξη μεταξύ ακεραίων. Μπορούμε να επιφορτώνουμε τελεστές για δικούς μας τύπους.
- *Δεν μπορούμε να αλλάξουμε το συντακτικό χρήσης των τελεστών*: αν ένας τελεστής είναι ενικός, όπως ο "`++`", θα πρέπει να επιφορτωθεί ως ενικός, αν είναι δυαδικός, όπως ο "`/`", θα πρέπει να επιφορτωθεί ως δυαδικός.
- *Δεν επιτρέπεται να επιφορτώσουμε τους τελεστές*: "`::`", "`.`", "`.*`", "`?:`".[12]

Και γιατί να επιφορτώσουμε έναν τελεστή, θα ρωτήσεις. Για να κάνουμε τη ζωή μας (προγραμματιστικώς) πιο εύκολη. Αν, για παράδειγμα, η *wd* είναι τύπου *WeekDay*, είναι προτιμότερο να προχωρούμε στην επόμενη μέρα εβδομάδας γράφοντας `++wd` αντί για `next(wd)` ή `advance(wd)`. Βεβαίως, να τονίσουμε ότι, αν επιφορτώσουμε για τη δουλειά αυτή τον "`--`", τότε... καλύτερα να μην επιφορτώσουμε! Να θυμάσαι πάντοτε έναν βασικό κανόνα:[13]

- ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***
  Από τεχνική άποψη:
- ***Η επιφόρτωση ενός τελεστή, ας πούμε του "`@`", γίνεται με την επιφόρτωση της συνάρτησης "`operator@()`".***

Προς το παρόν, ο μόνος τύπος που έχουμε ορίσει είναι ο *WeekDay* (§4.8). Στη συνέχεια λοιπόν θα δώσουμε παραδείγματα επιφόρτωσης των τελεστών "`++`" (ενικός) και "`<<`" (δυαδικός) για τον τύπο αυτόν και θα δεις τις τεχνικές λεπτομέρειες της επιφόρτωσης.

14.6.1 Τελεστής για Έξοδο Στοιχείων Τύπου *WeekDay*

Θέλουμε να επιφορτώσουμε τον "`<<`" έτσι ώστε αν η *wd*, τύπου *WeekDay*, έχει τιμή `sunday` και δώσουμε:

```
tout << wd << endl;
```

να γραφεί στο αρχείο text που έχει συνδεθεί με το *tout*:

```
Κυριακή
```

Αν, αντί για *tout*, γράψουμε στο *cout* θα πρέπει να το δούμε στην οθόνη.

Ένας δυαδικός τελεστής, σαν τον "`<<`", υλοποιείται (επιφορτώνεται) με συνάρτηση που έχει δύο παραμέτρους και όνομα "`operator<<`":

- Η πρώτη αντιστοιχεί στο όρισμα που εμφανίζεται αριστερά του "`<<`", στο παράδειγμά μας το "`tout`".

---

[12] Αργότερα θα τους μάθουμε όλους!

[13] Αυτή είναι και η σύσταση DCL11 του (CERT 2009): "Preserve operator semantics when overloading operators."

- Η δεύτερη αντιστοιχεί στο όρισμα που εμφανίζεται δεξιά του "`<<`", στο παράδειγμά μας το "`wd`".

  Αν θυμηθούμε τώρα ότι, όπως λέγαμε στην §13.9.2:

- «*Η* `cout << " x = "` *είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.» (Για την περίπτωσή μας να σκέφτεσαι "`tout`" αντί για "`cout`")
- «*Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* ***in out*** *(αναφοράς).*»
- «*Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο cout βάζε παράμετρο τύπου* `ostream` *και όχι* `ofstream`.»

καταλήγουμε στο συμπέρασμα ότι η επιφόρτωση θα πρέπει να γίνει με συνάρτηση που έχει επικεφαλίδα:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
```

και όχι "`void operator<<(…`" όπως λένε οι κανόνες μας.

Τα υπόλοιπα είναι απλά:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   string fv;
   switch ( rhs )
   {
      case sunday:    fv = "Κυριακή";  break;
      case monday:    fv = "Δευτέρα";  break;
      case tuesday:   fv = "Τρίτη";  break;
      case wednesday: fv = "Τετάρτη";  break;
      case thursday:  fv = "Πέμπτη";  break;
      case friday:    fv = "Παρασκευή";  break;
      case saturday:  fv = "Σάββατο";  break;
   } // switch
   return ( tout << fv );
} // operator<< WeekDay
```

Βέβαια, μπορούμε να το γράψουμε και έτσι:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   switch ( rhs )
   {
      case sunday:    return ( tout << "Κυριακή" );
      case monday:    return ( tout << "Δευτέρα" );
      case tuesday:   return ( tout << "Τρίτη" );
      case wednesday: return ( tout << "Τετάρτη" );
      case thursday:  return ( tout << "Πέμπτη" );
      case friday:    return ( tout << "Παρασκευή" );
      case saturday:  return ( tout << "Σάββατο" );
   } // switch
} // operator<< WeekDay
```

Αυτή η μορφή είναι πιο γρήγορη από την πρώτη αλλά παραβιάζει (άλλον) έναν από τους κανόνες μας: έχει πολλές `return`!

Όπως θα δούμε και στη συνέχεια, έτσι θα επιφορτώνουμε τους τελεστές εξόδου (αλλά και εισόδου) για όποιους τύπους μας ενδιαφέρει.

14.6.2 Ο Τελεστής "++" για τον Τύπο *WeekDay*

Όπως, πιθανότατα, μαντεύεις, αυτός ο τελεστής θα επιφορτωθεί με συνάρτηση που έχει όνομα "`operator++()`" και μια παράμετρο. Θέλουμε να τον ορίσουμε έτσι που *να μας δίνει την επόμενη ημέρα εβδομάδας*, όπως για ακέραιους μας δίνει τον επόμενο ακέραιο. Αν δεν δώσουμε δικό μας ορισμό, οι

```
   WeekDay d;
   . . .
```

```
   ++d;
```

δεν έχουν πρόβλημα αν η *d* έχει τιμή από *sunday* μέχρι *friday*: η τιμή της *d* προχωρεί στην επόμενη μέρα (ας πούμε: `d = d + 1`). Αν όμως η *d* έχει τιμή *saturday* τότε παίρνει τιμή έξω από τα όρια του τύπου ενώ θα έπρεπε να «ξαναγυρίζει» στη *sunday*.

Να πώς διορθώνουμε αυτό το πρόβλημα:[14]

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday ) lhs += 1;
                    else lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Ας δούμε τώρα μια δοκιμή (με οποιαδήποτε από τις δύο μορφές του `operator<<`):

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

enum WeekDay { sunday, def, tuesday, wednesday, thursday, friday,
               saturday };

ostream& operator<<( ostream& tout, WeekDay rhs );
WeekDay operator++( WeekDay& lhs );

int main()
{
// . . .
   WeekDay d( sunday );
   for ( int k(0); k < 10; ++k )
   {
      cout << d << endl;
      ++d;
   }
// . . .
} // main
```

Αποτέλεσμα:

```
Κυριακή
Δευτέρα
Τρίτη
Τετάρτη
Πέμπτη
Παρασκευή
Σάββατο
Κυριακή
Δευτέρα
Τρίτη
```

[14] Η Borland C++ v.5.5 θα δεχτεί αυτήν τη μορφή. Η gcc θα σου πει ότι δεν έχει ορισθεί ο "`+=`" για τον τύπο *WeekDay*. Στην περίπτωση αυτή γράψε την επιφόρτωση ως εξής:

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday )
   {
      int iv( static_cast<int>(lhs) );
      ++iv;
      lhs = static_cast<WeekDay>(iv);
   }
   else
      lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Πού καλείται η `operator++()`; Στην εντολή "`++d`"! Δεν το πιστεύεις; Λοιπόν άλλαξε τη "`++d`" σε:

```
operator++( d );
```

Εδώ δεν έχεις πρόβλημα: Καλείται η συνάρτηση `operator++()` με όρισμα *d*. Ε, το πρόγραμμά σου θα περάσει από τον μεταγλωττιστή και εκτελούμενο θα δώσει ακριβώς τα ίδια αποτελέσματα που πήραμε παραπάνω! Μπορείς να θεωρήσεις ότι αυτή είναι η «κανονική» κλήση ενώ η "`++d`" είναι μια συντομογραφία με οικείο συντακτικό.

Και πού καλείται η `operator<<()`; Στην "`cout << d`". Για να το πιστέψεις, άλλαξε και εδώ τη "`cout << d << endl`" σε:

```
operator<<( cout, d ) << endl;
```

και θα δεις ότι –και πάλι– το πρόγραμμά θα δώσει τα ίδια αποτελέσματα! Και εδώ, μπορείς να θεωρήσεις ότι η η "`cout << d`" είναι συντομογραφία ενώ «κανονική» κλήση είναι η "`operator<<( cout, d )`".

Παρατήρηση: ▶

Η `operator++()` δεν είναι συμβατή με τους κανόνες μας: αλλάζει την τιμή της παραμέτρου και επιστρέφει τιμή. Εδώ όμως προτιμούμε να είμαστε συμβατοί με τη φιλοσοφία της C++ για τους τελεστές "`++`" και "`--`".

Στην επόμενη υποπαράγραφο θα μάθουμε ότι η «φιλοσοφία της C++» (για την ακρίβεια της C) απαιτεί τύπο αποτελέσματος της συνάρτησης `operator++()` όχι `WeekDay` αλλά `WeekDay&`.◀

Και αν θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" τι κάνουμε; Κάτι «παράξενο»:

```
WeekDay operator++( WeekDay& lhs, int )
{
   WeekDay sv( lhs );
   ++lhs;
   return sv;
} // operator++( WeekDay, int
```

Η (ανύπαρκτη) δεύτερη παράμετρος είναι *σήμα προς τον μεταγλωττιστή ότι εδώ μιλάμε για τον μεταθεματικό τελεστή* και όχι για τον προθεματικό.

Όσο για την υλοποίηση: χρησιμοποιούμε τον προθεματικό που έχουμε ήδη ορίσει.

14.6.3 Η Πράξη της Εκχώρησης (ξανά)

Έχουμε μάθει ότι με την `v =` *Π* γίνονται τα εξής:

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

Σε αυτό βασίζεται και η πολλαπλή εκχώρηση:

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

που εκτελείται ως:

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

Τι θα έλεγες τώρα αν έβλεπες:

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

Δεν σου κάνει νόημα, ε; Δες το παρακάτω πρόγραμμα:

```
#include <iostream>

using namespace std;

int main()
```

```
{
   int u(1), v(2), w(3);
   (w = v) = u;
   cout << u << "  " << v << "  " << w << endl;
}
```

Αποτέλεσμα:

```
1  2  1
```

Η πρώτη έκπληξη είναι ότι έγινε δεκτή, δηλαδή η "`(w = v)`" θεωρήθηκε ως τιμή-*l*. Η δεύτερη έκπληξη είναι ότι η *w* πήρε τιμή "`1`" (της *u*) και όχι "`2`" (της *v*).

Παρόμοια ισχύουν και για τις συντομογραφίες της εκχώρησης. Για παράδειγμα οι:

```
   cout << u << endl;
   (u += 7) = 11;
   cout << u << endl;
   (++u) = 17;
   cout << u << endl;
```

δίνουν:

```
1
11
17
```

Πώς εξηγούνται τα παραπάνω; Με την υποσημείωση της §11.3 «*Πρόσεξε: «η* `v`*» και όχι «η τιμή της* `v`*». Θα καταλάβεις αργότερα...*» Το «αργότερα» είναι τώρα.

- Ας πάρουμε την "`(w = v) = u`". Η τιμή της *v* αποθηκεύεται στην *w*, αλλά το αποτέλεσμα της πράξης "`w = v`" είναι *η μεταβλητή w*, δηλαδή μια τιμή-*l*. Έτσι, στη συνέχεια εκτελείται η πράξη "`w = u`" και η *w* παίρνει την τιμή της *u* που είναι "`1`".
- Ας πάρουμε την "`(++u) = 17`" που είναι ισοδύναμη με "`(u=u+1) = 17`". Η "`u=u+1`" θα επιστρέψει ως αποτέλεσμα τη μεταβλητή *u* και στη συνέχεια θα εκτελεσθεί η "`u = 17`".

Για να είμαστε σύμφωνοι με αυτά, όταν επιφορτώνουμε οποιονδήποτε τελεστή εκχώρησης θα πρέπει να βγάζουμε ως αποτέλεσμα της πράξης τη «μεταβλητή του αριστερού μέρους». Θα βάλουμε λοιπόν ως τύπο αποτελέσματος όχι τον τύπο *T* της μεταβλητής αλλα τον *T*`&`.

- ***Για να μην αποκλίνουμε από τη «φιλοσοφία της C++» (της C) θα πρέπει και εμείς να επιφορτώνουμε τους τελεστές εκχώρησης ως:***

  ```
  T& operator=( T& lhs, const T& rhs )
  T& operator+=( T& lhs, const T& rhs )
  T& operator++( T& lhs )
  ```

  (*όχι* `const T&`).[15] ***Τα ίδια ισχύουν και για τους "--", "-=", "*=" κλπ.***

  Επομένως, θα ξαναγράψουμε την επιφόρτωση του

```
WeekDay& operator++( WeekDay& v )
{
   if ( v < saturday ) v += 1;
                  else v = sunday;
   return v;
} // operator++( WeekDay
```

Για τους μεταθεματικούς "`++`", "`--`" ισχύουν αυτά που είπαμε στην προηγούμενη παράγραφο. Στην περίπτωση αυτή δεν μπορούμε να επιστρέψουμε τύπο αναφοράς, αφού επιστρέφεται μια μεταβλητή (*sv*) τοπική στη συνάρτηση.

14.6.4 Γενικώς ...

Γενικώς, μπορούμε να πούμε ότι

[15] Δεν θα παραλείψουμε όμως να τονίσουμε ότι η σύσταση 34 της (ELLEMTEL 1992) λέει: "An assignment operator ought to return a `const` reference to the assigning object."

- Επιφορτώνουμε έναν *δυαδικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( Tl lhs, Tr rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *Tl* ο τύπος της πρώτης παραμέτρου και *Tr* ο τύπος της δεύτερης παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, y )...`" είτε ως "`...x @ y...`". Στην πρώτη παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*) και στη δεύτερη (*rhs*) αυτό που εμφανίζεται μετά από αυτόν (*y*).
- Επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( T rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x )...`" είτε ως "`...@x...`". Στη μοναδική παράμετρο (*rhs*) θα πάει το όρισμα που εμφανίζεται μετά από τον τελεστή (*x*).
- Επιφορτώνουμε έναν *μεταθεματικό ενικό* τελεστή `@` (όταν υπάρχει και *προθεματικός ενικός* τελεστής `@`) με μια συνάρτηση
  `Trv operator@( T lhs, int )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, 1 )...`" είτε ως "`...x@...`". Στην πρώτη (και στην πραγματικότητα) μοναδική παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*).

Αργότερα θα δούμε ότι για κλάσεις μερικά από αυτά τα πράγματα θα αλλάξουν. Αλλά –να επαναλάβουμε με άλλα λόγια– σε κάθε περίπτωση, το ουσιώδες είναι το εξής:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

14.7 Γενικές Συναρτήσεις

Στην §14.5 γράψαμε και επιφορτώσαμε τέσσερις φορές τη *swap*. Αλλά, η μόνη διαφορά στις τέσσερις συναρτήσεις είναι ο τύπος των παραμέτρων. Ο μεταγλωττιστής της C++ μας δίνει την εξής δυνατότητα: να του δώσουμε το σχέδιο –το «πατρόν– και να γράψει εκείνος όποιες συναρτήσεις θα χρειαστούν. Αυτό το σχέδιο λέγεται **περίγραμμα συνάρτησης** (function template).

Για το περίγραμμα θα δεις και τον όρο **γενική συνάρτηση** (generic function). Πάντως, όπως θα καταλάβεις στη συνέχεια *μια γενική συνάρτηση δεν είναι συνάρτηση αφου δεν έχει συγκεκριμένο πεδίο ορισμού ή/και πεδίο τιμών*.

14.7.1 Περιγράμματα Συναρτήσεων

Ξαναγράφουμε το παράδειγμα της §14.5 ως εξής:

```
#include <iostream>

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

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

template < typename T >
void swap( T& x, T& y );

int main()
{
```

```
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Η βασική διαφορά[16] είναι: αντί για τις τέσσερις συναρτήσεις έχουμε το:

```
template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Αυτό είναι ένα **περίγραμμα** (template) της συνάρτησης *swap* με παράμετρο τον τύπο *T*. Αντί για "`typename T`" θα μπορούσαμε να γράψουμε και "`class T`". Τα "`class`", "`template`" και "`typename`" είναι λέξεις-κλειδιά της C++.

Το πρόγραμμα αυτό δίνει:

```
20  10
2.34  1.23
B  A
2  0
```

Όταν ο μεταγλωττιστής βρει τη "`swap(j1, j2)`" δημιουργεί ένα **στιγμιότυπο** (instance) του περιγράμματος σε μια συνάρτηση σαν την

```
void swap( int& x, int& y )
```

Το στιγμιότυπο δημιουργείται *αυτομάτως* αφού δεν υπάρχει αμφιβολία για το ότι θα πρέπει να βάλει όπου *T* τον `int`.

Παρομοίως, όταν βρεί τη "`swap(d1, d2)`" ο μεταγλωττιστής θα δημιουργήσει ένα στιγμιότυπο:

```
void swap( double& x, double& y )
```

κ.ο.κ.

Ένα περίγραμμα μπορεί να έχει ως παραμέτρους

- έναν ή περισσότερους *τύπους* ή/και
- *ακέραιους*,
- *βέλη* ή *αναφορές*.

και μπορεί να μας δώσει μια συνάρτηση –που τη λέμε **στιγμιότυπο του περιγράμματος** (template instance)– με δύο τρόπους:

- Όπως στα παραπάνω παραδείγματα, όπου οι τύποι που θα αντικαταστήσουν τους τύπους-παραμέτρους συνάγονται αυτομάτως από τους τύπους των ορισμάτων της κάθε κλήσης. Αυτή είναι η **συναγόμενη δημιουργία στιγμιοτύπου** (implicit instantiation).
- Με τη **ρητή** (explicit) δημιουργία στιγμιοτύπου κατά την οποία γράφουμε, στην κλήση, τις τιμές των παραμέτρων μετά το όνομα του περιγράμματος. Π.χ. στο παράδειγμά

[16] Υπάρχει και η «μικροδιαφορά» στο "`using`". Θα τη συζητήσουμε παρακάτω.

μας, αντί για `swap(d1, d2)`, θα μπορούσαμε να γράψουμε: `swap<double>(d1, d2)`. Ας δούμε ένα

Παράδειγμα 1

Έστω ότι θέλουμε να μετατρέψουμε σε περίγραμμα τη *vectorSum* (§12.1) Μια απλή προσπάθεια μας οδηγεί στο εξής:

```
template < typename CT >
CT vectorSum( const CT x[], int n, int from, int upto )
{
   CT sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

Εδώ θα πρέπει να σκεφτούμε την εξής περίπτωση: Έστω ότι βάζουμε ως *CT* τον `double`. Παρ' όλο όμως που τα στοιχεία του πίνακα είναι `double` μπορεί το άθροισμα τους να είναι μεγαλύτερο από το *DBL_MAX*. Σε μια τέτοια περίπτωση θα μπορούσαμε να πάρουμε το σωστό αποτέλεσμα σε `long double`. Την αλλάζουμε λοιπόν:

```
template < typename CT, typename RT >
RT vectorSum( const CT x[], int n, int from, int upto )
{
   RT sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

Τώρα όμως έχουμε άλλο πρόβλημα: Έστω ότι έχουμε δηλώσει

```
   double ar[7971];
   long double res;
```

Αν δοκιμάσουμε να δώσουμε

```
   res = vectorSum( ar, 7971, 0, 7970 );
```

ο μεταγλωττιστής δεν θα μπορέσει να βγάλει άκρη. Το σωστό είναι να γράψουμε τις οδηγίες για τη δημιουργία στιγμιοτύπου:

```
   res = vectorSum<double, long double>( ar, 7971, 0, 7970 );
```

Όταν η παράμετρος είναι *ακέραιος*

- Αυτή μπορεί να χρησιμοποιηθεί στη συνάρτηση ως σταθερά.
- Το αντίστοιχο όρισμα θα πρέπει να είναι σταθερά.

Παράδειγμα 2

Στο Παράδ. 6 της §13.9.3 συζητήσαμε για δυο συναρτήσεις

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
void output2DAr( ostream& tout,
                 const int* a, int nRow, int nCol )
```

για να μη γράφουμε ξανά και ξανά τις ίδιες εντολές στο πρόγραμμα πολλαπλασιασμού πινάκων (Παράδ. 4, §12.4). Εκεί γράψαμε την πρώτη από αυτές και εσύ, πού έλυσες την Άσκ. 13-1, έχεις γράψει τη δεύτερη.

Τώρα όμως μας κατεβαίνει μια ιδέα: Αφού μπορούμε να βάλουμε στο περίγραμμα ακέραιη παράμετρο που έρχεται στη συνάρτηση ως σταθερά, να βάλουμε εκεί τον αριθμό στηλών του πίνακα και να δηλώσουμε τον πίνακα ως δισδιάστατο:

```
template< typename T, unsigned int nCol >
void input2DAr( istream& tin, T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
```

```
} // input2DAr
```

Καλό; Εξαιρετικό! Να κάνουμε έτσι και την άλλη:

```
template < typename T, unsigned int nCol >
void output2DAr( ostream& tout,
                 const T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Τώρα, στο πρόγραμμά μας διαβάζουμε με τις:

```
   input2DAr< int, m >( atx, a, l );
   input2DAr< int, n >( atx, b, m );
```

γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr< int, m >( cout, a, l );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr< int, n >( cout, b, m );
   cout << " Στοιχεία του πίνακα c" << endl;
   output2DAr< int, n >( cout, d, l );
```

και όλα είναι μια χαρά![17] Πρόσεξε ότι εδώ *είναι απαραίτητη η ρητή δημιουργία στιγμιοτύπων* αφού ο αριθμός στηλών του πίνακα δεν υπάρχει στην κλήση ώστε να μπορεί να συναχθεί η τιμή της δεύτερης παραμέτρου.

Αλλά τίποτε δεν δίνεται δωρεάν στον κόσμο αυτόν! Αν χρησιμοποιήσουμε τις συναρτησεις με τον «γραμμοποιημένο» πίνακα το μέγεθος του εκτελέσιμου είναι 156160 ψηφιολέξεις ενώ με τα περιγράμματα έχουμε 156672.[18]

Γιατί; Διότι στην πρώτη λύση έχουμε δύο συναρτήσεις ενώ στη δεύτερη έχουμε πέντε: δύο *input2DAr* και τρεις *output2DAr*! Γράφηκαν αυτομάτως βέβαια, αλλά γράφηκαν.

Και μετά από αυτό, να απαντήσουμε και το ερώτημα που βάλαμε στην υποσημείωση: Αν θέλουμε παράμετρο για το εύρος πεδίου θα τη βάλουμε στη συνάρτηση. *Αν τη βάλεις ως παράμετρο του περιγράμματος για κάθε τιμή της θα έχουμε διαφορετικό στιγμιότυπο*· αυτό είναι υπερβολή.

Ας πούμε και κάτι ακόμη που –προς το παρόν– μπορεί να μην σου κάνει πολύ νόημα:

- Μπορείς να έχεις στιγμιότυπα της *swap* για οποιονδήποτε τύπο
  - έχει τον τελεστή εκχώρησης και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *vectorSum()* που έχουν για δεύτερο όρισμα (*RT*) οποιονδήποτε τύπο
  - έχει τον τελεστή "`+=`" ώς `operator+=( RT, CT )` και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *output2DAr* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`<<`" για έξοδο στοιχείων προς αρχείο text. Π.χ. μπορείς να έχεις στιγμιότυπο για τον *WeekDay*.

---

[17] Πάντως εκείνο το "`3`" στη "`tout.width(3)`" είναι μαγική σταθερά. Να το βάλουμε ως παράμετρο στη συνάρτηση ή μήπως στο περίγραμμα;

[18] Borland C++, v.5.5.

- Μπορείς να έχεις στιγμιότυπα της *input2Dar()* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`>>`" για είσοδο στοιχείων από αρχείο text. Π.χ. *δεν* μπορείς να δημιουργήσεις στιγμιότυπο για τον *WeekDay*.

Ας πούμε τώρα ότι θέλουμε να μετατρέψουμε σε περίγραμμα και τη *max*. Αυτό είναι εύκολο:

```
template< typename T >
T max( T x, T y )  {  return (x > y) ? x : y;  }
```

Από αυτό το περίγραμμα μπορούμε να πάρουμε στιγμιότυπα για οποιονδήποτε τύπο ορίζεται ο τελεστής "`>`", όπως είναι οι `int`, `double`, `char` κ.ο.κ. Και αν θέλουμε να συγκρίνουμε δύο ορμαθούς C (πίνακες χαρακτήρων); Τώρα *T* είναι ο `char*` και προφανώς δεν μπορούμε να πάρουμε στιγμιότυπο. Η C++ όμως μας δίνει τη δυνατότητα να κάνουμε μια **εξειδίκευση** (specialization) του περιγράμματος για τον `char*`:

```
template<>
char* max<char*>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

- Η δήλωση μιας εξειδίκευσης περιγράμματος αρχίζει με "`template<>`".
- Οι τιμές των παραμέτρων για την εξειδίκευση γράφονται μέσα σε "`<`", "`>`" μετά το όνομα του περιγράμματος.

Αν οι τιμές των παραμέτρων της εξειδίκευσης συνάγονται από τους τύπους των παραμετρων της συνάρτησης μπορείς να τις παραλείψεις. Για τη *max* μπορούμε να γράψουμε:

```
template<>
char* max<>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

Ακόμη, μπορείς να παραλείψεις τις παραμέτρους εξειδίκευσης αν α) υπάρχουν προκαθορισμένες τιμές για τις παραμέτρους του περίγράμματος και β) η εξειδίκευση γίνεται για τις προκαθορισμένες τιμές. Με τις προκαθορισμένες τιμές που βάλαμε στις παραμέτρους του *output2DAr()* γράφοντας:

```
template <>
void output2DAr<>( ostream& tout,
                   const int a[][nCol], int nRow )
// . . .
```

δηλώνουμε εξειδίκευση για τύπο `int` και *nCol* `==` 2.

14.7.1.1 Η «Μικροδιαφορά» στο "`using`"

Όταν αλλάξαμε το πρόγραμμα για να δοκιμάσουμε το περίγραμμα της *swap* βγάλαμε το γενικό

```
using namespace std;
```

και βάλαμε συγκεκριμένα:

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

Γιατί; Διότι στο *sdt* υπάρχει ήδη –μεταξύ άλλων– περίγραμμα *swap()* που μας έρχεται έτοιμο από τη βιβλιοθήκη της C++! Στα «άλλα» περιλαμβάνονται περιγράμματα για τις *min()*, *max()* και άλλες συνηθισμένες συναρτήσεις. Δοκίμασέ τα! Αν δεν φτάνει το "`using namespace std`" βάλε και "`#include <algorithm>`".

14.7.2 Επιφόρτωση στο Περίγραμμα

Ας γυρίσουμε στο περίγραμμα της *swap()* για να σκεφτούμε την εξής περίπτωση: μπορούμε να το χρησιμοποιήσουμε για να αντιμεταθέσουμε τις τιμές των:

```
   char s1[15], s2[20];
```

Όχι, φυσικά! Θα πρέπει να γράψουμε μια:

```
   void swap( char x[], char y[] )
```

και να την επιφορτώσουμε στο περίγραμμα που προϋπάρχει. Ας πούμε ότι τη γράφουμε:

```
void swap( char x[], char y[] )
{
   string s( x );
   strcpy( x, y );
   strcpy( y, s.c_str() );
} // swap( char*
```

Παρατηρήσεις: ▶

1. Γιατί «ας πούμε...»; Διότι για να δουλέψει θα πρέπει η τιμή του x να «χωράει» στον y και το αντίστροφο. Αυτό το αφήνουμε στον προγραμματιστή-χρήστη της συνάρτησης.
2. Για να τη δοκιμάσεις θα πρέπει –εκτός από τις "`using std::cout`" και "`using std::endl`"– να βάλεις και μια "`using std::string`".◀

Όταν ο μεταγλωττιστής βρει την εντολή:

```
   swap( s1, s2 );
```

θα κάνει τη σωστή επιλογή.

Αν θέλεις κάνε και το εξής πείραμα: όρισε

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }
```

δήλωσε:

```
   int k( 23 );
   double e( 7.33 );
```

και ζητήσε:

```
   qsc( k, 'a' );
   qsc( e, 'c' );
```

Η πρώτη κλήση θα εξυπηρετηθεί από εξειδίκευση του περιγράμματος ενώ η δεύτερη από την απλή συνάρτηση και θα πάρεις:

```
in template
in function
```

Μπορείς να κάνεις και επιφόρτωση περιγραμμάτων: Για παράδειγμα, μπορείς πέρα από τους παραπάνω ορισμούς της *qsc* να δώσεις και

```
template< typename T >
void qsc( int x, T c )
{  cout << "in template 2" << endl;  }
```

Δεν υπάρχει οποιοδήποτε πρόβλημα εκτός από την περίπτωση που θα βάλεις στο πρόγραμμά σου:

```
   qsc( k, 'a' );
```

Στην περίπτωση αυτήν ο μεταγλωττιστής δεν μπορεί να καταλάβει ποιο περίγραμμα εννοείς.

14.7.2.1 Επιφόρτωση, Εξειδίκευση και Άλλα Ψιλά Γράμματα...

Όταν ο μεταγλωττιστής βρει κλήση σε συνάρτηση που μπορεί να προκύψει από περίγραμμα που είναι επιφορτωμένο με άλλο (-α) περίγραμμα (-τα) ή/και απλή (-ές) συνάρτηση (-σεις) επιλέγει την κατάλληλη συνάρτηση ως εξής:

- Από όλα τα περιγράμματα με το όνομα συνάρτησης της κλήσης γίνεται προσπάθεια δημιουργίας στιγμιοτύπων που να ταιριάζουν με την κλήση.

- Από όλα τα στιγμιότυπα και τις απλές συναρτήσεις που τυχόν ταιριάζουν επιλέγεται η «πιο κοντινή» προς την κλήση με βάση αυτά που είπαμε στο τέλος της §14.5.
- Αν υπάρχει «ισοπαλία» στιγμιοτύπου περιγράμματος και απλής συνάρτησης προτεραιότητα έχει η απλή συνάρτηση.
- Αν η επιλογή είναι στιγμιότυπο κάποιου περιγράμματος και υπάρχει εξειδίκευση με την ίδια υπογραφή χρησιμοποιείται η εξειδίκευση.

Ας πούμε ότι έχουμε:

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }

template<> void qsc<double>( double x, char c )
{  cout << "in template spec" << endl;  }
```

Δηλαδή: γράφουμε μια εξειδίκευση του περιγράμματος ακριβώς για την περίπτωση που έχουμε την επιφόρτωση! Τα καταφέραμε να μπερδέψουμε τον μεταγλωττιστή με την κλήση:

```
   qsc( e, 'c' );
```

Όχι! Το πρόγραμμα θα μεταγλωττισθεί και θα δώσει:

```
in function
```

Ας δούμε γιατί γίνεται αυτό: Σύμφωνα με αυτά που είπαμε παραπάνω έχουμε να επιλέξουμε μεταξύ του στιγμιοτύπου:

```
void qsc<double>( double x, char c );
```

και της απλής συνάρτησης

```
void qsc( double x, char c );
```

Αφού στην κλήση "`qsc(e,'c')`" το πρώτο όρισμα είναι `double` και το δεύτερο `char` και τα δύο έχουν πρώτη προτεραιότητα. Αφού έχουμε «ισοπαλία» επιλέγεται η απλή συνάρτηση.

Όπως βλέπεις, στο παράδειγμά μας, ο μεταγλωττιστής δεν πρόκειται να φτάσει να εξετάσει την «υποψηφιότητα» της εξειδίκευσης του περιγράμματος για την κλήση "`qsc(e,'c')`".

Αν είχαμε επιλογή του στιγμιοτύπου του περιγράμματος τότε θα είχαμε χρήση της εξειδίκευσης.

14.8 Η Στοίβα

Στην επόμενη παράγραφο θα προσπαθήσουμε να δούμε πώς δουλεύει *ο μηχανισμός των εξαιρέσεων* της C++. Στη συνέχεια θα (ξανα)μιλήσουμε για *αναδρομή*. Και στις δύο περιπτώσεις θα αναφερθούμε στη στοίβα. Καλό είναι λοιπόν να πούμε από πριν δυο λόγια για αυτήν την πολύτιμη περιοχή μνήμης.

Μέχρι στιγμής έχουμε γνωρίσει δύο είδη[19] –από την άποψη της διαχείρισης– μνήμης:

- τη **στατική** (static), για τα στατικά αντικείμενα· οτιδήποτε βάλουμε εκεί δημιουργείται μια φορά και παραμένει στην ίδια διεύθυνση μέχρι το τέλος της εκτέλεσης του προγράμματος,
- την **αυτόματη** (automatic) ή μνήμη **στοίβας** (stack) για τα ορίσματα και τα τοπικά αντικείμενα συναρτήσεων και ομάδων· ό,τι βάλουμε εκεί ζει όσο ζει η αντίστοιχη

[19] Αργότερα θα μάθουμε και ένα τρίτο είδος: τη **δυναμική** μνήμη ή μνήμη **σωρού**.

συνάρτηση: δημιουργείται με την κλήση της και καταστρέφεται με το τέλος της εκτέλεσής της.

Το παρακάτω πρόγραμμα σου επιτρέπει να διακρίνεις αυτές τις δύο περιοχές:

```
#include <iostream>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long fl( double p )
{
   long s;

   cout << " &p = " << &p << "    &s = " << &s << endl;
   p = fd(p, 1.56);
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   y = fl(5);
   x[2] = fd( x[0], y );
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B178
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 &p = 0012FF44    &s = 0012FF38
 &p1 = 0012FF28    &p2 = 0012FF30
 &p1 = 0012FF3C    &p2 = 0012FF44
```

Εδώ, αφού πάρεις υπόψη σου ότι έχουμε δεκαεξαδικό σύστημα, παρατήρησε τα εξής:

- Η αποθήκευση της καθολικής μεταβλητής *g* έγινε σε άλλη περιοχή της μνήμης (0041B178) από αυτήν που αποθηκεύονται οι υπόλοιπες (0012FF...). Η *g* είναι στη στατική μνήμη ενώ οι υπόλοιπες είναι στη μνήμη στοίβας.
- Όπως αναμένεται, η διεύθυνση της παραμέτρου αναφοράς *a*3, στην *q*, είναι ίδια με αυτήν της *b* στη **main**.
- Η εκτέλεση του προγράμματος αρχίζει από τη **main**. Έτσι, οι πρώτες μεταβλητές που υλοποιούνται στη στοίβα είναι οι τοπικές της **main** στις διευθύνσεις από 0012FF54 (x) μέχρι 0012FF88 (*a*).
- Με την κλήση της *fl* υλοποιούνται και οι δικές της τοπικές μεταβλητές στη στοίβα από 0012FF38 (*s*) μέχρι 0012FF44 (*p*).
- Με την κλήση της *fd*, μέσα από την *fl*, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF28 (*p*1) μέχρι 0012FF30 (*p*2).
- Μετά την ολοκλήρωση της κλήσης της *fl*, όλη η μνήμη που χρησιμοποιήθηκε από την *fl* και την *fd* επιστρέφεται στη στοίβα. Αυτό φαίνεται από το επόμενο βήμα:

- Με την κλήση της *fd*, από τη **main**, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF3C (*p*1) μέχρι 0012FF44 (*p*2). Η μνήμη αυτή είχε χρησιμοποιηθεί προηγουμενως για την υλοποίηση των μεταβλητών της *fl*.

Αυτή η λειτουργία της αυτόματης μνήμης, ό,τι εισάγεται τελευταίο να εξάγεται πρώτο (Last In First Out), είναι χαρακτηριστικό της, συχνότατα χρησιμοποιούμενης, δομής στοιχείων που ονομάζεται **στοίβα** (stack, LIFO stack).

14.8.1 Η Συνάρτηση *stackavail*

Αν δουλεύεις με τον μεταγλωττιστή Borland C++, v.5.5 (ή v.5.02) μπορείς να δεις πιο καλά τη χρήση της στοίβας χρησιμοποιώντας τη συνάρτηση (εκτός προτύπου) *stackavail* που δίνει τη διαθέσιμη μνήμη στοίβας:

```
#include <iostream>
#include <malloc.h>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " μέσα στην fd, stackavail: " << stackavail() << endl;
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long int fl( double p )
{
   long s;

   cout << " μέσα στην fl, stackavail: " << stackavail() << endl;
   cout << " &p = " << &p << "    &s = " << &s << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   p = fd(p, 1.56);
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long int a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   cout << " πριν κληθεί η fl, stackavail: " << stackavail()
        << endl;
   y = fl(5);
   cout << " μετά την fl, stackavail: " << stackavail() << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   x[2] = fd( x[0], y );
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B17C
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 πριν κληθεί η fl, stackavail: 1048376
```

```
μέσα στην fl, stackavail: 1048356
&p = 0012FF40    &s = 0012FF34
πριν κληθεί η fd, stackavail: 1048356
μέσα στην fd, stackavail: 1048332
&p1 = 0012FF24    &p2 = 0012FF2C
μετά την fd, stackavail: 1048356
μετά την fl, stackavail: 1048376
πριν κληθεί η fd, stackavail: 1048376
μέσα στην fd, stackavail: 1048352
&p1 = 0012FF38    &p2 = 0012FF40
μετά την fd, stackavail: 1048376
```

Πρόσεξε τα εξής:

- Πριν κληθεί η *fl* έχουμε μνήμη στοίβας 1048376 ψηφιολέξεις. Μόλις άρχισε η εκτέλεσή της η διαθέσιμη μνήμη γίνεται 1048356 ψηφιολέξεις. Οι 20 ψηφιολέξεις διατίθενται για την υλοποίηση των μεταβλητών (8+4) και για τη διαχείριση της κλήσης. Η διαθέσιμη μνήμη στοίβας ξαναπαίρνει την αρχική της τιμή «`μετά την fl, stackavail: 1048376`».
- Η κάθε κλήση της *fd* κοστίζει 24 ψηφιολέξεις αλλά σε διαφορετική περιοχή της στοίβας κάθε φορά.

14.9 Διαχείριση Εξαιρέσεων με Δυο Λόγια

Ας ξεκινήσουμε με το παράδειγμα που είδαμε στην §7.8. Γράψαμε τη συνάρτηση *V*, που δεν ορίζονταν στα σημεία $\{ k: \mathbb{Z} \bullet 2k \}$:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      cerr << " η v κλήθηκε με όρισμα: " << x << endl;
      exit( EXIT_FAILURE );
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Εδώ εφαρμόσαμε το βήμα 7 της συνταγής του Πλ. 7.4 που λέει:
«*Αν η συνάρτηση είναι μερική, γράψε τις εντολές που εξαιρούν τις τιμές του ορίσματος για τις οποίες δεν ορίζεται η συνάρτηση:*

```
if (x δεν ανήκει στο πεδίο ορισμού)
{
   cerr << " η ... κλήθηκε με x = " << x << endl;
   exit( EXIT_FAILURE );
}
else
   Υπολόγισε την τιμή της συνάρτησης»
```

Φυσικά η συνταγή αυτή εφαρμόζεται και στις συναρτήσεις χωρίς τύπο. Αν ας πούμε, έχουμε το πρόβλημα:

> *Γράψε συνάρτηση, με το όνομα pqer, που θα τροφοδοτείται μέσω των ορισμάτων της με τρεις πραγματικές τιμές, ας πούμε x, y, z και θα υπολογίζει και θα επιστρέφει τις τιμές των παραστάσεων:*

$$t = \frac{xy}{x^2 - y^2} z^{x-y}, \quad u = \frac{xy - \dfrac{1}{x}}{z}, \quad \text{αν } |x| \neq |y| \text{ και } z > 0.$$

θα γράψουμε:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
   {
      cerr << " η pqer κλήθηκε με ορίσματα " << x << ", "
           << y << ", " << z << endl;
      exit( EXIT_FAILURE );
   }
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Αυτός ο τρόπος διαχείρισης τέτοιων καταστάσεων –ορίσματα εκτός πεδίου ορισμού– έχει δυσάρεστο αποτέλεσμα. Π.χ. οι παρακάτω εντολές προσπαθούν να δώσουν πίνακα τιμών της *v* από -5 ως 5 ανά 0.5:

```
   for ( int k(-10); k <= 10; ++k )
      cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v κλήθηκε με όρισμα: -4
```

Θα δούμε τώρα έναν άλλον τρόπο, σαφώς πιο ευέλικτο:

```
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

double v( double x );

int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " η v δεν ορίζεται στο " << x << endl;
      }
   } // for (int k...
} // main

double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      throw x;
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
```

```
η v δεν ορίζεται στο -4
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
η v δεν ορίζεται στο -2
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
η v δεν ορίζεται στο 4
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, έχουμε πολύ πιο «ήπια» αντιμετώπιση του σφάλματος. Παίρνουμε τουλάχιστον τις τιμές στις οποίες υπολογίζεται η συνάρτησή μας.

Ποιες διαφορές έχουμε τώρα; Οι «μαγικές» λέξεις είναι: `throw`, `try` και `catch`.

- Όταν καταλαβαίνουμε ότι έχουμε τιμή εκτός πεδίου ορισμού, με τη "`throw x`" **ρίχνουμε** (ή **εγείρουμε**) **μιαν εξαίρεση** (throw (raise) an exception). Η *x* είναι το **αντικείμενο της εξαίρεσης** (exception object). Ύστερα από αυτό διακόπτεται η εκτέλεση της συνάρτησης *v()* και ο έλεγχος επιστρέφει στη συνάρτηση που την κάλεσε, στην περίπτωσή μας στη **main**.
- Στη **main**, η κλήση της *v* γίνεται μέσα σε μια εντολή `try` (δοκίμασε). Στην ομάδα της εντολής `try` βάζουμε τις εντολές –κλήσεις συναρτήσεων– από τις οποίες περιμένουμε ότι μπορεί να ριχθεί κάποια εξαίρεση.
- Η εξαίρεση **συλλαμβάνεται** από κάποια `catch` (σύλλαβε) που ακολουθεί την `try`. Μια `catch` συλλαμβάνει εξαιρέσεις του τύπου που αναφέρεται στην παράμετρό της. Μέσα στην ομάδα της `catch` κάνουμε **διαχείριση της εξαίρεσης** (exception handling): λέμε στον υπολογιστή τι θέλουμε να κάνει όταν τη συλλάβει.

Στο παράδειγμα που δώσαμε δεν τονίζουμε και τόσο την υπεροχή αυτού του τρόπου. Ας δούμε μια άλλη παραλλαγή, κάπως πιο «εποικοδομητική»:

```
int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " v(" << (x+1e-5) << ") = " << v(x+1e-5) << endl;
      }
   } // for (int k...
} // main
```

που μας δίνει:

```
v(-5) = -2.33333
v(-4.5) = -3.06667
v(-3.99999) = -100001
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
v(-1.99999) = -100001
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
v(4.00001) = -100001
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, στη διαχείριση της εξαίρεσης, *ξανακαλούμε τη συνάρτηση* με τιμή που σίγουρα δεν έχει πρόβλημα, αλλά είναι κοντά στην τιμή που έριξε την εξαίρεση. Βέβαια, σε μια πραγματική εφαρμογή, κάτι τέτοιο είναι απίθανο να έχει νόημα.

Βάζοντας διαφορετικά τις **try-catch** παίρνουμε αποτελέσματα παρόμοια με αυτά που παίρναμε με την *exit()*:

```
int main()
{
   try
   {
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
} // main
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v δεν ορίζεται στο -4
```

Πάντως το πρόγραμμα είναι ουσιωδώς διαφορετικό:

- Στην αρχική μορφή *η συνάρτηση παίρνει την απόφαση* να διακόψει την εκτέλεση του προγράμματος καλώντας την *exit()*.
- Στη νέα μορφή η συνάρτηση στέλνει μήνυμα ότι συνάντησε ανυπέρβλητο πρόβλημα. Την απόφαση για το τι θα γίνει παίρνει ο «γενικός διεύθυντής», δηλαδή η **main**.

Θα πρέπει όμως, να διαλύσουμε δυο συνηθισμένες παρεξηγήσεις:

- Θα διαχειριζόμαστε πάντοτε τις εξαιρέσεις στη **main**; Όχι. Στη συνέχεια, θα δούμε πώς αποφασίζουμε πού και πώς μπορεί να γίνει η καλύτερη διαχείριση της κάθε εξαίρεσης.
- Οι εξαιρέσεις ρίχνονται μόνο από *συναρτήσεις* που καλούμε στην ομάδα **try**; Όχι! Αργότερα θα δεις παραδείγματα που θα έχουμε εντολές **throw** *μέσα στην ομάδα* της **try**.

Ας πούμε τώρα ότι στο ίδιο πρόγραμμα χρησιμοποιούμε και την *pqer*, που την αλλάζουμε ως εξής:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
      throw " η pqer κλήθηκε με ορίσματα εκτός πεδίου ορισμού";
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Τώρα, η *pqer* ρίχνει εξαίρεση τύπου **char*** (το βέλος προς το μήνυμα).

Θα πεις: «Ναι, αλλά τώρα, όταν πιάσουμε την εξαίρεση, το πολύ-πολύ να γράψουμε το μήνυμα. Δεν μπορούμε όμως να έχουμε τις τιμές των *x*, *y*, *z* που προκάλεσαν το πρόβλημα και να τις χειριστούμε "πιο δημιουργικά".» Αργότερα θα μάθουμε πώς μπορούμε να περνούμε τέτοιες πληροφορίες.

Για να γράψουμε τη **main** έχουμε το εξής πρόβλημα: η *v* μπορεί να ρίξει αντικείμενο εξαίρεσης τύπου **double** ενώ η *pqer* μπορεί να ρίξει αντικείμενο τύπου **char***. Πώς τα συλλαμβάνουμε; Δες πώς γίνεται η **main**:

```
int main()
{
   double t, u;

   try
   {
```

```
   // . . .
      pqer( 1, 2, 3, t, u );
   // . . .
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
} // main
```

Όπως βλέπεις, μια `try` μπορεί να έχει περισσότερες από μια `catch` για να χειριστεί διάφορες εξαιρέσεις που προέρχονται από μια ή περισσότερες συναρτήσεις που καλούνται μέσα στην ομάδα της.

Αν βάλεις μια `catch( ... )` συλλαμβάνεις όλες τις εξαιρέσεις. Π.χ. στο τελευταίο παράδειγμα μπορείς να βάλεις:

```
   try
   {
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
   catch ( ... )
   {
      cout << " μη αναμενόμενη εξαίρεση" << endl;
   }
```

Τέλος, να πούμε ότι μπορείς, αν θέλεις, να δηλώνεις στην επικεφαλίδα της συνάρτησης τις **προδιαγραφές εξαιρέσεων** (exception specification) –δηλαδή τους τύπους των εξαιρέσεων που μπορεί να ρίξει– ως εξής:

```
double v( double x ) throw( double );
void pqer( double x, double y, double z,
           double& t, double& u ) throw( char* );
double f( double x ) throw( double, int );
```

Αν μια συνάρτηση ρίξει εξαίρεση εκτός προδιαγραφών προκαλεί διακοπή της εκτέλεσης του προγράμματος. Προσοχή όμως: η

```
double g( double x ) throw()
```

σημαίνει ότι *από τη g() δεν θα ριχτεί ούτε θα περάσει οποιαδήποτε εξαίρεση* και είναι διαφορετική από την

```
double g( double x )
```

που σημαίνει ότι *από την g() περιμένεις οποιαδήποτε εξαίρεση.*

14.9.1 Μια Ιστορία με Εξαιρέσεις

Τώρα, θα δούμε ένα πιο πολύπλοκο παράδειγμα για να σου δώσουμε μια καλύτερη αίσθηση της λειτουργίας του μηχανισμού των εξαιρέσεων. Ξαναγράφουμε το πρόγραμμα του παραδ. 1 της §7.7 ως εξής:

```
#include <iostream>

using namespace std;

unsigned long int comb( int m, int n );

int main()
{
   int m, n;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   try
   {
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb(m,n) << endl;
   }
   catch( int )
   {
      cout << " Λάθος δεδομένα" << endl;
   }
} // main

// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
unsigned long int natProduct( int m, int n )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// 0 < m <= n
   unsigned long int fv( m );
   for ( int k(m+1); k <= n; ++k )  fv *= k;
   return fv;
} // natProduct

// factorial -- Υπολογίζει το a!
unsigned long int factorial( int a )
{
   return (a == 0) ? 1 : natProduct(1, a);
} // factorial

// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
unsigned long int comb( int m, int n )
{
   unsigned long int fv;

   if ( n < m-n )  fv = natProduct(m-n+1, m)/factorial(n);
             else  fv = natProduct(n+1, m)/factorial(m-n);
   return fv;
} // comb
```

Κατ' αρχάς, να εξηγήσουμε τι κάνουμε: Ας πούμε ότι έχουμε να υπολογίσουμε τους $\binom{5}{2}$. Στο αρχικό πρόγραμμα ζητούσαμε τον υπολογισμό:

$$\binom{5}{2} = \frac{5!}{2!(5-2)!} = \frac{5!}{2!3!} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5}{2!\cdot 1\cdot 2\cdot 3}$$

Φυσικά, μπορούμε να απλοποιήσουμε το 1·2·3 και να αποφύγουμε άσκοπες πράξεις. Η *comb()* κάνει ακριβώς αυτό, επιλέγοντας τη μέγιστη απλοποίηση. Για τον σκοπό αυτόν

όμως χρειαζόμαστε τη `natProduct( int m, int n )` που υπολογίζει το γινόμενο φυσικών αριθμών: $m\cdot(m+1)\cdot\ldots\cdot(n-1)\cdot n$, με την προϋπόθεση $0 < m \le n$. Αν η *natProduct* κληθεί χωρίς να ισχύει η προϋπόθεση ρίχνει εξαίρεση.

Έχοντας τη *natProduct()* μπορούμε να γράψουμε, όπως βλέπεις, πολύ πιο απλά τη *factorial()*.

Ο μόνος έλεγχος εγκυρότητας των δεδομένων γίνεται στη *natProduct()*. Αυτό έχει ως αποτέλεσμα να είναι το πρόγραμμα καθαρογραμμένο!

Αν η *natProduct()* κληθεί με ορίσματα που δεν ικανοποιούν την $0 < m \le n$ ρίχνει την εξαίρεση (τύπου `int`) "`-1`". Ας πούμε λοιπόν ότι, από λάθος, ζητείται στη **main** ο υπολογισμός των συνδυασμών $\binom{8}{10}$.

- Η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση "`comb(m, n)`" της *comb()*. Επειδή η $n < m-n$ δεν ισχύει, η εκτέλεση συνεχίζεται στην περιοχή του `else`. Η *comb()* για να κάνει τον υπολογισμό θα ζητήσει τους υπολογισμούς "`natProduct(11, 8)`" και "`factorial(-2)`". Ας πούμε τώρα ότι ζητείται πρώτα ο υπολογισμός "`factorial(-2)`".
- Και πάλι η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *factorial()*. Η *factorial()*, με τη σειρά της, κάνει την κλήση `natProduct(1, -2)`.
- Και αυτήν τη φορά η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *natProduct()*. Η *natProduct()* βρίσκει ότι έχουμε $m \le 0$, δεν κάνει υπολογισμούς ρίχνει την εξαίρεση "`-1`".
- Τα σχετικά με την κλήση της *natProduct()* φεύγουν από τη στοίβα και επιστρέφουμε στη *factorial()*.
- Η *factorial()* δεν συλλαμβάνει την εξαίρεση. Τα σχετικά με την κλήση της *factorial()* φεύγουν από τη στοίβα και επιστρέφουμε στην *comb()*.
- Αλλά ούτε η *comb()* έχει κάποια `catch(int ...)` για να συλλάβει την εξαίρεση· τα σχετικά με την κλήση της *comb()* φεύγουν από τη στοίβα και επιστρέφουμε στη **main** όπου η εξαίρεση συλλαμβάνεται.

Το πέρασμα του ελέγχου από κάθε συνάρτηση προς αυτήν που την κάλεσε χωρίς να γίνεται οποιαδήποτε άλλη δουλειά λέγεται **ξετύλιγμα της στοίβας** (stack unwinding). Αν η εξαίρεση δεν συλληφθεί ούτε στη **main** θα προκαλέσει τελικώς διακοπή της εκτέλεσης του προγράμματος.

Αργότερα θα δούμε τη διαχείριση των εξαιρέσεων πιο εκτεταμένα, αλλά προς το παρόν θα τονίσουμε ότι:

1. Ο μηχανισμός των εξαιρέσεων, που υλοποιείται με τις εντολές `throw`, `try` και `catch`, επιτρέπει στον προγραμματιστή να σχεδιάσει σωστά το πρόγραμμά του ώστε να διαχειρίζεται προβληματικές καταστάσεις.
2. Στις συναρτήσεις χωρίς τύπο, μπορείς, αν δεν θέλεις να χρησιμοποιήσεις εξαιρέσεις, να βάζεις μια παράμετρο από την οποίαν θα επιστρέφεις κάποια πληροφορία σχετικώς με το σφάλμα. Ξαναδές το παράδειγμα της §13.11. Πάντως το ξετύλιγμα της στοίβας με τις εξαιρέσεις είναι πιο απλό.
3. Σε συνέχεια του παραπάνω: Δες πόσο απλά και *καθαρά* γράφηκαν οι *factorial()* και *comb()*!

14.10 Αναδρομή (ξανά)

Όπως λέγαμε και στην §7.10, στη C++ υπάρχει η δυνατότητα αναδρομικής διατύπωσης μιας συνάρτησης, όπως και στα μαθηματικά υπάρχει η δυνατότητα αναδρομικής διατύπωσης μερικών ορισμών. Είδαμε ακόμη δύο παραδείγματα: μια συνάρτηση που υπολογίζει το $n!$ και μια που υπολογίζει τον μέγιστο κοινό διαιρέτη δύο ακεραίων και είχαμε παρατηρήσει

ότι ο αναδρομικός τρόπος, συγκρινόμενος με τον ισοδύναμο επαναληπτικό τρόπο, είναι πιό απλός, πιό σύντομος και πλησιέστερος προς τον μαθηματικό ορισμό.

Συχνά όμως, όπως θα δούμε παρακάτω, ο αναδρομικός τρόπος είναι λιγότερο αποδοτικός από τον επαναληπτικό και σε χρόνο και σε μνήμη.

Τί πληρώνουμε για αυτήν την καλύτερη γραφή; Για να γίνει ο υπολογισμός με την αναδρομική μορφή θα χρειαστούμε n+1 ενεργοποιήσεις της συνάρτησης, που κάθε μια παίρνει μνήμη από τη στοίβα. Χρησιμοποιούμε δηλαδή μνήμη ανάλογη του n και φυσικά αντίστοιχο χρόνο.

Ας δούμε δύο ακόμη παραδείγματα. Το πρώτο είναι κλασικό και πολυσυζητημένο. Ο (Dijkstra 1976), στη μονογραφία του "A Discipline of Programming", άφηνε έξω από το δομημένο προγραμματισμό την αναδρομή. Οι θιασώτες της φυσικά δεν το δέχτηκαν. Οι (Manna & Waldinger 1978), που δούλευαν ερευνητικώς στην αυτόματη σύνθεση προγράμματος, θεωρούσαν την αναδρομή απαραίτητη τουλάχιστον στην αρχική σχεδίαση· έγραψαν λοιπόν μια εργασία που είναι −οξύτερη από ό,τι συνηθίζεται στις επιστημονικές εργασίες− κριτική στις θέσεις του Dijkstra. Στην εργασία τους σχολιάζουν δυο (το 6ο και το 2ο) από τα «οκτώ μικρά παραδείγματα» του Dijkstra. Στη συνέχεια θα δούμε το πρώτο από αυτά τα παραδείγματα.

Παράδειγμα

Μας δίνονται δυο ακέραιοι, $x > 1$ και $y \geq 0$ (προϋπόθεση), και θέλουμε να βρούμε κάποιο z τέτοιο ώστε να ισχύει η (απαίτηση):

R: $z == x^y$

Ο Dijkstra προτείνει να χρησιμοποιήσουμε μια βοηθητική μεταβλητή h, τέτοια ώστε να ισχύει η

P: $h \cdot z == x^y$

και να γράψουμε μια

while ($h \neq 1$) «συμπίεσε» το h κρατώντας αναλλοίωτη την P

Για να ισχύει αρχικώς η αναλλοίωτη βάζουμε: "`h = x^y; z = 1`". Είναι φανερό ότι, όταν τελειώσει η εκτέλεση της **while**, θα έχουμε $h == 1$ που μαζί με την αναλλοίωτη μας δίνουν την R.

Η τιμή x^y δεν μας είναι γνωστή, ώστε να την εκχωρήσουμε αρχικά στην h. Το πρόβλημα αυτό μπορεί να λυθεί αν γράψουμε την h ως xx^{yy} οπότε η αναλλοίωτή μας γράφεται

P: $xx^{yy} \cdot z == x^y$

ενώ το πρόγραμμά μας γίνεται:

```
xx = x;  yy = y;  z = 1;
while (yy != 0)
   «συμπίεσε» το yy κρατώντας αναλλοίωτη την P
```

Η «συμπίεση» μπορεί να γίνεται με την εντολή "`yy = yy - 1`", οπότε η P παραμένει αναλλοίωτη αν συγχρόνως αλλάζουμε και την τιμή της `z = z*xx`. Είναι φανερό ότι η εκτέλεση της **while** κάποτε τερματίζεται. Να λοιπόν η συνάρτηση υπολογισμού της δύναμης:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      --yy;
      z *= xx;
   } // while
   return z;
```

```
} // power
```

Ύστερα από αυτό, ο Dijkstra ψάχνει για πιο γρήγορο αλγόριθμο. Παρατηρεί ότι αν ο *yy* είναι άρτιος και βάλουμε **xx = xx²** και **yy = yy/2** η τιμή της *h* δεν αλλάζει:

$$h == xx^{yy} == (xx^2)^{yy/2}$$

Έτσι, καταλήγει στην[20]

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      while ( even(yy) )
      {
         xx *= xx;
         yy /= 2;
      } // while (even(yy))
      --yy;
      z *= xx;
   } // while
   return z;
} // power
```

Ο τερματισμός της εσωτερικής **while** είναι σίγουρος, διότι ο μόνος άρτιος που μπορεί να διαιρείται επ' άπειρον δια 2, χωρίς να δώσει περιττό, είναι ο 0 (μηδέν), αλλά από αυτόν μας προστατεύει η εξωτερική **while**.

Ενώ η πρώτη λύση έχει χρόνο εκτέλεσης ανάλογο του *y*, η δεύτερη έχει χρόνο εκτέλεσης ανάλογο του $log_2 y$.

Οι Manna και Waldinger αντιτείνουν ότι η αναλλοίωτη, η συνθήκη συνέχισης και –τελικώς– το πρόγραμμα δεν βγήκαν, αλλά γράφτηκαν μια και ήταν γνωστά εκ των προτέρων. Και δίνουν, ως το πιο απλό πρόγραμμα που θα μπορούσε να γραφτεί για την περίπτωση, το:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
   if ( y == 0 )  fv = 1;
            else  fv = x * power( x, y-1 );
   return fv;
} // power
```

Τί πιο απλό! Δεν γράφουμε παρά τις βασικές ιδιότητες: $x^0 == 1$ και $x^y == x \cdot x^{y-1}$.

Η βελτίωση του αλγορίθμου, η αντίστοιχη αυτής που κάνει ο Dijkstra, είναι επίσης απλούστατη:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
```

[20] Η *even* παίρνει ένα όρισμα τύπου **int** και μας δίνει **true** αν το όρισμα είναι άρτιος (even), **false** αν το όρισμα είναι περιττός. Θα μπορούσες να τη γράψεις ως εξής:

```
bool even( int x )  {  return ( x % 2 == 0 );  }
```

```
   if ( y == 0 )
      fv = 1;
   else if ( even(y) )
   {  unsigned int z( power(x,y/2) );
      fv = z*z;  }
   else
      fv = x * power( x, y-1 );
   return fv;
} // power
```

και αν είχες πρόβλημα να καταλάβεις το βελτιωμένο πρόγραμμα του Dijkstra, δεν πρέπει να έχεις δυσκολία να καταλάβεις αυτό εδώ!

Δεν μπορούμε παρά να θαυμάσουμε την ομορφιά και την απλότητα του αναδρομικού προγράμματος. Αλλά, δεν είναι δωρεάν! Τι πληρώνουμε; Για σκέψου πώς θα εκτελεσθεί αυτή η συνάρτηση; Οι διαδοχικές αναδρομικές κλήσεις εισάγουν στη στοίβα τις τιμές y, y-1,... 1, 0. Στη συνέχεια αφαιρεί όλες αυτές τις τιμές, υπολογίζοντας το γινόμενο $1 \cdot x \cdot x \cdot \ldots \cdot x$. Στη βελτιωμένη περίπτωση οι κλήσεις είναι, στην καλύτερη περίπτωση, $log_2 y$. Μικρό το κακό (φυσικά για μικρά y);

Πάντως, υπάρχουν και περιπτώσεις όπου η απλότητα του αναδρομικού προγράμματος δρα σαν «σειρήνα» που μας τραβάει στον «κακό δρόμο». Τυπικό παράδειγμα οι αριθμοί Fibonacci, που ως γνωστόν ορίζονται ως εξής:[21]

$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$ για $n \geq 2$

Μεταφράζουμε λοιπόν αμέσως:

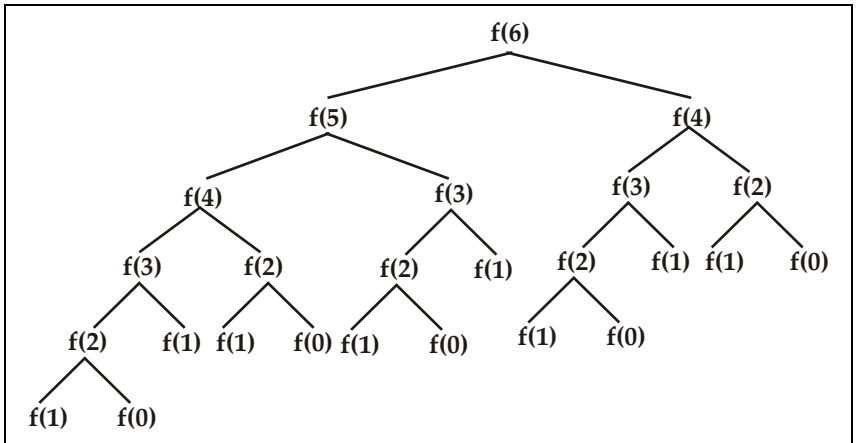
```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fv;
   if ( n < 2 )  fv = n;
           else  fv = f(n-1) + f(n-2);
   return fv;
} // f
```

και στο Σχ. 14-1 βλέπεις τι γίνεται για να υπολογιστεί ο $f(6)$. Ο $f(4)$ υπολογίζεται 2 φορές, το $f(3)$ 3, το $f(2)$ 5 και το $f(1)$ 8.

Στην περίπτωση αυτή συμφέρει να γράψουμε μια επαναληπτική συνάρτηση, φροντίζοντας να φυλάγουμε *δυο* προηγούμενους αριθμούς σε κάθε βήμα:

```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fn, fp, fpp;
   if ( n < 2 )
   {
      fn = n;
   }
   else
   {
      fp = 0;  fn = 1;
      for ( int j(1); j < n; ++j )
      {
         fpp = fp;  fp = fn;
         fn = fp + fpp;
      } // for
```

[21] Έλυσες την Άσκηση 7-14;



Σχ. 14-1 Αναδρομικές κλήσεις για τον υπολογισμό f(6).

```
   } // if (n < 2)
   return fn;
} // f
```

Για τον υπολογισμό της δύναμης είπαμε «μικρό το κακό»· εδώ δεν μπορούμε να πούμε το ίδιο. Θα γράφουμε αναδρονικές συναρτήσεις, αλλά καλό θα είναι να σκεφτόμαστε που οδηγούν (και πόσο καλές είναι άλλες εναλλακτικές λύσεις).

Ας ξαναγυρίσουμε στο παράδειγμα της *power()*. Μερικοί μπορεί να αναρωτιούνται «Είναι δυνατόν, στο επαναληπτικό πρόγραμμα να σπαζοκεφαλιάζω για αναλλοίωτες και για τερματισμούς, ενώ όταν γράφω το αναδρομικό να μην υπάρχουν τέτοια προβλήματα;» Όχι βέβαια! Δεν συμβαίνει κάτι τέτοιο. Εκείνο που συμβαίνει είναι ότι αυτά τα πράγματα είναι, συχνά, πολύ απλά για το αναδρομικό πρόγραμμα. Η αναλλοίωτη εμφανίζεται τώρα σαν συμμόρφωση προς τις προδιαγραφές της συνάρτησης· δηλαδή, η κλήση μιας συνάρτησης είτε απ' έξω είτε από μέσα από το σώμα του υποπρογράμματος, με αναδρομή, θα πρέπει να γίνεται με τους ίδιους όρους. Για να έχουμε τερματισμό, σε μια αναδρομική συνάρτηση *θα πρέπει να υπάρχει μια τουλάχιστον περίπτωση εκτέλεσης χωρίς αναδρομική κλήση*. Ακόμη, *οι αναδρομικές κλήσεις θα πρέπει να οδηγούν σε μια τέτοια εκτέλεση*.

Στην *power()* οι προδιαγραφές είναι:

το όρισμα που αντιστοιχεί στη x πρέπει να είναι θετικός ακέραιος,
το όρισμα που αντιστοιχεί στη y, πρέπει να είναι μη αρνητικός ακέραιος,
το αποτέλεσμα είναι ίσο με x^y.

Οι δυο προϋποθέσεις ελέγχονται στην αρχή της συνάρτησης. Φυσικά, δεν φτάνει αυτό: Κάθε αναδρομική κλήση μειώνει την τιμή της y κατά 1 εκτός από την περίπτωση που έχουμε y == 1. Η παραπάνω παρατήρηση μας βεβαιώνει και για τον τερματισμό. Για y == 1 δεν έχουμε αναδρομική κλήση και είναι σίγουρο ότι θα φτάσουμε σε μια κλήση της συνάρτησης με y == 1.

14.11 * Ακαθόριστο Πλήθος Παραμέτρων

Είδαμε πιο πριν ότι η κλήση μιας συνάρτησης μπορεί να έχει λιγότερες πραγματικές από τις τυπικές παραμέτρους. Η C++ σου επιτρέπει να γράφεις συναρτήσεις με μη προκαθορισμένο πλήθος παραμέτρων.

Ας πούμε ότι θέλουμε μια συνάρτηση που θα τροφοδοτείται με μια τιμή τύπου `char` και μη προκαθορισμένο πλήθος τιμών τύπου `double` και αν η τιμή τύπου `char` είναι `'>'` θα μας επιστρέφει ως τιμή τη μέγιστη από τις τιμές τύπου `double` ενώ αν είναι `'<'` θα μας επιστρέφει την ελάχιστη.

Σύμφωνα με όσα έχουμε μάθει, η συνάρτησή μας, ας την πούμε *maxormin()*, θα είναι συνάρτηση με τύπο: θα επιστρέφει τιμή τύπου **double**. Θα έχει μια παράμετρο τιμής, τύπου **char**, ας την πούμε *tel*, δηλαδή:

```
double maxormin(char tel, ...)
```

και μετά; Ε λοιπόν: η C++ δέχεται αυτήν την επικεφαλίδα·[22] τα **αποσιωπητικά** (ellipsis) δείχνουν ότι, όταν κληθεί, μπορεί να πάρει πολλά ορίσματα.

Δες πώς γράφεται η συνάρτηση και ας τη συζητήσουμε στη συνέχεια:

```
#include <iostream>
#include <cstdarg>

using namespace std;

double maxormin( char tel, ... )
{
   if ( tel != '<' && tel != '>' )
   {
      throw tel;
   }
   va_list ap;
   va_start( ap, tel );

   double x( va_arg(ap, double) );
   if ( x == 0 )
   {
      throw 0;
   }
   double fv( x );
   if ( tel == '>' )
   {
      while ( x != 0 )
      {
         if ( x > fv )  fv = x;
         x = va_arg( ap, double );
      }
   }
   else // tel == '<'
   {
      while ( x != 0 )
      {
         if ( x < fv )  fv = x;
         x = va_arg( ap, double );
      } // while (x != 0)
   } // if (tel == '>')
   va_end( ap );
   return fv;
} // maxormin

int main()
{
   double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
   double mn = maxormin( '<', 1.1, 2.2, 1.3, 0.4, 7.1, 0.0 );
   cout << mx << "  " << mn << endl;
} // main
```

1. Περιλάβαμε στο πρόγραμμά μας το cstdarg. Στο αρχείο αυτό ορίζεται ο τύπος *va_list* και δηλώνονται οι συναρτήσεις *va_arg()*, *va_start()* και *va_end()*.

2. Μέσα στη συνάρτησή μας δηλώσαμε τη μεταβλητή *ap* τύπου *va_list*. Πρόκειται για έναν πίνακα.

[22] Κληρονομιά από τη C!

3. Με την κλήση "`va_start(ap, tel)`" τροφοδοτούμε τη *va_start()* με τη μοναδική σταθερή παράμετρο (*tel*) και αυτή μας επιστρέφει πληροφορίες για τις υπόλοιπες πραγματικές παραμέτρους στην *ap*.

4. Καλούμε ξανά και ξανά τη "`va_arg(ap, double)`". Κάθε φορά μας επιστρέφει ως τιμή *την τιμή του επόμενου ορίσματος* και αλλάζει την τιμή της *ap* ώστε στη συνέχεια να προχωρήσουμε παρακάτω (στο μεθεπόμενο). Πρόσεξε ότι δεύτερο όρισμα στην κλήση της *va_arg* είναι ένας *τύπος* (!). Είναι *ο τύπος του ορίσματος που περιμένουμε να διαβάσουμε*.

5. Στο τέλος βάζουμε την κλήση της `va_end(ap)`. Είναι απαραίτητη για την ολοκλήρωση της κλήσης (αυτή θα κάνει ανάταξη της στοίβας).

Ας πούμε ότι η συνάρτησή μας καλείται να βρει τον μέγιστο. Ξεχνούμε για λίγο τις δύο **if** και παρακολουθούμε την εκτέλεση:

```
x = va_arg( ap, double );  // πάρε το πρώτο όρισμα στη x...
fv = x;                    // ... και θεώρησε ότι είναι μέγιστο
x = va_arg( ap, double );  // πάρε το επόμενο όρισμα στη x...
while ( x != 0 )           // όσο δεν είναι 0 (μηδέν)
{
   if ( x > fv )  fv = x;   //  αν x > fv διόρθωσε το μέγιστο
   x = va_arg( ap, double );//  πάρε το επόμενο όρισμα στη x...
} // while
```

Ο τρόπος που υπολογίζουμε το μέγιστο είναι ο γνωστός αλλά είναι φανερό ότι η **while** ελέγχεται με φρουρό την τιμή 0 (μηδέν) στη *x*! Ακριβώς! Όπως μπορείς να δεις στην κλήση της συνάρτησης έχουμε:

```
double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
```

Το "`0.0`" στο τέλος είναι φρουρός! Δεν θα μπορούσαμε να βάλουμε μετρούμενη επανάληψη; Βέβαια, αρκεί να βάλουμε άλλη μια σταθερή παράμετρο στην αρχή (1η ή 2η) που να περνάει στη συνάρτηση το πλήθος των ορισμάτων που ακολουθούν.

Αν θελήσεις να γράψεις τέτοια συνάρτηση (συνήθως δεν είναι η καλύτερη λύση) πρόσεξε τα εξής:

1. Ο φρουρός θα πρέπει να είναι του ίδιου τύπου με τα άλλα ορίσματα και να επιλέγεται όπως ξέρουμε (το "`0.0`" στο παράδειγμά μας δεν είναι πολύ καλή επιλογή). Αν προτιμήσεις να περάσεις το πλήθος πρόσεξε να μετρήσεις σωστά!

2. Η συνάρτησή σου θα πρέπει να έχει μια τουλάχιστον σταθερή παράμετρο. Είναι προφανές ότι οι σταθερές παράμετροι μπαίνουν στην αρχή.

3. Αν έχεις περισσότερες από μία σταθερές παραμέτρους, στην κλήση της *va_start() θα βάλεις ως δεύτερη παράμετρο την τελευταία σταθερή παράμετρο* που πρέπει να μην είναι παράμετρος αναφοράς.

4. Οι μη σταθερές παράμετροι θα πρέπει να είναι του ίδιου τύπου ή να εμφανίζουν κάποιο σταθερό σχήμα επανάληψης (π.χ. μια τύπου `char` μια τύπου `int`). Μόνον έτσι μπορείς να χρησιμοποιήσεις επαναληπτική εντολή (και έχει νόημα αυτό το είδος της συνάρτησης).

5. Οι πραγματικές παράμετροι θα πρέπει να συμφωνούν πλήρως με τους τύπους που βάζεις στην κλήση της *va_arg()*. Αν, στο παράδειγμά μας βάζαμε ως 3η παράμετρο όχι 2.0 αλλά 2 θα είχαμε πρόβλημα.

14.12 Συνοψίζοντας...

Κυρίως σε μαθηματικές εφαρμογές (αλλά όχι μόνον), παρουσιάζεται η ανάγκη γράψουμε συναρτήσεις με παράμετρο συνάρτηση. Η C++ λύνει αυτό το πρόβλημα με παράμετρο-βέλος-προς-συνάρτηση.

Πολύ συχνά παρουσιάζεται η ανάγκη να αλλάξουμε τύπους κάποιων μεταβλητών. Είναι πολύ σημαντικό να μην χρειάζεται να αλλάξουμε και τα ονόματα συναρτήσεων ή να

αλλάξουμε πράξεις με τελεστές σε κλήσεις συναρτήσεων. Η *επιφόρτωση* συναρτήσεων και τελεστών είναι η τεχνική με την οποία λύνουμε αυτά τα προβλήματα.

Αν οι συναρτήσεις που επιφορτώνονται διαφέρουν μόνο σε τύπους δεδομένων και δώσουμε στον μεταγλωττιστή ένα σχέδιο, το *περίγραμμα* (template), αυτός θα τις γράψει μόνος του.

Ας πούμε τώρα ότι για να λύσεις ένα συγκεκριμένο πρόβλημα γράφεις κάποια ή κάποιες συναρτήσεις που καταλαβαίνεις ότι θα σου είναι χρήσιμη/ες γενικότερα. Τα περιγράμματα συναρτήσεων και οι συναρτησιακές παράμετροι είναι δύο εργαλεία που σου επιτρέπουν να δώσεις την κατάλληλη παραμετρική μορφή ώστε να τα χρησιμοποιείς με ευκολία και στο μέλλον.

Πιθανότατα μεγαλύτερης σπουδαιότητας είναι ένα άλλο εργαλείο που είδαμε στο κεφάλαιο αυτό: οι *εξαιρέσεις*. Μαθαίνοντας να τις χρησιμοποιείς θα δεις ότι θα αλλάξει συνολικώς ο τρόπος που γράφεις τα προγράμματά σου: θα γίνονται πιο ευέλικτα, πιο λειτουργικά και –καθώς θα μαθαίνεις πώς να χειρίζεσαι τις εξαιρέσεις– πιο καλοσχεδιασμένα και με λιγότερα λάθη. Οι δομές (κλάσεις) εξαιρέσεων θα είναι ένα πολύ σοβαρό εργαλείο για την ανάπτυξη αλλά και τη συντήρηση του λογισμικού που γράφεις.

Τρία θέματα που ασχοληθήκαμε ακόμη ήταν

- Οι *συναρτήσεις* **inline**: με αυτές μπορείς να επιταχύνεις κάπως την εκτέλεση του προγράμματός σου.
- Η *αναδρομή*: ένα πολύ καλό εργαλείο που έχει όμως και μερικές παγίδες. Τώρα μπορείς να το χρησιμοποιείς «μετά λόγου γνώσεως».
- Το *ακαθόριστο πλήθος παραμέτρων*: Βάλαμε αυτήν την παράγραφο μόνο για να καταλαβαίνεις συναρτήσεις που έχουν γραφεί έτσι. *Μη χρησιμοποιείς αυτήν τη δυνατότητα.* Αντιθέτως, *η δυνατότητα να παραλείπεις παραμέτρους που έχουν ερήμην καθορισμένη τιμή είναι μια χαρά.*

Ασκήσεις

Α Ομάδα

14-1 (Γενίκευση της Ασκ.9-5) Γράψε μια

```
double vectorSumIf( const double x[], int n, bool (*predic)(double) )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία ισχύει η *predic*(x[*k*]).

Β Ομάδα

14-2 Θέλουμε συνάρτηση με όνομα *pluMin()* που

α) όταν καλείται με δύο ορίσματα *a*1, *a*2, τύπου `int`, θα αυξάνει την τιμή του πρώτου ορίσματος, *a*1, κατά 1 και θα μειώνει την τιμή του *a*2 κατά 1.

β) όταν καλείται με τέσσερα ορίσματα *a*1, *a*2, *b*1, *b*2 τύπου `int`, θα προσθέτει στην τιμή του *a*1 την τιμή του *b*1 και θα αφαιρεί από την τιμή του *a*2 την τιμή του *b*2.

Τι συνάρτηση θα γράψουμε, με τύπο ή χωρίς τύπο; Δικαιολόγησε την απάντησή σου.

Για κάθε μια παράμετρο δικαιολόγησε όλα τα χαρακτηριστικά της (τιμής ή αναφοράς, τύπο κλπ).

Γράψε τη συνάρτηση και δώσε παράδειγμα χρήσης με 2 και 4 ορίσματα. Στην κάθε περίπτωση θα βάζεις μέσα σε σχόλια τις τιμές των μεταβλητών (που χρησιμοποιούνται) πριν και μετά την κλήση.

Η συνάρτηση που έγραψες μπορεί να κληθεί με 3 ορίσματα; Δικαιολόγησε την απάντησή σου· αν είναι θετική δώσε παράδειγμα όπως παραπάνω.

14-3 (Να την απαντήσεις εσύ και *όχι ο μεταγλωττιστής C++ που χρησιμοποιείς*.) Έστω ότι έχουμε:[23]

```
template<typename T1, typename T2>
void f( T1, T2 );                       // 1
template<typename T> void f( T );       // 2
template<typename T> void f( T, T );    // 3
template<typename T> void f( T* );      // 4
template<typename T> void f( T*, T );   // 5
template<typename T> void f( T, T* );   // 6
template<typename T> void f( int, T* ); // 7
template<> void f<int>( int );          // 8
void f( int, double );                  // 9
void f( int );                          // 10
```

Αν

```
int             i;
double          d;
float           ff;
```

ποια από τις παραπάνω θα κληθεί για την κάθε μια από τις:

```
f( i );         // a
f<int>( i );    // b
f( i, i );      // c
f( i, ff );     // d
f( i, d );      // e
f( i, &d );     // f
f( &d, d );     // g
f( &d );        // h
f( d, &i );     // i
f( &i, &i );    // j
```

Γ Ομάδα

14-4 Όπως είναι γνωστό, το ορισμένο ολοκλήρωμα: $\int_a^b f(x)dx$ υπολογίζει το εμβαδό που περικλείεται ανάμεσα στον άξονα *x'x* και στην καμπύλη *y* = *f* (*x*), από *x* = *a* μέχρι *b*. Μια αριθμητική μέθοδος για να προσεγγίσουμε την τιμή του ολοκληρώματος είναι αυτή του **μέσου** (midpoint):

Διαιρούμε το διάστημα [*a*,*b*] σε *N* ίσα τμήματα που το καθένα έχει μήκος $h = (b - a)/N$. Κατασκευάζουμε *N* παραλληλόγραμμα, που το καθένα έχει βάση *h* και ύψος $f(x_k)$, όπου $x_k = a + (k-0.5)h$, είναι το μέσο του *k* διαστήματος. Το άθροισμα των εμβαδών των παραλληλογράμμων μας δίνει μια προσέγγιση του ολοκληρώματος.

$$\int_a^b f(x)dx \approx h\sum_{k=1}^{N} f(x_k)$$

Γράψε μια συνάρτηση *midPoint* που θα τροφοδοτείται με τη συνάρτηση που έχουμε να ολοκληρώσουμε, τα άκρα του διαστήματος της ολοκλήρωσης και το *N* και θα υπολογίζει και θα επιστρέφει την προσέγγιση της τιμής του ολοκληρώματος με τη μέθοδο του μέσου. Τέλος θα έχει μία ακόμη παράμετρο, τη *errCode*· αν η *midPoint* κληθεί με *N* < 1, δεν θα γίνονται υπολογισμοί και η *errCode* θα επιστρέφει τιμή 1, αλλιώς, αν όλα πάνε καλά, θα επιστρέφει τιμή 0.

14-5 Στην §14.3, λέγαμε: Αν m_0 είναι το μέσο του διαστήματος [a_0,b_0], ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
```

[23] Από το (Sutter 1998).

```
    αλλιώς
       ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

και γράψαμε σε C++:

```
    a = a0;  b = b0;
    m = (a + b) / 2;
    if (f(a)*f(m) <= 0.0)  b = m;
                   else  a = m;
```

Αλλά, το «ψάχνουμε για τη λύση στο διάστημα [a_0,m_0]» μπορεί να μεταφρασθεί και ως εξής: «κάλεσε τον εαυτό σου για να ψάξει λύση στο διάστημα [a_0, m_0]». Παρομοίως μπορεί να μεταφρασθεί και το «ψάχνουμε για τη λύση στο διάστημα [m_0,b_0]». Με βάση αυτές τις παρατηρήσεις γράψε μια *αναδρομική* μορφή της *bisection*.

14-6 Με βάση το πρόγραμμα για τη συγχώνευση ταξινομημένων πινάκων (§9.4) γράψε περίγραμμα συνάρτησης:

```
template<typename T> void merge( T v[], T w[], T z[],
                                 int pv, int tv, int pw, int tw,
                                 int pz, int tz,
                                 bool& ok )
```

που θα συγχωνεύει τα τμήματα v[*pv*]...v[*tv*] και w[*pw*]...w[*tw*] των πινάκων v, w στο z[*pz*]...z[*tz*]. Προφανώς θα πρέπει να έχουμε $0 \le pv \le tv$, $0 \le pw \le tw$ και $tz - pz + 1 \ge (tv - pv + 1) + (tw - pw + 1)$. Αν δεν ισχύουν οι παραπάνω, η *ok* επιστρέφει `false`. Τα τμήματα v[*pv*]...v[*tv*] και w[*pw*] ... w[*tw*] πρέπει να είναι ταξινομημένα κατ' αύξουσα τάξη. Την ίδια ταξινόμηση θα έχει και το z[*pz*]... z[*tz*].

14-7 Στο τέλος της §9.6 δίναμε μια άλλη ιδέα για ταξινόμηση:

- Κόβουμε τον πίνακά μας στη μέση και έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε ένα από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικά: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

  Με τις διαδικασίες που γράψαμε, αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα απ' αυτό, δεν μπορούμε να μη σκεφτούμε: «*γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;*» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log(N)$, που φυσικά είναι καλύτερος από N^2.

Γράψε και δοκίμασε τη *mergeSort()* για πίνακες με στοιχεία τύπου *string*. Μετάτρεψε τη συνάρτηση που έγραψες σε περίγραμμα.

κεφάλαιο

15

Δομές – Αρχεία ΙΙ

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις

- να ομαδοποιείς δεδομένα που αναφέρονται σε μια φυσική οντότητα σε ένα *αντικείμενο*, με τον «παλιό» τρόπο, της C, κατά βάση[1],
- να τα φυλάγεις σε *μη-μορφοποιημένα αρχεία*,
- να διαχειρίζεσαι ένα (μη-μορφοποιημένο) αρχείο με *τυχαία πρόσβαση* στις συνιστώσες του,
- να χρησιμοποιείς *δομές εξαιρέσεων*.

Προσδοκώμενα αποτελέσματα:

- Θα κάνεις την πρώτη γνωριμία με την αντικειμενοστρέφεια.
- Θα μάθεις να χρησιμοποιείς τύπους (δομές) εξαιρέσεων που έχουν νόημα.
- Θα μάθεις να χρησιμοποιείς μη-μορφοποιημένα αρχεία και να κρίνεις πότε να τα χρησιμοποιήσεις.
- Θα μάθεις να χρησιμοποιείς αρχεία τυχαίας πρόσβασης.

Έννοιες κλειδιά:

- *δομή - αντικείμενο*
- *μέλος δομής - μέλος αντικειμένου*
- *δημιουργός*
- *ερμηνευτική τυποθεώρηση* - `reinterpret_cast`
- *μή μορφοποιημένο αρχείο*
- *αρχείο τυχαίας πρόσβασης*
- *μέθοδοι seek - μέθοδοι tell*

Περιεχόμενα:

15.1 Δομές 432
15.1.1 Παράμετρος – Δομή 436
15.2 Μέλη Δομής 436
15.3 Δημιουργοί 437
15.3.1 Αποκάλυψη Τώρα! 439
15.4 Βέλος προς Τιμή-Δομή 439
15.5 Επιφόρτωση Τελεστών για Τύπους Δομών 440
15.5.1 Συγκρίσεις και Κλειδιά 441
15.6 Αποθήκευση Μελών Δομής 442
15.6.1 * Σκαλίζοντας τη Μνήμη 442

[1] Ο τρόπος της C++; Στο *Μέρος Γ*.

15.7 Ερμηνευτική Τυποθεώρηση **444**
15.8 * Ψηφιοπεδία **447**
15.9 * union **448**
15.10 Δομές για Εξαιρέσεις **451**
15.11 Μη Μορφοποιημένα Αρχεία **454**
15.12 Τυχαία Πρόσβαση σε Αρχεία - Μέθοδοι *seek* και *tell* **457**
15.12.1 Πώς Βρίσκουμε το Μέγεθος Αρχείου **459**
15.13 Τιμή-Δομή σε Μη Μορφοποιημένο Αρχείο **460**
15.13.1 * Να Προτιμήσουμε τον Τύπο *string*; **461**
15.14 Ένα Παράδειγμα **462**
15.14.1 Το Πρώτο Πρόγραμμα **463**
15.14.2 Το Δεύτερο Πρόγραμμα **467**
15.14.3 Για το Παράδειγμά μας **472**
15.15 Ανακεφαλαίωση **474**
Ασκήσεις **474**
Β Ομάδα **474**
Γ Ομάδα **475**

Εισαγωγικές Παρατηρήσεις:

E Pluribus Unum

Μια από τις εμβληματικές φράσεις των ΗΠΑ είναι η «*E Pluribus Unum*». Σημαίνει «από πολλά ένα» και εννοεί ότι πολλές πολιτείες αποτελούν ένα κράτος.

Και τι σχέση έχει αυτό με αυτά που μας ενδιαφέρουν; Στο κεφάλαιο αυτό θα ασχοληθούμε με το πώς μπορούμε να χειριζόμαστε ως ένα αντικείμενο πολλά στοιχεία που αναφέρονται σε μια οντότητα. Ένας πίνακας έχει –κατ' αρχήν– τέτοια χαρακτηριστικά αλλά με έναν βασικό περιορισμό: *όλα τα στοιχεία του πίνακα είναι του ίδιου τύπου*. Αν θέλεις να έχεις ένα αντικείμενο που να περιγράφει έναν άνθρωπο θα πρέπει να μπορεί να κρατήσει επώνυμο, όνομα, ημερομηνία γέννησης, τόπο γέννησης, διεύθυνση κατοικίας, αριθμό τηλεφώνου κλπ. Ο πίνακας δεν μπορεί να φιλοξενήσει αυτά τα στοιχεία.

Η ανάγκη για ομαδοποίηση στοιχείων που αναφέρονται στο ίδιο φυσικό αντικείμενο παρουσιάστηκε πολύ νωρίς στις εμπορικές εφαρμογές. Δεν είναι λοιπόν περίεργο το ότι πρωτοπόρος στον τομέα ήταν η COBOL (COmmon Business Oriented Language) –και αργότερα τη μιμήθηκε η PL/I– ενώ Algol-60 και FORTRAN δεν είχαν οτιδήποτε σχετικό.

Η Pascal έδωσε τη δυνατότητα στον προγραμματιστή να ορίσει **τύπους εγγραφών** (record types) σύμφωνα με τις ανάγκες του, με απλό και κομψό τρόπο. Η Ada σχεδόν αντίγραψε την Pascal.

Οι **δομές** (`struct`(ure)s) της C μοιάζουν πολύ με τις εγγραφές της Pascal. Η C++ είδε τις δομές ως «*κλάσεις με όλα τα μέλη ανοικτά*». Η Java τις αγνόησε: επιτρέπει μόνον κλάσεις. Η C# τις είδε ως *κλάσεις με όλα τα μέλη ανοικτά που δεν κληρονομούν ούτε κληρονομούνται.*

Τιμές τέτοιων τύπων μπορούμε να βάλουμε και σε αρχεία. Τώρα πρόσεξε: αν τις γράψουμε (σχεδόν) όπως είναι αποθηκευμένες στη μνήμη (εσωτερική παράσταση) όλες θα καταλαμβάνουν τον ίδιο χώρο. Αυτό όμως μας δίνει τη δυνατότητα να έχουμε και *τυχαία* (και όχι μόνον σειριακή) *πρόσβαση* στις συνιστώσες του αρχείου. Αξίζει λοιπόν τον κόπο να δούμε τα μη μορφοποιημένα αρχεία και να καταλάβουμε τα μειονεκτήματα και τα πλεονεκτήματά τους.

15.1 Δομές

Είδαμε ότι οι πίνακες μας δίνουν έναν τρόπο πρόσβασης σε πολλά στοιχεία με ένα όνομα. Αλλά, τα στοιχεία αυτά πρέπει να είναι όλα του ίδιου τύπου.

Πολύ συχνά όμως, θέλουμε να αποθηκεύουμε και να επεξεργαζόμαστε στοιχεία διαφορετικού τύπου που αναφέρονται στο ίδιο αντικείμενο.

Παράδειγμα 1 ✍

Ένα πρόγραμμα μισθοδοσίας θα πρέπει να χρησιμοποιεί στοιχεία όπως:

επώνυμο
όνομα
ηλικία
βαθμός
ημερομηνία πρόσληψης
οικογενειακή κατάσταση
αριθμός παιδιών κ.λ.π.

Αυτά, ενώ όλα αναφέρονται σε ένα μισθωτό, δεν είναι του ίδιου τύπου ώστε να μπορούν να μπούν σε έναν πίνακα και να τα επεξεργαστούμε με έναν ενιαίο τρόπο.

Η C++ μας διευκολύνει με μία κατηγορία τύπων που προσφέρει γι' αυτές τις περιπτώσεις: τις **κλάσεις** (class). Η απλούστερη περίπτωση κλάσης είναι η **δομή** (structure) και με αυτήν θα αρχίσουμε τη γνωριμία μας με τη σύγχρονη τεχνική προγραμματισμού που λέγεται **αντικειμενοστρέφεια** (object orientation). Έτσι, για το παραπάνω παράδειγμα, μπορούμε να ορίσουμε τον τύπο:

```
struct Employee
{
   char    surname[24];
   char    firstname[16];
   Date    birthDate;
   int     rank;
   Date    emplDate;
   int     numberOfChld;
   Address home;
}; // Employee
```

Αν στην συνέχεια δηλώσουμε:

```
   Employee clerk, typist, manager;
```

μπορούμε μέσα στο πρόγραμμά μας να δίνουμε:

```
   strcpy( clerk.surname, "ΝΙΚΟΛΟΠΟΥΛΟΣ" );
```

ή

```
   if ( manager.birthDate < clerk.birthDate )
   {
      megalhYpotesh();
      grafto( clerk );
   }
   else
      tiPerimenes();
```

✍✍✍

Μια μεταβλητή (ή σταθερά) που ο τύπος της είναι δομή (ή γενικότερα: κλάση) ονομάζεται και **αντικείμενο** (object) αυτής της δομής (κλάσης).

Κι εδώ λοιπόν, όπως στους πίνακες, με το ίδιο όνομα αναφερόμαστε σε μία ομάδα μεταβλητών. Αντί όμως να χρησιμοποιήσουμε τις αγκύλες με το δείκτη για να τις ξεχωρίσουμε, βάζουμε, μετά το όνομα του αντικειμένου, μια τελεία (`.`) και το όνομα της μεταβλητής.

Οι μεταβλητές που εμφανίζονται στη δήλωση μιας δομής, όπως οι *surname*, *birthDate*, *numberOfChld*, *rank*, λέγονται **μέλη** (members)[2] της δομής.

Η περιγραφή μιας δομής ξεκινάει με τα
"`struct`", *όνομα*, "`{`"
και τελειώνει με "`};`". Ενδιαμέσως γράφονται τα μέλη του τύπου, όπως ακριβώς γράφονται οι δηλώσεις μεταβλητών:

---

[2] Γενικώς, στην Επεξεργασία Δεδομένων ονομάζονται **πεδία** (fields). Η C++ έχει για τον όρο αυτόν άλλη χρήση.

τύπος, *όνομα*, ";"

Παράδειγμα 2

Στην §8.13 είδαμε ότι για τον χειρισμό ρευμάτων αρχείων η C χρησιμοποιεί βέλη προς μεταβλητές τύπου *FILE*. Το πρότυπο της γλώσσας (ISO/IEC 1999) λέει ότι μια τέτοια μεταβλητή «*θα πρέπει να έχει τη δυνατότητα να καταγράφει όλην την πληροφορία που χρειάζεται για τον έλεγχο ρεύματος περιλαμβάνοντας ενδείκτη θέσης αρχείου, βέλος προς τον αντίστοιχο ενταμιευτή (αν υπάρχει), ενδείκτη σφάλματος που καταγράφει τον άν έγινε σφάλμα ανάγνωσης/εγγραφής, ενδείκτη τέλους αρχείου που καταγράφει αν έφτασε το τέλος αρχείου.*»

Η Borland C++, v.5.5 υλοποιεί τον τύπο ως εξής:

```
struct FILE
{
   unsigned char* curp;        // ενδείκτης θέσης αρχείου
   unsigned char* buffer;      // βέλος προς ενταμιευτή
   int            level;       // επίπεδο πληρότητας ενταμιευτή
   int            bsize;       // μέγεθος ενταμιευτή
   unsigned short istemp;      // ενδείκτης προσωρινότητας αρχείου
   unsigned short flags;       // σημαίες κατάστασης αρχείου
   wchar_t        hold;        // για ungetc αν δεν υπάρχει
                               // ενταμιευτής
   char           fd;          // File descriptor
   unsigned char  token;       // Used for validity checking
}; // FILE
```

Παραδείγματα 3

```
struct complex
{
   double re;
   double im;
}; // complex
```

Εδώ έχουμε μια δομή που κάθε αντικείμενό της θα έχει δυο μέλη τύπου **double**. Το ένα με το όνομα *re* και το άλλο με το όνομα *im*. Όπως φαίνεται και από το όνομα (complex = μιγαδικός), τα αντικείμενα της δομής *complex* μπορούν να παραστήσουν μιγαδικούς αριθμούς. Θα μπορούσαμε να τη γράψουμε και ως:

```
struct complex
{
   double re, im;
}; // complex
```

αλλά ο πρώτος τρόπος είναι προτιμότερος.

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
}; // Date
```

Οι μεταβλητές αυτού του τύπου κρατούν ημερομηνίες ενώ του επόμενου κρατούν διευθύνσεις:

```
struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address
```

Μετά τη δήλωση της δομής μπορούμε να δηλώσουμε μεταβλητές (αντικείμενα) της δομής. Μετά τις δηλώσεις των τύπων που είδαμε παραπάνω μπορούμε να δηλώσουμε:

```
complex i, j, k;
Date    yesterday, today, tomorrow;
Address residence;
```

Σημείωση: ▶

Όπως είπαμε, η **struct** υπήρχε και στη C, από όπου την κληρονόμησε (και την εμπλούτισε) η C++. Θα τη βρεις λοιπόν πολύ συχνά μπροστά σου. Αλλά εκεί ο χειρισμός των ορισμών και δηλώσεων είναι κάπως διαφορετικός. Π.χ., για να μπορέσεις να κάνεις τη δήλωση:

```
complex i, j, k;
```

θα πρέπει να έχεις ορίσει: [3]

```
typedef struct {  double re;  double im;  }  complex;
```

◀

Μια δομή μπορεί να έχει μέλη τύπου πίνακα ή δομής· π.χ. στον τύπο *Employee* έχουμε δύο μέλη δομής *Date* και ένα μέλος δομής *Address*, ενώ ο τύπος των *surname*, *firstname* είναι πίνακες.

Τέλος, όπως στις απλές μεταβλητές και τους πίνακες έτσι και στα αντικείμενα μπορείς να δώσεις αρχική τιμή με τη δήλωση, π.χ.:

```
complex j = { 0, 1 }, isqrt2 = { 1/sqrt(2), 1/sqrt(2) };
Date d1 = { 2004, 11, 17 };
```

Μετά από αυτό, οι:

```
cout << "( " << j.re << ", " << j.im << " )" << endl;
cout << "( " << isqrt2.re << ", " << isqrt2.im << " )" << endl;
cout << int(d1.day) << '.' << int(d1.month)
     << '.' << d1.year << endl;
```

θα δώσουν:

```
( 0, 1 )
( 0.707107, 0.707107 )
17.11.2014
```

Δηλαδή, με τη δήλωση: **d1 = { 2004, 11, 17 }** η πρώτη τιμή (2004) πηγαίνει στο πρώτο μέλος (*d1.year*), η δεύτερη (11) στο δεύτερο (*d1.month*) κ.ο.κ.

Μπορείς να διαχειρίζεσαι τα αντικείμενα είτε ανά μέλος, όπως θα δούμε στην επόμενη παράγραφο, είτε ολόκληρα: την τιμή μιας τέτοιας μεταβλητής

- μπορείς να την εκχωρείς σε μιαν άλλη, του ίδιου τύπου,
- μπορείς να τη δίνεις ως παράμετρο στην κλήση κάποιας συνάρτησης,
- μπορείς να τη βάζεις σε εντολή **return** και να επιστρέφεται ως τιμή συνάρτησης (του ίδιου τύπου).

Παράδειγμα 4 🗲

Η συνάρτηση:

```
complex conj( complex c )
{
   complex cj = { c.re, -c.im };
   return cj;
} // conj
```

επιστρέφει τον συζυγή του ορίσματος. Οι:

```
k = conj( j );
cout << "( " << k.re << ", " << k.im << " )" << endl;
```

θα δώσουν:

---

[3] Αν κρατήσεις τον αρχικό ορισμό θα πρέπει να δηλώσεις τις μεταβλητές ως:

```
struct complex i, j, k;
```

```
( 0, -1 )
```

15.1.1 Παράμετρος – Δομή

Μπορείς να δώσεις ένα αντικείμενο «*ως παράμετρο στην κλήση κάποιας συνάρτησης*» αλλά πρέπει να λάβεις υπόψη σου και μερικά πράγματα σχετικώς με τη διαδικασία περάσματος των παραμέτρων (§13.3, 14.8).

Δες την *conj()* και την κλήση "`k = conj(j)`"· για την εκτέλεση της κλήσης δημιουργείται μια τοπική μεταβλητή *c* –στη στοίβα– με αρχική τιμή ίση με αυτήν της *j*. Στην περίπτωσή αυτήν η αντιγραφή της *j* στη *c* σημαίνει αντιγραφή 16 ψηφιολέξεων (ή κάτι παρόμοιο). Αν όμως το αντικείμενό μας είναι μεγάλο η αντιγραφή μπορεί

- να μεγαλώσει τον χρόνο εκτέλεσης του προγράμματός μας και
- να φορτώσει πολύ τη στοίβα.

Έτσι, αντί για παράμετρο τιμής επιλέγουμε γενικώς την εξής λύση:

```
complex conj( const complex& c )
{
   complex cj = { c.re, -c.im };
   return cj;
} // conj
```

Δηλαδή παράμετρο αναφοράς με "`const`" ωστε να μην επιτρέπεται η αλλαγή τιμής του ορίσματος. Με αυτόν τρόπο όταν καλείται η συνάρτηση αντιγράφεται μόνον ένα βέλος.

Εδώ θα πρέπει να κάνουμε και μια συμπλήρωση του κανόνα που δώσαμε στην §13.3:

♦ ***Η πραγματική παράμετρος που αντιστοιχεί σε μια παράμετρο αναφοράς χωρίς "`const`" δεν μπορεί να είναι σταθερά ή παράσταση, αλλά κάτι που να μπορεί να αλλάξει η τιμή του δηλαδή τροποποιήσιμη τιμή-l, π.χ. μια μεταβλητή ή ένα στοιχείο πίνακα.***

Αν η παράμετρος αναφοράς έχει "`const`" τότε το αντίστοιχο όρισμα μπορεί να είναι «*σταθερά ή παράσταση*».

Δηλαδή η παράμετρος αναφοράς με "`const`" μπορεί να υποκαταστήσει πλήρως την παράμετρο τιμής; Σχεδόν·

- Αν σου χρειάζεται, μπορείς να χρησιμοποιήσεις την παράμετρο τιμής ως τοπική μεταβλητή και να αλλάξεις την τιμή της χωρίς να αλλάξει η τιμή του ορίσματος.
- Η τιμή παραμέτρου αναφοράς με "`const`" δεν επιτρέπεται να αλλάξει.

15.2 Μέλη Δομής

Στην προηγούμενη παράγραφο είδαμε το μέλος ως **χαρακτηριστικό** (attribute) μιας δομής. Είπαμε όμως και ότι μπορούμε να χειριζόμαστε τα αντικείμενα μιας δομής ανά μέλος.

♦ ***Μπορούμε να μεταχειριζόμαστε κάθε μέλος ενός αντικειμένου όπως οποιαδήποτε μεταβλητή του τύπου του.***

Όπως είπαμε και στην εισαγωγή αυτού του κεφαλαίου, ένα μέλος αναφέρεται με το όνομα του αντικειμένου, μια τελεία και το όνομα του μέλους.

Παράδειγμα 1

Με βάση τα παραδείγματα της προηγούμενης παραγράφου έχουν νόημα τα:

```
i.re
j.im
today.month
residence.city
```

Στον τύπο *complex* υπάρχουν τα μέλη *re* και *im*. Τα `i.re` και `j.im` είναι μέλη των μεταβλητών *i*, *j* τύπου *complex*. Παρομοίως, τα *month* και *city* είναι μέλη των δομών *Date* και

Address αντιστοίχως. Αφού έχουμε δηλώσει: `Date today` και `Address residence` τα `today.month` και `residence.city` είναι μέλη των μεταβλητών *today* και *residence* αντιστοίχως.

Με βάση τον ορισμό της δομής *Employee* στην προηγούμενη παράγραφο, μπορούμε να έχουμε:

```
clerk.emplDate
```

Αυτό το μέλος μπορούμε να το χειριζόμαστε ως μεταβλητή τύπου *Date*. Φυσικά μπορούμε να έχουμε και μέλη αυτού του αντικειμένου:

```
clerk.emplDate.year
clerk.emplDate.day
```

Η **εμβέλεια** (scope) του ονόματος ενός μέλους είναι η δομή όπου ορίζεται. Έτσι, μπορείς να χρησιμοποιείς ως όνομα μέλους ένα όνομα που το χρησιμοποιείς και αλλού στο πρόγραμμά σου, χωρίς να υπάρχει κίνδυνος σύγχυσης του μεταγλωττιστή. Πάντως *υπάρχει κίνδυνος σύγχυσης των προγραμματιστών*: κάτι τέτοιο μπορεί να κάνει το πρόγραμμά σου λιγότερο ευανάγνωστο. Είναι λοιπόν προτιμότερο να το αποφεύγεις.

Ας δούμε τώρα την εντολή εκχώρησης για αντικείμενα. Αν έχουμε ορίσει:

```
struct Q
{
   T1 x1;  T2 x2; ... Tn xn;
}; // Q
```

και δηλώσουμε:

```
Q a, b;
```

τότε η εντολή

```
a = b;
```

είναι ισοδύναμη με τις:

```
a.x1 = b.x1;  a.x2 = b.x2; ... a.xn = b.xn;
```

Παράδειγμα 2

Η εντολή:

```
today = tomorrow;
```

(δες την προηγούμενη παράγραφο) ισοδυναμεί με τις:

```
today.day   = tomorrow.day;
today.month = tomorrow.month;
today.year  = tomorrow.year;
```

15.3 Δημιουργοί

Είδαμε πώς μπορούμε να δίνουμε αρχικές τιμές σε μια μεταβλητή-δομή:

```
complex j = { 0, 1 }, isqrt2 = { 1/sqrt(2), 1/sqrt(2) };
Date d1 = { 2004, 11, 17 };
```

Αυτό το στυλ είναι κληρονομιά από τη C. Μπορούμε να χρήσιμοποιούμε παρενθέσεις όπως κάνουμε με τις άλλες μεταβλητές της C++; Ναι, αρκεί να ορίσουμε έναν δημιουργό.

Ο **δημιουργός** ή **κατασκευαστής** (constructor) είναι μια συνάρτηση που περιγράφει τι πρέπει να γίνει για να δημιουργηθεί μια τιμή ενός τύπου. Το όνομά της είναι το όνομα του τύπου και δεν έχει (άλλον) τύπο αφού είναι μέρος του ορισμού του τύπου.

Ας ορίσουμε έναν δημιουργό που θα μας επιτρέπει να δηλώσουμε:

```
complex j( 0, 1 ), isqrt2( 1/sqrt(2), 1/sqrt(2) );
```

Αυτό γίνεται με συμπλήρωση του ορισμού της δομής ως εξής:

```
struct complex
```

```
{
   double re;
   double im;
   complex( double rp, double ip )  {  re = rp;  im = ip;  };
}; // complex
```

Εδώ έχουμε έναν πολύ απλό δημιουργό: το μόνο που κάνει είναι να βάζει το πρώτο όρισμά του στο μέλος *re* και το δεύτερο στο μέλος *im*.

Μπορείς να δοκιμάσεις τη δήλωση με τις παρενθέσεις και θα δεις ότι «περνάει». Μπορείς όμως να δεις ότι χάσαμε τη δυνατότητα να κάνουμε τη δήλωση: `complex q`. Αυτό διορθώνεται με έναν δεύτερο δημιουργό:

```
struct complex
{
   double re;
   double im;
   complex( double rp, double ip )  {  re = rp;  im = ip;  };
   complex() {  re = 0.0;  im = 0.0;  };
}; // complex
```

Αυτό που ορίζουμε είναι ότι αν δεν δοθούν από τον προγραμματιστή αρχικές τιμές βάζουμε *ερήμην* αρχικές τιμές μηδέν και στα δύο μέλη. Αυτός είναι ο λεγόμενος **ερήμην δημιουργός** (default constructor). Αυτός ο δημιουργός θα κληθεί όταν δηλώνουμε και έναν πίνακα με στοιχεία τύπου *complex* (μια φορά για κάθε στοιχείο).

Γενικώς, μπορούμε να ορίζουμε πολλούς δημιουργούς, αλλά αυτούς τους δύο μπορούμε να τους συγχωνεύσουμε σε έναν. *Ο δημιουργός είναι συνάρτηση*· ειδική περίπτωση συνάρτησης αλλά συνάρτηση. Μπορούμε λοιπόν να χρησιμοποιήσουμε προκαθορισμένες τιμές παραμέτρων (§14.2):

```
struct complex
{
   double re;
   double im;
   complex( double rp=0.0, double ip=0.0 )
   {  re = rp;  im = ip;  };
}; // complex
```

Τώρα έχουμε έναν δημιουργό «2 σε 1»: έναν ερήμην δημιουργό που όμως μπορεί να δεχθεί και αρχικές τιμές.

Πρόσεξε και μια άλλη χρήση του δημιουργού με αρχικές τιμές: Δηλώνουμε

```
   complex q;
```

και στη συνέχεια της δίνουμε τιμή ως εξής:

```
   q = complex( 1.7, 8.1 );
```

Δηλαδή, καλούμε τον δημιουργό για να δημιουργήσει μια τιμή τύπου *complex* και στη συνέχεια την εκχωρούμε στην *q*. Έτσι, θα μπορούσαμε να ξαναγράψουμε την *conj* πιο απλά:

```
complex conj( complex c )
{
   return complex( c.re, -c.im );
} // conj
```

Και για την κλάση *Date* θα μπορούσαμε να ορίσουμε:

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
   Date( int yp=1, int mp=1, int dp=1 )
   {  year = yp;  month = mp;  day = dp;  }
}; // Date
```

Αυτός ο δημιουργός μας δίνει το δικαίωμα να δηλώσουμε:

```
   Date d0( 2011, 3, 5 )  // d0 == 05.03.2011
   Date d1;               // d1 == 01.01.0001
```

αλλά και:

```
Date d2( 2011, 3 )     // d2 == 01.03.2011
Date d3( 2011 );       // d3 == 01.01.2011
```

Πάντως, εδώ να επισημάνουμε ένα πρόβλημα: Ο δημιουργός δεν μας απαγορεύει να δηλώσουμε:

```
Date d1( 2004, 14, 37 );
```

Θα πρέπει να τον εφοδιάσουμε με μερικούς ελέγχους ώστε να απαγορεύει τέτοια λάθη.

Περισσότερα για τους δημιουργούς αργότερα, όταν θα συζητούμε για κλάσεις.

15.3.1 Αποκάλυψη Τώρα!

Πώς κάναμε το παράδειγμα της §11.1; Ορίζοντας τον τύπο:

```
struct MyType
{
   int v;
   MyType( int a=0 )
   {
      cout << "creating a MyType variable. Initialize with "
           << a << endl;
      v = a;
   }
   ~MyType()
   {
      cout << "destroying a MyType variable having value "
           << v << endl;
   }
}; // MyType
```

Ναι, αλλά εδώ έχει καινούρια πράγματα· τι είναι αυτό το *~MyType()*; Αυτή η συνάρτηση είναι ο **καταστροφέας** (destructor) της *MyType*. Καλείται για να καταστρέψει μια μεταβλητή τύπου *MyType*. Αν ξεχάσουμε το παράδειγμά μας με τα μηνύματα καταστροφής, η *MyType* δεν χρειάζεται καταστροφέα διότι η καταστροφή γίνεται αυτομάτως. Σε άλλες περιπτώσεις όμως είναι απαραίτητος.

15.4 Βέλος προς Τιμή–Δομή

Πολύ συχνά θα χρειαστεί να χειριστούμε κάποια τιμή-δομή με βέλος προς αυτήν. Η C++ μας δίνει μια μικρή διευκόλυνση για τον χειρισμό των μελών.

Ας πούμε ότι δηλώνουμε:

```
complex  q;
complex* pC( &q );
```

Μπορούμε να δώσουμε τιμές στα μέλη της *q* μέσω του *pC* ως εξής:

```
(*pC).re = 2.5;  (*pC).im = 1.03;
```

και να γράψουμε τις τιμές τους ως εξής:

```
cout << "*pC = ( " << (*pC).re << ", " << (*pC).im << " )"
     << endl;
```

Μέχρι εδώ τίποτε περίεργο: με την αποπαραπομπή (`*pC`) παίρνουμε τη μεταβλητή (*q*) που δείχνει το βέλος *pC*. Από εκεί και πέρα χρησιμοποιούμε την τελεία για να ξεχωρίσουμε τα μέλη.

Η C++ μας επιτρέπει να κάνουμε τα παραπάνω ως εξής:

```
pC->re = 2.5;  pC->im = 1.03;
cout << "*pC = ( " << pC->re << ", " << pC->im << " )" << endl;
```

Τι κερδίσαμε; Αντί να γράφουμε για κάθε μέλος "`(*).`" γράφουμε: "`->`".

Θα βλέπεις συχνά αυτόν τον συμβολισμό και –παρ' όλο που μπορείς να ζήσεις χωρίς αυτόν– καλά θα κάνεις να τον χρησιμοποιείς.

15.5 Επιφόρτωση Τελεστών για Τύπους Δομών

Αν έχουμε ορίσει, όπως πριν:

```
struct Q
{
   T1 x1;  T2 x2; ... Tn xn;
}; // Q
```

και δηλώσουμε:

```
Q a, b;
```

τότε η σύγκριση

```
a == b
```

είναι, κατ' αρχήν, ισοδύναμη με τις:

```
(a.x1 == b.x1) && (a.x2 == b.x2) && ... && (a.xn == b.xn)
```

Αυτή σύγκριση δεν γίνεται αυτομάτως. Για κάθε τύπο δομής θα πρέπει να επιφορτώνεις τον τελεστή "`==`" αν τον χρειάζεσαι. Πώς θα γίνει αυτό, ας πούμε για τον τύπο `complex`; Όπως είπαμε στην §14.6.4, δηλαδή:[4]

```
bool operator==( const complex& lhs, const complex& rhs )
{
   return lhs.re == rhs.re && lhs.im == rhs.im;
} // bool operator==( const complex
```

και μπορείς να κάνεις συγκρίσεις όπως:

```
if ( isqrt2 == j ) . . .
```

Για τον "`==`" (και τους άλλους τελεστές σύγκρισης) θα τα ξαναπούμε στη συνέχεια.

Παρομοίως θα επιφορτώσεις και τις αριθμητικές πράξεις "`+`", "`-`", "`*`", "`/`".

Ας πούμε τώρα ότι θέλεις να επιφορτώσεις το πρόσημο "`-`" που είναι ενικός (προθεματικός) τελεστής. Σύμφωνα με αυτά που λέμε στην §14.6.4, θα έχουμε:

```
complex operator-( const complex& lhs )
{
   return complex( -lhs.re, -lhs.im );
} // operator-( const complex
```

Ας πούμε τώρα ότι θέλουμε να μπορούμε να γράφουμε:

```
cout << isqrt2 << endl;
```

και να έχουμε το ίδιο αποτέλεσμα που έχουμε με την:

```
cout << "( " << isqrt2.re << ", " << isqrt2.im << " )" << endl;
```

Θα επιφορτώσουμε τον τελεστή "`<<`" όπως μάθαμε στην §14.6.1:

```
ostream& operator<<( ostream& tout, const complex rhs )
{
   return tout << "( " << rhs.re << ", " << rhs.im << " )";
} // operator<<( ostream& tout, const complex
```

Μπορούμε να τον επιφορτώσουμε και για τη *Date* ώστε η:

```
cout << d1 << endl;
```

να κάνει το ίδιο με την

```
cout << int(d1.day) << '.' << int(d1.month)
     << '.' << d1.year << endl;
```

Και αυτή η επιφόρτωση είναι απλή:

---

[4] Κατά τη συνήθειά μας, θα υπενθυμίσουμε ότι σύγκριση για ισότητα στον τύπο `double` δεν έχει και πολύ νόημα.

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << static_cast<unsigned int>(rhs.day) << '.'
               << static_cast<unsigned int>(rhs.month) << '.'
               << rhs.year;
} // operator<<( ostream& tout, const Date
```

Να τον επιφορτώσουμε και για τον τύπο *Address*; Δοκίμασέ το, αλλά δεν θα σου αρέσει!...

Στην §15.1 έχουμε τη σύγκριση "`manager.birthDate < clerk.birthDate`". Για να μπορεί να γίνει θα πρέπει να επιφορτώσουμε τον τελεστή "`<`" για τον τύπο *Date*. Αν έχουμε

```
Date d1, d2;
```

η σύγκριση `d1 < d2` θα πρέπει να δίνει τιμή `true` μόνο στην περίπτωση που η ημερομηνία *d*1 προηγείται της *d*2.

Ακολουθώντας αυτά που λέγαμε στην§14.6.4, θα πρέπει να κάνουμε την επιφόρτωση με μια συνάρτηση της μορφής:

```
Trv operator<( Tl lhs, Tr rhs )
```

όπου, στην περίπτωσή μας:

- *Tl* και *Tr* είναι ο `const Date&` και
- *Trv* είναι ο `bool`.

δηλαδή:

```
bool operator<( const Date& lhs, const Date& rhs )
```

Η υλοποίηση είναι απλή: ξεκινάμε από τα έτη. Αν δεν βγαίνει συμπέρασμα –αν δηλαδή είναι ίσα– πάμε στους μηνες και αν δεν βγαίνει και εκεί συμπέρασμα πάμε στις ημέρες:

```
bool operator<( const Date& lhs, const Date& rhs )
{
   bool fv;

   if ( lhs.year < rhs.year )           fv = true;
   else if ( lhs.year > rhs.year )      fv = false;
   else // lhs.year == rhs.year
   {
      if ( lhs.month < rhs.month )      fv = true;
      else if ( lhs.month > rhs.month ) fv = false;
      else // lhs.year == rhs.year && lhs.month == rhs.month
         fv = ( lhs.day < rhs.day );
   }
   return fv;
} // operator<( const Date . . .
```

Όπως καταλαβαίνεις, τώρα μπορούμε να ορίσουμε πολλούς δικούς μας τύπους και να επιφορτώσουμε πολλούς τελεστές σε αυτούς.

15.5.1 Συγκρίσεις και Κλειδιά

Είπαμε παραπάνω ότι η ισότητα ορίζεται όπως είδαμε «κατ' αρχήν». Τι θα πει αυτό; Αυτός ο ορισμός της ισότητας είναι ο –ας πούμε– «προγραμματιστικός». Αν πάρουμε όμως υπόψη μας και το νόημα των αντικειμένων τα πράγματα μπορεί να απλουστευθούν. Ας πούμε ότι σε μια εφαρμογή, όπου χρησιμοποιούμε αντικείμενα τύπου *Employee*, έχουμε δεχθεί ότι *δεν υπάρχει περίπτωση να έχουμε δύο εργαζόμενους με ίδιο όνομα, επώνυμο και ημερομηνία γέννησης*. Με βάση αυτό μπορούμε να ορίσουμε την ισότητα μεταβλητών τύπου *Employee* ως εξής:

```
bool operator==( const Employee& lhs, const Employee& rhs )
{
   return ( strcmp(lhs.surname, rhs.surname)==0 &&
            strcmp(lhs.firstname, rhs.firstname)==0 &&
            lhs.birthDate == rhs.birthDate );
```

```
} // operator==( const Employee
```

Πρόσεξε ότι εδώ συγκρίνουμε αντικείμενα τύπου *Date*. Θα πρέπει λοιπόν να έχουμε ορίσει:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day == rhs.day );
}; // operator==( const Date
```

Το κέρδος μας είναι ότι δεν χρειάζεται να επιφορτώσουμε τον "==" και για τον τύπο *Address*.

Λέμε ότι τα μέλη (χαρακτηριστικά) *surname, firstname, birthDate* αποτελούν **κλειδί** (key) για αντικείμενα τύπου *Employee*.[5]

Φυσικά, με βάση το κλειδί γίνονται και οι επιφορτώσεις των άλλων τελεστών σύγκρισης. Ας πούμε ότι χρειαζόμαστε και τον "<" για τον *Employee*.

```
bool operator<( const Employee& lhs, const Employee& rhs )
{
   bool fv;
   if ( strcmp(lhs.surname, rhs.surname) < 0 ) fv = true;
   else if ( strcmp(lhs.surname, rhs.surname) > 0 ) fv = false;
   else // strcmp(lhs.surname, rhs.surname)==0
   {
      if ( strcmp(lhs.firstname, rhs.firstname) < 0 )  fv = true;
      if ( strcmp(lhs.firstname, rhs.firstname) > 0 )  fv = false;
      else // strcmp(lhs.surname, rhs.surname)==0 &&
           // strcmp(lhs.firstname, rhs.firstname)==0
         fv = lhs.birthDate < rhs.birthDate;
   }
   return fv;
}; // operator<( const Employee
```

Όπως βλέπεις είναι πιο πολύπλοκος από τον "==" και για να υλοποιηθεί θα πρέπει θα πάρουμε μια απόφαση: με ποια σειρά θα συγκρίνουμε τα μέρη του κλειδιού. Εδώ είπαμε: πρώτα τα επώνυμα –αλφαβητικώς– μετά τα ονόματα –και αυτά αλφαβητικώς– και τέλος οι ημερομηνίες γέννησης. Για την τελευταία σύγκριση θα πρέπει να επιφορτώσουμε τον "<" για τον *Date*· αυτό σου το αφήνουμε ως άσκηση.

15.6 Αποθήκευση Μελών Δομής

Το πρότυπο της C++ λέει ότι

- ***Σε μια τιμή-δομή η διεύθυνση του κάθε μέλους είναι ανώτερη (μεγαλύτερη) από τη διεύθυνση του προηγούμενου (κατά τη σειρα δήλωσης).***

Τίποτε περισσότερο!

Δουλεύοντας με ορισμένους μεταγλωττιστές μπορεί να σχηματίσεις μια λαθεμένη εικόνα. Στην επόμενη υποπαράγραφο σου δείχνουμε πώς μπορείς να το ψάξεις.

15.6.1 * Σκαλίζοντας τη Μνήμη

Για να το επιβεβαιώσουμε δοκιμάζουμε το παρακάτω πρόγραμμα:

```
#include <iostream>
```

[5] Το συγκεκριμένο κλειδί του παραδείγματος δεν είναι ικανοποιητικό. Η ταυτοποίηση προσώπων είναι χαρακτηριστική περίπτωση όπου χρειαζόμαστε κλειδί με πολλά χαρακτηριστικά. Συνήθως, σε τέτοιες περιπτώσεις, ορίζουμε ένα πιο εύχρηστο **υποκατάστατο** (surrogate) κλειδί όπως είναι ο αριθμός ταυτότητας, ο ΑΦΜ, ο ΑΜΚΑ και διάφοροι άλλοι αριθμοί μητρώου.

```
using namespace std;

struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address

int main()
{
   Address a;

   cout << " sizeof a: " << (sizeof a) << endl;
   cout << " members: "
        << sizeof(a.country) + sizeof(a.city) +
           sizeof(a.areaCode) + sizeof(a.street) +
           sizeof(a.number) << endl;
   cout << " address of a: "
        << reinterpret_cast<long int>(&a) << endl;
   cout << " address of a.country: "
        << reinterpret_cast<long int>(a.country) << "  "
        << reinterpret_cast<long int>(&a) << endl;
   cout << " address of a.city: "
        << reinterpret_cast<long int>(a.city) << "  "
        << reinterpret_cast<long int>(a.country)+sizeof(a.country)
        << endl;
   cout << " address of a.areaCode: "
        << reinterpret_cast<long int>(&a.areaCode) << "  "
        << reinterpret_cast<long int>(a.city)+sizeof(a.city)
        << endl;
// . . .
```

Όπως βλέπεις, προσπαθούμε να συγκρίνουμε

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (`sizeof a`) με το άθροισμα των μεγεθών των μελών της (`sizeof(a.country) + sizeof(a.city) + sizeof(a.areaCode) + sizeof(a.street) + sizeof(a.number)`),
- Τη διεύθυνση του πρώτου μέλους (`a.country`) με τη διεύθυνση της μεταβλητής-δομής (`&a`).
- Τη διεύθυνση του κάθε επόμενου μέλους (π.χ. `a.city`) με αυτήν που προκύπτει αν στη διεύθυνση του προηγούμενου (`a.country`) προσθέσουμε το μέγεθός του (`sizeof(a.country)`).

Το `reinterpret_cast<long int>(&a)` σημαίνει: ερμήνευσε την εσωτερική παράσταση του βέλους `&a` ως τιμή τύπου `long int` (δες την επόμενη παράγραφο)· τελικώς, μας δίνει τη διεύθυνση σε δεκαδική μορφή αντί για δεκαεξαδική.

Αν περάσουμε το πρόγραμμά μας από τον Borland C++ 5.02 (για Win32) θα πάρουμε:

```
sizeof a: 59
members: 59
address of a: 1245008
address of a.country: 1245008  1245008
address of a.city: 1245025  1245025
address of a.areaCode: 1245042  1245042
address of a.street: 1245046  1245046
address of a.number: 1245063  1245063
```

Δηλαδή:

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (59) ισούται με το άθροισμα των μεγεθών των μελών της.

- Η αποθήκευση του πρώτου μέλους (1245008) αρχίζει εκεί που αρχίζει η αποθήκευση της *a*.
- Η αποθήκευση του κάθε επόμενου μέλους αρχίζει ακριβώς μετά το τέλος του προηγούμενου.

Αν όμως χρησιμοποιήσουμε τον Borland C++ 5.5 (για Win32) –που συμμορφώνεται πολύ περισσότερο με το πρότυπο της C++– θα πάρουμε:

```
sizeof a: 64
members: 59
address of a: 1245004
address of a.country: 1245004  1245004
address of a.city: 1245021  1245021
address of a.areaCode: 1245040  1245038
address of a.street: 1245044  1245044
address of a.number: 1245064  1245061
```

που είναι μια διαφορετική εικόνα. Παρόμοια θα πάρεις και από τον gcc (Dev-C++). Δηλαδή:

- Το μέγεθος ολόκληρης της μεταβλητής-δομής (64) *δεν* ισούται με το άθροισμα των μεγεθών των μελών της (59).
- Η αποθήκευση του μέλους `a.areaCode` *δεν* αρχίζει ακριβώς μετά το τέλος του προηγούμενου (1245038) αλλά μετά από δύο ψηφιολέξεις (1245040). Παρομοίως, η αποθήκευση του μέλους `a.number` *δεν* αρχίζει ακριβώς μετά το τέλος του προηγούμενου (1245061) αλλά μετά από τρεις ψηφιολέξεις.

Θα περιγράψουμε με απλά λόγια έναν λόγο που δεν ισχύουν γενικώς τα παραπάνω: τη λεγόμενη **πλήρωση** (padding). Ορισμένοι υπολογιστές, για να αυξήσουν την απόδοσή τους δεν ξεκινούν την αποθήκευση μιας μεταβλητής από οποιαδήποτε ψηφιολέξη αλλά *από την αρχή μιας λέξης* (word) που μπορεί να περιέχει, π.χ., 2 ή 4 ψηφιολέξεις. Αν λοιπόν έχεις μια:

```
struct cl1
{
   char c1;
   int klm;
// ...
};
```

σε έναν υπολογιστή που θέλει τα πάντα να ξεκινούν από την αρχή λέξης τεσσάρων ψηφιολέξεων, το *c*1 θα καταλάβει μεν μια ψηφιολέξη αλλά οι επόμενες τρεις θα παραμείνουν κενές.

Κάτι τέτοιο κάνουν στο παράδειγμά μας οι μεταγλωττιστές Borland C++ 5.5 και gcc: η αποθήκευση των δύο μελών τύπου `int` αρχίζει απο᾽διεύθυνση που είναι πολλαπλάσιο του 4.

15.7 Ερμηνευτική Τυποθεώρηση

Ένας νεόκοπος προγραμματιστής (που είχε ξεχάσει αυτά που λέγαμε στο Κεφ. 1 και θα παρουσιάσουμε εν εκτάσει στο επόμενο κεφάλαιο) είχε την εξής απορία: Στη C++ που χρησιμοποιούσε οι τιμές `long int` έπιαναν 4 ψηφιολέξεις, το ίδιο και οι τιμές `float`. Πόσο να μοιάζουν άραγε οι παραστάσεις των `7.345F` και `7L`, Αν δούμε τα δυαδικά ψηφία του `7.345F` ως τιμή τύπου `long int`, θα δούμε άραγε κάποιο εφτάρι (“`111`”); (!!!) Η πρώτη σκέψη ήταν να δοκιμάσει τα:

```
   float    ff( 7.345 );
   long int nn( static_cast<long>(ff) );
```

αλλά μέχρι εδώ ήξερε: Καμιά σχέση με αυτό που ήθελε, διότι με το `static_cast<long>(ff)`

- *κάνει μετατροπή σε άλλη εσωτερική παράσταση* (από `float` σε `long`) και φυσικά

- *αλλάζει την τιμή* (από 7.345 σε 7).

Μετά σκέφτηκε το εξής: να βάλει

```
long* pl( &ff );
cout << (*pl) << endl;
```

και να δει την τιμή του ***pl***. Αυτό είναι ακριβώς που θέλει να κάνει αλλά ο μεταγλωττιστής του είχε αντιρρήσεις:

```
"Cannot convert 'float*' to 'long*'"
```

Πώς μπορεί να γίνει; Έτσι:

```
long* pl( reinterpret_cast<long*>(&ff) );
cout << (*pl) << endl;
```

Αποτέλεσμα:

```
7  1089145405
```

Στην πρώτη περίπτωση βλέπουμε τα γνωστά και δεν έχουμε καμιά απορία για το πώς βγαίνει το «7» στη γραμμή αποτελεσμάτων. Να τονίσουμε ότι, όπως ξέρουμε και φαίνεται από το παράδειγμα,

- αλλάζει η *εσωτερική παράσταση* (από `long` σε `float`)
- αλλάζει η τιμή (από 7.345 σε 7).

Τι γίνεται στη δεύτερη περίπτωση; Στο βέλος *pl*, τύπου `long*`, δίνουμε ως τιμή την τιμή του βέλους `&ff`, που είναι τύπου `float*`. Και γιατί το γράφουμε έτσι; Διότι, όπως είδες, η C++, ελέγχοντας τις συμβατότητες τύπων, δεν θα μας επιτρέψει να γράψουμε `long* ll( &ff )`[6]. Εδώ

- αλλάζει μεν ο τύπος αλλά δεν αλλάζει η τιμή: το βέλος δείχνει την ίδια θέση της μνήμης,
- δεν αλλάζει η εσωτερική παράσταση (τα ίδια δυαδικά ψηφία),
  - ούτε για το βέλος,
  - ούτε για τον στόχο.

Έτσι, στο *pl* αποθηκεύεται η διεύθυνση της *ff* και έχουμε ένα βέλος τύπου `long*` να δείχνει μια μεταβλητή τύπου `float`. Αυτό που παίρνουμε από αποπαραπομπή του *pl* και βλέπεις ως αποτέλεσμα (1089145405) είναι αυτό που βγαίνει αν «διαβάσουμε» τα δυαδικά ψηφία της *ff* σαν να παριστάνουν μια τιμή τύπου `long`.

Γενικώς αν έχουμε:

```
reinterpret_cast< T >( Π )
```

Ο τύπος *T* πρέπει να είναι ακέραιος, βέλους, αναφοράς. Τα ίδια ισχύουν και για τον τύπο του αποτελέσματος της παράστασης *Π*. Όπως θα κατάλαβες, η **ερμηνευτική τυποθεώρηση** (reinterpretation casting), μας επιτρέπει να ερμηνεύσουμε τα δυαδικά ψηφία κάποιας θέσης της μνήμης με τους κανόνες κάποιου άλλου τύπου. Για τον ίδιο σκοπό μπορούμε να χρησιμοποιήσουμε και τη `union`, όπως θα δούμε στη συνέχεια.

Ας δούμε άλλο ένα παράδειγμα: Κάποιος, που μαθαίνει προγραμματισμό με C++, μαθαίνει ένα «μυστικό»: Αν, στη C++ που δουλεύει, μια τιμή *k*, τύπου `unsigned short int`, αποθηκεύεται σε δύο ψηφιολέξεις, τότε στη μια από αυτές υπάρχει το *k*`/`256 και στην άλλη το *k*`%`256. Αν, ας πούμε, έχουμε:

```
unsigned short int k( 8548 );
```

Η εντολή:

```
cout << (k%256) << "  " << (k/256) << endl;
```

δίνει:

```
100  33
```

[6] Και γιατί θα θέλαμε να κάνουμε κάτι τέτοιο; Θα δεις στη συνέχεια ότι αυτό είναι απαραίτητο σε πολλές περιπτώσεις.

Πώς μπορούμε να διαπιστώσουμε ότι αυτές οι τιμές υπάρχουν στις δύο ψηφιολέξεις; Θυμήσου αυτά που διάβασες την προηγούμενη παράγραφο· δηλώνουμε:

```
char* p;
```

Η *p* είναι μια μεταβλητή-βέλος που δείχνει προς μια μεταβλητή τύπου `char`. Και τώρα δίνουμε:

```
p = reinterpret_cast<char*>(&k);
```

Ποιο είναι το νόημά της; Πάρε το βέλος προς την *k* (τη διεύθυνση της *k*) και αφού το δεις ως βέλος προς μια θέση τύπου `char`, να το κάνεις τιμή της *p*.

Έτσι, έχουμε μια μεταβλητή-βέλος προς μια θέση τύπου `char`, που όμως δείχνει την *k*, μια μεταβλητή τύπου `int`. Όπως έχουμε πει στην §12.1, για τη C++ πίνακας και βέλος είναι το ίδιο πράγμα. Ε, τότε ας δοκιμάσουμε την:

```
cout << p[0] << "  " << p[1] << endl;
```

Λοιπόν: γίνεται δεκτή χωρίς πρόβλημα και δίνει:

```
d  !
```

ενώ αν δώσουμε:

```
cout << static_cast<int>(p[0]) << "  "
     << static_cast<int>(p[1]) << endl;
```

θα πάρουμε αποτέλεσμα:

```
100  33
```

Είδαμε (§15.7) πώς ερμηνεύουμε –με ερμηνευτική τυποθεώρηση– την τιμή βέλους (μια διεύθυνση) ως ακέραιη τιμή. Είπαμε όμως παραπάνω ότι, με τον ίδιο τρόπο μπορούμε να δούμε και έναν ακέραιο ως βέλος. Για παράδειγμα, ας πούμε ότι δηλώνουμε:

```
float a[10];
int   aToInt( reinterpret_cast<int>(a) );
```

Στην *aToInt* δίνουμε ως τιμή τη διεύθυνση που αρχίζει η αποθήκευση του πίνακα a. Η παράσταση:

```
reinterpret_cast<float*>( aToInt + k*sizeof(float) )
```

είναι ένας άλλος τρόπος να γράψεις το "`a+k`" ή "`&a[k]`". Φυσικά, *είναι σαφώς χειρότερος από τους άλλους δύο*.

Κατά το πρότυπο της C++, για την ερμηνευτική τυποθεώρηση το μόνο σίγουρο είναι το εξής: Αν μετατρέψεις ένα βέλος σε ακέραιο –με αρκετά μεγάλο μέγεθος[7]– και μετά τον ακέραιο στον ίδιο τύπο βέλους θα πάρεις την αρχική τιμή. Δηλαδή αν *IT* ακέραιος τύπος με ικανοποιητικό μέγεθος και

```
T* p;
```

τότε:

```
reinterpret_cast<T*>(reinterpret_cast<IT>(p)) ≡ p
```

Κατά τα άλλα:

- Η αντιστοίχιση βελών και ακεραίων εξαρτάται από την υλοποίηση.
- Για να την καταλάβεις θα πρέπει να ξέρεις τη δομή *διευθυνσιοδότησης* (addressing) του υπολογιστή σου.

Χρησιμοποιήσαμε και θα ξαναχρησιμοποιήσουμε την ερμηνευτική τυποθεώρηση για να «σκαλίσουμε» τη μνήμη του υπολογιστή και να καταλάβουμε πώς δουλεύουν ορισμένα πράγματα. Θα τη χρησιμοποιούμε σε «πραγματικό» πρόγραμμα; Μόνο σε μια περίπτωση: τη διαχείριση μη μορφοποιημένων αρχείων που θα δούμε παρακάτω. Γενικώς, θεωρείται ως επικίνδυνο εργαλείο.

---

[7] Για παράδειγμα: Στο περιβάλλον Windows (Win32) οι διευθύνσεις περιγράφονται με 32 δυαδικά ψηφία. Στη BC++ 5.5 και στη Dev-C++ σε 32 δυαδικά ψηφία αποθηκεύονται οι τιμές των τύπων `int` και `long int`.

15.8 * Ψηφιοπεδία

Ας ξεκινήσουμε με ένα παράδειγμα –ελαφρώς τροποποιημένο– από το (Kernighan & Ritchie 1988):

```
struct flags1
{
   bool isKeyword;
   bool isExtern;
   bool isStatic;
}; // flags1
```

Με την

```
   cout << "sizeof flags1: " << (sizeof(flags1)) << endl;
```

παίρνουμε:

```
sizeof flags1: 3
```

(ψηφιολέξεις) ενώ ξέρουμε ότι χρησιμοποιούμε 3 δυαδικά ψηφία. Μπορούμε να κάνουμε οικονομία; Ναι, έτσι:

```
struct flags2
{
   bool isKeyword: 1;
   bool isExtern:  1;
   bool isStatic:  1;
}; // flags2
```

Μια μεταβλητή τύπου *flags*2 αποθηκεύεται σε 1 ψηφιολέξη.

Τα μέλη *isKeyword*, *isExtern*, *isStatic* ονομάζονται **πεδία** (fields) ή –καλύτερα– **ψηφιοπεδία** (bit-fields). Γενικώς, στη δήλωση μιας δομής μπορείς να δηλώσεις ένα ψηφιοπεδίο ως εξής:

τύπος, *όνομα*, "`:`", *φυσικός*

Ο *τύπος* μπορεί να είναι: `int`, `unsigned int`, `char`, `unsigned char`, `wchar_t`, `bool`. Ο *φυσικός* δίνει το πλήθος των δυαδικών ψηφίων που θα χρησιμοποιηθούν για την παράσταση του ψηφιοπεδίου. Όπως καταλαβαίνεις, τα `int` και `char` δεν παίζουν και τόσο σημαντικό ρόλο. Το `unsigned` όμως είναι ουσιαστικό διότι, αν δεν υπάρχει, ένα δυαδικό ψηφίο θα χρησιμοποιηθεί για πρόσημο. Το όνομα μπορεί και να λείπει, οπότε το ψηφιοπεδίο δεν είναι προσβάσιμο (ούτε απ' ευθείας διαχειρίσιμο).

Η διαχείριση των ψηφιοπεδίων γίνεται με τους γνωστούς τελεστές επιλογής. Αν έχουμε δηλώσει:

```
   flags2  aSymbol;
   flags2* pToSymbol;
```

μπορούμε να χρησιμοποιούμε, με το νόημα που ξέρουμε, τα:

```
   aSymbol.isKeyword    aSymbol.isExtern
   pToSymbol->isExtern    pToSymbol->isStatic
```

Να δούμε άλλο ένα

Παράδειγμα

Έστω ότι σχεδιάζουμε έναν τύπο εγγραφής για να κρατούμε στοιχεία μιας χρονικής στιγμής:

```
struct time1
{
   unsigned short year;
   unsigned char  month;
   unsigned char  day;
   unsigned char  hour;
   unsigned char  min;
   unsigned char  sec;
}; // time1
```

Για όλα τα μέλη εκτός από το πρώτο, βάλαμε τύπο `unsigned char` –αν και θα τα χειριστούμε ως ακεραίους– με στόχο να παίρνουμε μία ψηφιολέξη μόνον για αποθήκευση. Μια μεταβλητή αυτού του τύπου καταλαμβάνει 8 ψηφιολέξεις (gcc - Dev-C++, BC++ v.5.5). Κάτι τέτοιο φαίνεται σπάταλο διότι:

- Για έτος μέχρι 4095 χρειαζόμαστε 12 δυαδικά ψηφία,
- για μήνα (1-12) όχι περισσότερα από 4 δυαδικά ψηφία (0 - 15),
- για ημέρα (1-31) όχι περισσότερα από 5 δυαδικά ψηφία (0 - 31),
- για ώρα (0 - 23) όχι περισσότερα από 5 δυαδικά ψηφία (0 - 31),
- για πρώτα λεπτά (0 - 59) όχι περισσότερα από 6 δυαδικά ψηφία (0 - 63)
- για δεύτερα λεπτά (0 - 59) όχι περισσότερα από 6 δυαδικά ψηφία (0 - 63).

Σύνολο: 38 δυαδικά ψηφία που μας τα δίνουν 5 ψηφιολέξεις και όχι 8 (που έχει 60% σπατάλη).

Η C++ σου επιτρέπει να περιορίσεις τη σπατάλη αν ορίσεις:

```
struct time2
{
   unsigned int year:  12;
   unsigned char month: 4;
   unsigned char day:   5;
   unsigned char hour:  5;
   unsigned char min:   6;
   unsigned char sec:   6;
}; // time2
```

που καταλαμβάνει 6 ψηφιολέξεις (gcc - Dev-C++, BC++ v.5.5).

Χρησιμοποιώντας δομές με ψηφιοπεδία μπορείς να κάνεις οικονομία στον χώρο που κατάλαμβάνουν τα στοιχεία σου, στη κύρια (και στη βοηθητική) μνήμη. Αλλά ποιο είναι το τίμημα; Το (εκτελέσιμο) πρόγραμμά σου γίνεται μεγαλύτερο και πιο αργό.

Μπορείς να χρησιμοποιείς ψηφιοπεδία μόνον μέσα σε **struct** ή σε **union** (τη βλέπεις στην επόμενη παράγραφο) ή σε **class** (θα τη δούμε αργότερα).

15.9 * union

Είδαμε ότι η ερμηνευτική τυποθεώρηση μας επιτρέπει να ερμηνεύσουμε το περιεχόμενο της μνήμης με διάφορους τρόπους. Έτσι είδαμε δύο διαδοχικές ψηφιολέξεις και ως τιμή τύπου `unsigned short int` και ως πίνακα τύπου `char` με δύο στοιχεία. Θα δούμε τώρα έναν άλλον τρόπο για να κάνουμε το ίδιο πράγμα.

Αυτό μπορεί να γίνει με ένα άλλο είδος δομής, αυτό της `union`. Ας το δούμε με ένα παράδειγμα. Ας πούμε ότι δηλώνουμε:

```
union U
{
   char c;
   int  i;
}; // U

   U  m;
```

και δίνουμε:

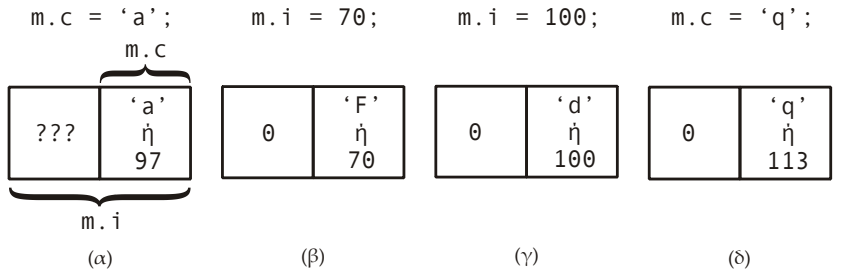
```
   m.c = 'a';  m.i = 70;
   cout << m.c << endl;
```

Αποτέλεσμα:

```
F
```

Δίνουμε:

```
   m.i = 180;  m.c = 'q';
```



Σχ. 15-1 Σχηματική παράσταση της `union { char c; int i; } m`. Το μέλος *m.i*, που είναι τύπου `int`, καταλαμβάνει δύο ψηφιολέξεις (α). Η μια από αυτές είναι η ψηφιολέξη που αποθηκεύεται το μέλος *m.c*. Αυτή η ψηφιολέξη είναι κοινή για τα δύο μέλη. Στο (α) βλέπεις την κατάσταση μετά την εκτέλεση της εντολής `m.c = 'a'`. Στην κοινή ψηφιολέξη αποθηκεύεται το `'a'`, που αν το δούμε ως ακέραιο έχει την τιμή 97. Στο (β) βλέπεις την κατάσταση μετά την εκτέλεση της εντολής `m.i = 70`. Η τιμή 70 αποθηκεύεται ως εξής: 70 στη μία ψηφιολέξη (την κοινή) και 0 στην άλλη. Αν ζητήσουμε να δούμε την τιμή του *m.c*, αυτή θα προκύψει αν δούμε ως χαρακτήρα το περιεχόμενο της κοινής ψηφιολέξης, δηλαδή `'F'`. Στα (γ) και (δ) βλέπεις την κατάσταση μετά τις εντολές `m.i = 180` και `m.c = 'q'` αντιστοίχως.

```
   cout << m.i << endl;
```

Αποτέλεσμα:

```
113
```

Δηλαδή δίνουμε στο *m.c* τιμή `'a'` και όταν την τυπώνουμε βγαίνει "`F`". Δίνουμε στο *m.i* τιμή 180 και μας βγαίνει "`113`". Τι γίνεται;

Σε μια **union** όλα τα μέλη αποθηκεύονται στην ίδια θέση της μνήμης. Ακριβέστερα, η αποθήκευση αρχίζει από την ίδια ψηφιολέξη, διότι τα μέλη μπορεί να έχουν διαφορετικά μεγέθη. Εδώ λοιπόν τι γίνεται; Έστω ότι μια θέση τύπου `int` πιάνει 2 ψηφιολέξεις και μια θέση τύπου `char` 1 ψηφιολέξη. Η αποθήκευση της μεταβλητής *m* και των μελών της φαίνεται στο Σχ. 15-1. Όπως καταλαβαίνεις, αλλάζοντας την τιμή του ενός μέλους αλλάζει ταυτοχρόνως και η τιμή του άλλου.

Θα μπορούσαμε λοιπόν να κάνουμε αυτό που είδαμε στην §15.7 ως εξής: Ορίζουμε μια

```
union U1
{
   char               c[2];
   unsigned short int i;
}; // U1
```

δηλώνουμε:

```
   U1 u1;
```

και οι

```
   u1.i = 8548;
   cout << u1.c[0] << "  " << u1.c[1] << endl;
   cout << static_cast<int>(u1.c[0]) << "  "
        << static_cast<int>(u1.c[1]) << endl;
```

θα δώσουν:

```
d  !
100  33
```

Πάντως η **union** μας δίνει και άλλες δυνατότητες. Ας πούμε, για παράδειγμα, ότι έχουμε την παράσταση:

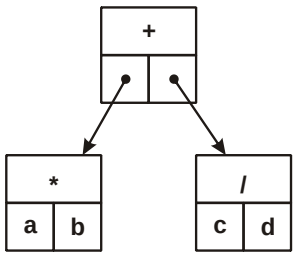
```
a * b + c / d
```

Συνήθως την παριστάνουμε όπως βλέπεις στο Σχ. 15-2. Αν θέλουμε να παραστήσουμε στο πρόγραμμά μας τον κόμβο του δένδρου έχουμε το εξής πρόβλημα: Έστω ότι ορίζουμε την:

```
   struct Node
```

```
   {
      char oper;
      Node* arg1;
      Node* arg2;
   }; // Node
```

με μέλη έναν χαρακτήρα, που σημειώνει την πράξη και δύο βέλη προς τους κόμβους των ορισμάτων. Αυτός ο τύπος είναι μια χαρά για τον κόμβο με την πρόσθεση. Αλλά δεν μας λύνει το πρόβλημα για τους άλλους δύο τύπους που έχουν αριθμητικά ορίσματα. Εκεί χρειαζόμαστε εγγραφές τύπου:



Σχ. 15-2 Δενδρική παράσταση απλής αριθμητικής παράστασης.

```
   struct Node
   {
      char oper;
      double arg1;
      double arg2;
   }; // Node
```

Το πρόβλημά μας μπορεί να λυθεί ως εξής· ορίζουμε:

```
struct Node;

union Arg
{
   double d;
   Node*  p;
}; // Arg

struct Node
{
   char oper;
   char leftArgKind;  Arg leftArg;
   char rightArgKind;  Arg rightArg;
}; // Node
```

Η αρχική `struct` *Node* είναι μια προαναγγελία δήλωσης, απαραίτητη για να γίνει δεκτός ο ορισμός της *Arg*, που έχει μέσα τη δήλωση `Node* p`.

Στην *Node*, εκτός από τα *oper*, *leftArg*, *rightArg*, υπάρχουν και τα μέλη *leftArgKind*, *rightArgKind*. Στην πρώτη αποθηκεύουμε μια τιμή που μας δείχνει ποιο από τα μέλη της *leftArg* είναι σε χρήση. Παρόμοιο ρόλο παίζει η *rightArgKind* σε σχέση με τη rightArg.

Ας πούμε ότι έχουμε δηλώσει:

```
   Node root, mulT, divT;
```

Στα μέλη της *pr* δίνουμε τις εξής τιμές:

```
   root.oper = '+';
   root.leftArgKind = 'p';  root.leftArg.p = &mulT;
   root.rightArgKind = 'p';  root.rightArg.p = &divT;
```

Όπως βλέπεις, στην *root*.*leftArgKind* δίνουμε τιμή στο μέλος (βέλος) *p* (τη διεύθυνση του κόμβου *mulT*). Στην *root*.*leftArgKind* δίνουμε τιμή `'p'` που σημαίνει ότι στη *root*.*leftArgKind* έχουμε σε χρήση το μέλος *p*.

Στα μέλη των *mulT* και *divT* δίνουμε:

```
   mulT.oper = '*';
   mulT.leftArgKind = 'd';  mulT.leftArg.d = a;
   mulT.rightArgKind = 'd';  mulT.rightArg.d = b;
   divT.oper = '/';
   divT.leftArgKind = 'd';  divT.leftArg.d = c;
   divT.rightArgKind = 'd';  divT.rightArg.d = d;
```

Τώρα στη *mulT*.*leftArg* δίνουμε τιμή στο μέλος *d*, την τιμή της μεταβλητής *a*. Στη *mulT*.*leftArgKind* δίνουμε τιμή `'d'` που σημαίνει ότι στη *mulT*.*leftArg* έχουμε σε χρήση το μέλος *d*.

Ο υπολογισμός της παράστασης γίνεται με την κλήση:

```
   . . . eval( &root ) . . .
```

προς την απλή αναδρομική συνάρτηση:

```
double eval( Node* r )
{
   double argL, argR, fv;

   argL = (r->leftArgKind == 'd') ?
             r->leftArg.d : eval(r->leftArg.p);
   argR = (r->rightArgKind == 'd') ?
             r->rightArg.d : eval(r->rightArg.p);
   switch ( r->oper )
   {
      case '+':  fv = argL + argR;  break;
      case '-':  fv = argL - argR;  break;
      case '*':  fv = argL * argR;  break;
      case '/':  fv = argL / argR;  break;
//    default: throw . . .
   }
   return fv;
} // eval
```

15.10 Δομές για Εξαιρέσεις

Όταν γράφεις κάποιο πρόγραμμα –πάνω από δυο γραμμές– έχεις μια σιγουριά: θα έχει λάθη! Τα βρίσκεις βέβαια και τα διορθώνεις αλλά, όπως λέει και ο πιο γνωστός νόμος του Murphy για τον προγραμματισμό: *Υπάρχει πάντοτε ακόμη ένα λάθος*.[8] Το πρόγραμμα θα πρέπει να είναι προετοιμασμένο ώστε, «όταν χτυπήσει ο κεραυνός», να μας αναφέρει για κάθε λάθος:

- Πού έγινε το λάθος.
- Τι είδους λάθος ήταν.
- Πώς προκλήθηκε.

Αυτές τις πληροφορίες θα πρέπει να μεταφέρουν οι εξαιρέσεις που ρίχνουμε. Θα ξαναγράψουμε τα παραδείγματα που δώσαμε στην §14.9 ορίζοντας όμως έναν τύπο εξαιρέσεων που μας επιτρέπει να μεταβιβάσουμε αυτές τις πληροφορίες:

```
struct ApplicXptn
{
   enum { domainErr, paramErr };
   char   funcName[100];
   int    errCode;
   double errDblVal1, errDblVal2, errDblVal3;

   ApplicXptn( const char* fn, int ec,
               double dv1, double dv2 = 0, double dv3 = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errDblVal1 = dv1;  errDblVal2 = dv2;  errDblVal3 = dv3;  }
}; // ApplicXptn
```

Αυτή είναι μια **δομή** (ή **κλάση**) **εξαιρέσεων**. Πρόσεξε τα μέλη της:

- `char funcName[100]`: εδώ θα βάζουμε το *όνομα της συνάρτησης* που ρίχνει την εξαίρεση.
- `int errCode`: εδώ θα βάζουμε έναν κωδικό που θα δείχνει το *είδος του λάθους* που παρουσιάστηκε. Οι κωδικοί λάθους απαριθμούνται στην αρχή. Στην περίπτωσή μας έχουμε δύο (*domainErr* = 0, *paramError* = 1). Χρησιμοποιούνται και στην έγερση και στη σύλληψη.
- `double errDblVal1, errDblVal2, errDblVal3`: εδώ θα βάζουμε *τις τιμές ή την τιμή* που προκάλεσαν το πρόβλημα.

[8] «There is always one more bug.»

Τέλος βλέπουμε:

- Έναν δημιουργό. Αυτός χρησιμοποιείται για την έγερση των εξαιρέσεων: δημιουργεί το αντικείμενο της εξαίρεσης. Μπορεί να χρειαστεί να γράψουμε και άλλους δημιουργούς.

Δες λοιπόν πώς μπορούμε να γράψουμε τις δύο συναρτήσεις:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
      throw ApplicXptn( "v", ApplicXptn::domainErr, x );
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Όπως βλέπεις, στη `throw` καλούμε τον δημιουργό της δομής για να δημιουργήσει το αντικείμενο της εξαίρεσης. Τι του δίνουμε:

- `"v"`: το όνομα της συνάρτησης.
- `ApplicXptn::domainErr`: τον κωδικό σφάλματος. Περίεργος συμβολισμός; Τον έχουμε ξαναδεί· στην §1.2 είχαμε δει τα `std::cout`, `std::endl`. Εδώ εννοούμε: το `domainErr` που ορίσαμε στον ονοματοχώρο `ApplicXptn`.[9]
- `x`: την τιμή που προκάλεσε το πρόβλημα.

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
      throw ApplicXptn( "pqer", ApplicXptn::paramErr, x, y, z );
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Πρόσεξε ότι εδώ περνούμε τρεις τιμές (x, y, z) στη συνάρτηση.

Τώρα έχουμε μια `catch` εκτός από την "`catch(...)`":

```
int main()
{
   double t, u;

   try
   {
   // . . .
      pqer( 1, 2, 3, t, u );
   // . . .
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   // . . .
   }
   catch ( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::domainErr:
            cout << " η " << x.funcName << " δεν ορίζεται στο "
                 << x.errDblVal1 << endl;
            break;
         case ApplicXptn::paramErr:
            cout << " η " << x.funcName
                 << " κλήθηκε με παραμέτρους " << x.errDblVal1
```

[9] Όπως θα μάθουμε αργότερα, κάθε δομή (κλάση) ορίζει τον δικό της ονοματοχώρο.

```
                << ", " << x.errDblVal2 << ", " << x.errDblVal2
                << endl;
            break;
         default:
            cout << " μη αναμενόμενη εξαίρεση ApplicXptn από "
                << x.funcName << endl;
      } // switch
   }
   catch( ... )
   {
      cout << " μη αναμενόμενη άγνωστη εξαίρεση" << endl;
   }
} // main
```

Ως δεύτερο παράδειγμα ας ξαναγράψουμε το πρόγραμμα με τους συνδυασμούς που είδαμε παραπάνω:

```
#include <iostream>

using namespace std;

struct ApplicXptn
{
   char funcName[100];
   int  errCode;
   int  iErrVal1;
   int  iErrVal2;

   enum { paramError, factOfNeg };
   ApplicXptn ( const char* fn, int ec, int iv1=0, int iv2=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      iErrVal1 = iv1;  iErrVal2 = iv2;  }
}; // ApplicXptn

   unsigned long int factorial( int n );
   unsigned long int comb( int m, int n );

int main()
{
   int m, n;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   try
   {
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb( m, n ) << endl;
   }
   catch ( ApplicXptn x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::paramError:
            cout << "Η " << x.funcName
                 << " κλήθηκε με λάθος παραμέτρους ( "
                 << x.iErrVal1 << " < " << x.iErrVal2 << " )"
                 << endl;
            break;
         case ApplicXptn::factOfNeg:
            cout << "Η " << x.funcName
                 << " κλήθηκε με αρνητικό όρισμα ( "
                 << x.iErrVal1 << " )" << endl;
            break;
         default:
            cout << "Μη αναμενόμενη εξαίρεση από την "
                 << x.funcName << " τιμές: " << x.iErrVal1 << ", "
                 << x.iErrVal2 << endl;
```

```
      } // switch
   } // catch
} // main

   // natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
   unsigned long int natProduct( int m, int n )
   {
      if ( m <= 0 || n < m )
         throw ApplicXptn( "natProduct",
                           ApplicXptn::paramError, m, n );
   // 0 < m <= n
      unsigned long int fv(m);
      for ( int k(m+1); k <= n; ++k )  fv *= k;
      return fv;
   } // natProduct

   // factorial -- Υπολογίζει το n!
   unsigned long int factorial( int n )
   {
      try
      {
         return ( n == 0 ) ? 1 : natProduct( 1, n );
      }
      catch( ApplicXptn x )
      {
         throw ApplicXptn( "factorial",
                           ApplicXptn::factOfNeg, n );
      }
   } // factorial

   // comb -- Υπολογίζει τους συνδυασμούς των m ανά n
   unsigned long int comb( int m, int n )
   {
      unsigned long int fv;

      if ( n < m-n )  fv = natProduct(m-n+1,m)/factorial(n);
                else  fv = natProduct(n+1,m)/factorial(m-n);
      return fv;
   } // comb
```

Εδώ πρόσεξε τα εξής:

1. Η δομή εξαιρέσεων κρατάει δύο «προβληματικές τιμές» διότι για ένα πρόβλημα που παρουσιάζεται στην *natProduct()* θα πρέπει να δώσουμε τις τιμές των *m* και *n*.
2. Αν προσπαθήσουμε να υπολογίσουμε το παραγοντικό ενός αρνητικού αριθμού, το να πιάσουμε μια εξαίρεση που θα έλθει από την *natProduct()* είναι μάλλον παραπλανητικό, Γι' αυτό στη *factorial()* πιάνουμε την εξαίρεση που έρχεται από την *natProduct()* και ρίχνουμε άλλη εξαίρεση με ακριβέστερη πληροφορία.

Τέτοιους τύπους εξαιρέσεων ορίζουμε για το πρόγραμμα, για μια βιβλιοθήκη συναρτήσεων κλπ. Έτσι, σε ένα πρόγραμμα μπορεί να ρίχνονται εξαιρέσεις διαφόρων τύπων και φυσικά θα πρέπει να έχουμε και τις κατάλληλες `catch`.

Αργότερα θα δούμε τις κλάσεις εξαιρέσεων πιο εκτεταμένα.

15.11 Μη Μορφοποιημένα Αρχεία

Μέχρι τώρα διαβάζουμε και γράφουμε μορφοποιημένα αρχεία, δηλαδή αρχεία με γραμμές, με (αναγνώσιμους) χαρακτήρες. Έχουμε πάντως τη δυνατότητα να γράφουμε σε αρχεία τα στοιχεία όπως είναι μέσα στη μνήμη του υπολογιστή, στην εσωτερική παράσταση.

Ξαναγράφουμε το πρώτο πρόγραμμα της §8.6 με κάποιες αλλαγές:

- Στο γράψιμο των τιμών στο αρχείο:

```
   double sum( 0 );
```

```
int    n( 0 );
ofstream a( "fltnum.dta", ios_base::binary );
cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
while ( x != sentinel )
{
// γράψε στο αρχείο την τιμή που πήρες
   a.write( reinterpret_cast<const char*>(&x), sizeof(x) );
   ++n;
   sum += x;
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
} // while
a.close();
cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
```

- Στο διάβασμα των τιμών από το αρχείο:

```
if ( n > 0 )
{
   double avrg( sum/n );
   cout << " ΑΘΡΟΙΣΜΑ = " << sum
        << "  <x> = " << avrg << endl;
   ifstream b( "fltnum.dta", ios_base::binary );
   int m( 0 );
   b.read( reinterpret_cast<char*>(&x), sizeof(x) );
   while ( !b.eof() )
   {
      ++m;
      y = exp( (avrg-x)/avrg );
      cout << " x[";
      cout.width(3);  cout << m << "] = ";
      cout.width(6);  cout << x << "   y[";
      cout.width(3);  cout << m << "] = " ;
      cout.width(6);  cout << y << endl;
      b.read( reinterpret_cast<char*>(&x), sizeof(x) );
   } // while
   b.close();
}
```

Τώρα, το βοηθητικό αρχείο **fltnum.dta** είναι **μη μορφοποιημένο** ή **δυαδικό** (binary, unformatted). Όταν δημιουργείς ή ετοιμάζεσαι να διαβάσεις ένα μη μορφοποιημένο αρχείο, πρέπει να βάλεις μια ακόμη παράμετρο. Ανοίγουμε δηλώνοντας το ρεύμα *a* με την:

```
ofstream a( "fltnum.dta", ios_base::binary );
```

και δείχνουμε ότι το αρχείο **fltnum.dta**, που θα δημιουργηθεί, θα είναι μη μορφοποιημένο. Θα μπορούσαμε, αν θέλαμε, να δηλώσουμε:

```
ofstream a;
```

και να ανοίξουμε με την:

```
a.open( "fltnum.dta", ios_base::binary );
```

Παρομοίως, ανοίγουμε δηλώνοντας το *b* με την:

```
   ifstream b( "fltnum.dta", ios_base::binary );
```

και δείχνουμε ότι το αρχείο **fltnum.dta** που θα διαβάσουμε είναι μη μορφοποιημένο. Και εδώ, θα μπορούσαμε, εναλλακτικώς, να δηλώσουμε:

```
ifstream b;
```

και να ανοίξουμε με τη:

```
   b.open( "fltnum.dta", ios_base::binary );
```

Αν μετατρέπαμε το δεύτερο πρόγραμμα, όπου χρησιμοποιούμε ρεύμα

```
fstream a;
```

θα ανοίγαμε το ρεύμα *a* με την:

```
a.open( "fltnum.txt",
        ios_base::in|ios_base::out|ios_base::binary );
```

Πώς γράφουμε σε ένα μη μορφοποιημένο αρχείο;

Η κλάση *ofstream* έχει τη μέθοδο *write()*, που περιμένει δύο ορίσματα:

- Το πρώτο είναι ένα βέλος, *p*, προς μεταβλητή τύπου `char` (ακριβέστερα: `const char*`).
- Το δεύτερο είναι ένας φυσικός *n*.

Αν δώσουμε `a.write( p, n )` εννοούμε:

- Στείλε στο ρεύμα *a*, *n* ψηφιολέξεις,
- ξεκινώντας από τη θέση μνήμης που δείχνει το *p*.

Στο παράδειγμά μας, έχουμε να γράψουμε στο αρχείο την τιμή της *x*. Θα μπορούσαμε να γράψουμε:

```
a.write( &x, sizeof(x) );
```

Όχι ακριβώς έτσι. Πράγματι, το βέλος `&x` δείχνει τη θέση μνήμης που αρχίζει η αποθήκευση των στοιχείων που θέλουμε να γράψουμε και η `sizeof(x)` μας δίνει τον αριθμό των ψηφιολέξεων. Το πρόβλημα είναι με τον τύπο του `&x`: είναι τύπου `double*` (βέλος προς μεταβλητή τύπου `double`) ενώ η *write()* περιμένει τιμή τύπου `const char*`. Εδώ χρειαζόμαστε την *ερμηνευτική τυποθεώρηση* που είδαμε πιο πριν. Γράφουμε λοιπόν:

```
a.write( reinterpret_cast<const char*>(&x), sizeof(x) );
```

Για διάβασμα, η κλάση *ifstream* έχει τη μέθοδο *read()*, που περιμένει δύο ορίσματα, σαν αυτά της *write()*. Αν δώσουμε `b.read(p, n)` εννοούμε: Πάρε από το ρεύμα *b*, *n* ψηφιολέξεις και αποθήκευσέ τις στη μνήμη, ξεκινώντας από τη θέση μνήμης που δείχνει το *p*. Εδώ, η *p* πρέπει να είναι τύπου `char*`. Έτσι, βλέπεις να διαβάζουμε με την:

```
b.read( reinterpret_cast<char*>(&x), sizeof(x) );
```

Η κλάση *fstream* έχει και τις δύο μεθόδους *write()* και *read()*.

Τα μη μορφοποιημένα αρχεία έχουν το πλεονέκτημα, σε σχέση με τα μορφοποιημένα, ότι

- επιτρέπουν μεγαλύτερη ταχύτητα στην επεξεργασία τους, διότι δεν καταναλίσκεται υπολογιστικός χρόνος για μετατροπές από την εσωτερική παράσταση σε χαρακτήρες (**μορφοποίηση**, formatting) –όταν γράφουμε– και από χαρακτήρες σε εσωτερική παράσταση –όταν διαβάζουμε.

Έχουν όμως και μειονεκτήματα:

- επειδή είναι γραμμένα σε εσωτερική παράσταση, πολλές φορές δεν είναι εύκολο να τα διαχειριστείς με προγράμματα που έχουν αναπτυχθεί σε διαφορετικά προγραμματιστικά περιβάλλοντα,
- δεν μπορείς να τα διαχειριστείς με κειμενογράφους.

Παρατήρηση: ▶
Να πούμε δυο λόγια παραπάνω για το τελευταίο «μειονέκτημα»: Το να διαχειρίζεσαι ένα αρχείο με κειμενογράφο σημαίνει και ότι μπορεί να εισαγάγεις σφάλματα. Το να διαχειρίζεσαι ένα αρχείο με ειδικό πρόγραμμα –όπως πρέπει να γίνει για ένα μη μορφοποιημένο αρχείο– σημαίνει ότι οι εισαγόμενες τιμές θα περνούν και κάποιους ελέγχους. Γι' αυτό όταν διαβάζουμε δεδομένα από μορφοποιημένο αρχείο κάνουμε όλους τους δυνατούς ελέγχους (σαν να διαβάζουμε από το πληκτρολόγιο). Αυτό δεν σημαίνει ότι δεν κάνουμε ελέγχους σε δεδομένα που διαβάζουμε από μη μορφοποιημένο αρχείο. ◀

Αν το σκεφτείς λιγάκι, το παράδειγμα που δώσαμε είναι από τα λίγα που σίγουρα συμφέρει να χρησιμοποιήσουμε μη μορφοποιημένο αρχείο. Σε άλλες περιπτώσεις υπερισχύουν τα μειονεκτήματα και αυτός είναι ο λόγος που αργήσαμε να τα παρουσιάσουμε. Στην επόμενη παράγραφο όμως θα δούμε και κάτι άλλο που τα κάνει ενδιαφέροντα.

15.12 Τυχαία Πρόσβαση σε Αρχεία – Μέθοδοι *seek* και *tell*

Σε ένα μη μορφοποιημένο αρχείο μπορούμε να γράφουμε τιμές διαφόρων τύπων. Αν όμως γράψουμε στοιχεία του ίδιου τύπου τότε όλες οι συνιστώσες ή **εγγραφές** (records) του αρχείου έχουν το ίδιο μήκος. Έτσι, στο παράδειγμα της προηγούμενης παραγράφου, αν, ας πούμε, το `sizeof(double)` είναι 8 ψηφιολέξεις, ξέρουμε ότι η εγγραφή 0 του αρχείου καταλαμβάνει τις ψηφιολέξεις από 0 μέχρι και 7, η εγγραφή 1 τις ψηφιολέξεις από 8 μέχρι και 15, η εγγραφή *k* τις ψηφιολέξεις από *k*·8 μέχρι και (*k*+1)·8–1. Αν λοιπόν μας δοθεί η δυνατότητα να διαβάζουμε ή να γράφουμε ξεκινώντας από οποιαδήποτε ψηφιολέξη του αρχείου, έχουμε τη δυνατότητα πρόσβασης σε οποιαδήποτε εγγραφή με την ίδια ευκολία· έχουμε, όπως λέμε, ένα **αρχείο τυχαίας πρόσβασης** (random access file). Στη C++ η τυχαία πρόσβαση επιτυγχάνεται με τις μεθόδους *seekg()* και *seekp()*.

Στο Κεφ. 8 είδαμε τη μέθοδο *seekg()*, και ειδικά τη χρήση `a.seekg(0)` που μας επιτρέπει την πρόσβαση στην αρχή ενός αρχείου συνδεμένου με το ρεύμα *a*. Τώρα θα δούμε τη χρήση της μεθόδου *seekg()* –των κλάσεων *ifstream*, και *fstream*– γενικότερα.

Αν έχουμε ένα εισερχόμενο ρεύμα *a*, συνδεμένο με κάποιο αρχείο, εκτέλεση της `a.seekg(n)`, όπου *n* φυσικός αριθμός, έχει ως αποτέλεσμα, η επόμενη εντολή εισόδου που θα εκτελεσθεί, να ξεκινήσει από την *n* ψηφιολέξη του αρχείου.

Όπως είδαμε παραπάνω

- ***Οι ψηφιολέξεις στο αρχείο αριθμούνται ξεκινώντας από 0 (μηδέν).***

Κατ' αρχάς να τελειώνουμε με το εξής: *Μπορούμε να χρησιμοποιούμε τη seekg() σε αρχεία text αλλά δεν έχει και τόσο νόημα*. Δες το παρακάτω

Παράδειγμα

Έστω τώρα ότι έχουμε ένα αρχείο –με όνομα `ranfl.txt`– και περιεχόμενο:

```
45␣␣␣␣89␣␣␣54
␣␣␣␣-3␣␣␣4.67␣␣␣dagdash␣␣␣jkfs␣jfejk
␣␣␣1
```

και το πρόγραμμα:

```
ifstream f( "ranfl.txt" );
double x;

for ( int k(0); k <= 12; k += 2 )
{
   f.seekg( k );  f >> x;
   cout << x << endl;
}
```

Τι θα πάρουμε από την εκτέλεσή του; Αυτά:

```
45
89
89
89
54
54
4
```

- Την πρώτη φορά έχουμε: `f.seekg(0); f >> x`, δηλαδή πήγαινε στην ψηφιολέξη 0 του αρχείου και διάβασε έναν πραγματικό· ο πρώτος αριθμός που διαβάζεται είναι ο 45.
- Την επόμενη φορά εκτελούνται οι `f.seekg(2); f >> x`, δηλαδή πήγαινε στην ψηφιολέξη 2 (το πρώτο διάστημα μετά το "45") του αρχείου και διάβασε έναν πραγματικό· τώρα, ο αριθμός που διαβάζεται είναι ο 89.
- Στη συνέχεια εκτελούνται οι `f.seekg(4); f >> x`, δηλαδή *επίστρεψε στο τρίτο διάστημα μετά το* "45" και διάβασε έναν πραγματικό· και πάλι διαβάζεται ο 89 που θα διαβαστεί και τρίτη φορά με την ανάγνωση που ακολουθεί την `f.seekg(6)`.

- Την πέμπτη και την έκτη φορά η *k* έχει τιμή 8 και 10 αντιστοίχως –έχει ξεπερασθεί το 89– και οι `f.seekg(k); f >> x` θα διαβάσουν το 54.
- Την τελευταία φορά η *k* έχει τιμή 12 και μετά την εκτέλεση της `f.seekg(k)`, το διάβασμα θα ξεκινήσει από το "4" του "54".

☜☜☜

Όπως βλέπεις, το πρόβλημα ξεκινάει από τον τρόπο που είναι γραμμένο ένα μορφοποιημένο αρχείο (text) σε αντιδιαστολή με ένα μη μορφοποιημένο που έχει (συνήθως) εγγραφές σταθερού μήκους.

Πάντως υπάρχουν δύο περιπτώσεις που χρησιμοποιούμε τη *seekg()* και σε μορφοποιημένα αρχεία:

- Με την `f.seekg(0)` πηγαίνουμε στην αρχή του αρχείου.
- Με την `f.seekg(0, ios_base::end)` πηγαίνουμε στο τέλος του αρχείου, όπως θα δούμε στη συνέχεια.

Οι κλάσεις *ofstream* και *fstream* έχουν τη μέθοδο *seekp*, για εξερχόμενα ρεύματα, με την οποίαν *καθορίζουμε την ψηφιολέξη του αρχείου από όπου θα αρχίσει το γράψιμο*.

Οι μέθοδοι *seekg()* και *seekp* είναι χρήσιμες σε μη μορφοποιημένα αρχεία με εγγραφές ίσου μήκους.

Έστω *x* μια μεταβλητή τύπου *T* και ένα ρεύμα `fstream a`, συνδεμένο με ένα αρχείο με τιμές τύπου *T*. Ανοίγουμε το ρεύμα, για διάβασμα και γράψιμο με την:

```
a.open( "...", ios_base::in|ios_base::out|ios_base::binary );
```

Οι εντολές

```
a.seekg( n*sizeof(T) );
a.read( reinterpret_cast<char*>(&x), sizeof(T) );
```

όπου *n* μη αρνητικός ακέραιος, ζητούν τα εξής:

- «να τοποθετηθεί η κεφαλή ανάγνωσης/εγγραφής» πριν από την υπ' αριθμό *n·sizeof*(*T*) ψηφιολέξη του αρχείου και
- η επόμενη *read* να ξεκινήσει από το σημείο αυτό, δηλαδή στην *x* θα αποθηκευτεί η υπ' αριθμόν *n* τιμή του αρχείου.

Παρομοίως, αν θέλεις να γράψεις το περιεχόμενο της *x* στην υπ' αριθμό *n* θέση του αρχείου θα πρέπει να δώσεις τις εντολές:

```
a.seekp( n*sizeof(T) );
a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

Χρησιμοποιώντας λοιπόν τις *seekg()* και *seekp()* μπορείς να διαβάζεις ή να γράφεις το ίδιο εύκολα οποιαδήποτε τιμή του αρχείου.

Αν λοιπόν έχεις να ενημερώσεις το περιεχόμενο κάποιας εγγραφής αυτό γίνεται ως εξής:

```
a.seekg( n*sizeof(T) );
a.read( reinterpret_cast<char*>(&x), sizeof(T) );
// Εντολές ενημέρωσης της x
a.seekp( n*sizeof(T) );
a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

«Δίδυμες» των *seekg()* και *seekp()* είναι οι *tellg()* και *tellp()*. Αν δώσεις

```
k = a.tellg();
```

όπου *a* ρεύμα ανοιγμένο για διάβασμα, στην *k* (`long`) θα έλθει ως τιμή η θέση από την οποία θα γίνει η επόμενη ανάγνωση. Παρομοίως, η:

```
k = a.tellp();
```

όπου *a* ρεύμα ανοιγμένο για γράψιμο, στην *k* (`long`) θα έλθει ως τιμή η θέση από την οποία θα ξεκινήσει η επόμενη εγγραφή.

15.12.1 Πώς Βρίσκουμε το Μέγεθος Αρχείου

Με χρήση των *seekg()* και *tellg()* μπορείς να βρεις το μέγεθος ενός αρχείου σε ψηφιολέξεις. Αν έχεις ανοίξει ένα ρεύμα με `ios_base::binary` για διάβασμα από το αρχείο που σε ενδιαφέρει τότε οι:

```
a.seekg( 0, ios_base::end );
k = a.tellg();
```

θα κάνουν τα εξής: Η πρώτη θα «πάει την κεφαλή ανάγνωσης στο τέλος του αρχείου» και η δεύτερη θα μας δώσει τη θέση της, που θα είναι ακριβώς το μέγεθος του αρχείου.

Γενικώς, η `a.seekg(n, ios_base::end)` ζητάει: η επόμενη ανάγνωση να ξεκινήσει *n* ψηφιολέξεις μετρώντας από το *τέλος του αρχείου*.

Παρατηρήσεις: ▶

Αν αυτή η δουλειά είναι η πρώτη που κάνεις μετά το άνοιγμα του αρχείου μπορείς να την κάνεις και ως εξής:

```
a.open( "myFileName.dta",
        ios_base::in|ios_base::binary|ios_base::ate );
k = a.tellg();
```

Θυμίσου (§8.12) ότι βάζοντας στην *open()* το `ios_base::ate` «*με το άνοιγμα του αρχείου τοποθετείται στο τέλος του.*» ◀

Παράδειγμα 🡆

Πολύ συχνά, όταν θέλουμε να επεξεργαστούμε κάποιο αρχείο text που δεν είναι πολύ μεγάλο, συμφέρει να το αντιγράψουμε ολόκληρο στην κύρια μνήμη του υπολογιστή, σε έναν πίνακα και να κάνουμε το πρόγραμμά μας πιο γρήγορο και πιο απλό. Να πώς μπορεί να γίνει αυτό με όσα μάθαμε στην παράγραφο αυτή:

```
ifstream a( "abc.txt",
            ios_base::in|ios_base::binary|ios_base::ate );
char t[16000];
long flSize;

// a.seekg( 0, ios_base::end ); αν δεν έχεις ios_base::ate
flSize = a.tellg();
if ( flSize <= 16000 )
{
   a.seekg( 0 );
   a.read( t, flSize );
}
a.close();
```

Παρατηρήσεις: ▶

1. Πρόσεξε το "`ios_base::binary`": είναι απαραίτητο για να φορτωθεί σωστά το αρχείο. Γενικώς: δίνεις *read()* ή *write()* για αρχείο που το έχεις ανοίξει με αυτό το χαρακτηριστικό.
2. Προσοχή στις αλλαγές γραμμών που, πιθανότατα, υπάρχουν στο αρχείο. Μπορεί να είναι **<cr><lf>** (στη C++:`'\r''\n'`), μπορεί να είναι κάτι άλλο![10]
3. Μη φανταστείς ότι, όταν φορτώσεις το αρχείο, θα μπει αυτομάτως κάποιο `'\0'` στο τέλος. Αν το χρειάζεσαι θα το βάλεις εσύ (`t[flSize] = '\0'`).
4. Αργότερα θα μάθουμε πώς να παίρνουμε ακριβώς όση μνήμη χρειαζόμαστε για το περιεχόμενο του αρχείου. ◀

🡆🡆🡆

Αν έχουμε ένα μη μορφοποιημένο αρχείο με τιμές τύπου *T* μπορούμε να υπολογίσουμε πόσες τιμές περιέχει[11]:

---

[10] Αν δεν βάλεις το "`ios_base::binary`" είναι πιθανό ότι ο μεταγλωττιστής θα βάλει τις εντολές που θα αλλάζουν (κατά την ανάγνωση) όλα τα `'\r''\n'` σε `'\n'`. Δεν το θέλεις όμως αυτό…

[11] Με την προϋπόθεση ότι η τιμές έχουν το ίδιο μέγεθος. Θα δεις στο επόμενο κεφάλαιο ότι αυτό δεν ισχύει πάντοτε.

- Βρίσκουμε το μέγεθος του αρχείου (όπως το *flSize* στο προηγούμενο πρόγραμμα) και
- Το διαιρούμε με το *μέγεθος φύλαξης ενός στοιχείου* τύπου *T* (`sizeof(T)` ή κάτι σχετικό).

15.13 Τιμή-Δομή σε Μη Μορφοποιημένο Αρχείο

Ενώ, όπως είδαμε, ένα μη μορφοποιημένο αρχείο με τιμές τύπου `int` ή `double` είναι περιορισμένης χρησιμότητας, ένα μη μορφοποιημένο αρχείο με τιμές τύπου `struct` (σταθερού μεγέθους) έχει ενδιαφέρον.

Ένα τέτοιο αρχείο έχει ανάγκη για πολλές ενημερώσεις. Όπως είδαμε παραπάνω, σε ένα τέτοιο αρχείο έχουμε τη δυνατότητα να κάνουμε ενημέρωση της κάθε τιμής **επι τόπου** (in situ, in place):

```
   a.seekg( n*sizeof(T) );
   a.read( reinterpret_cast<char*>(&x), sizeof(T) );
// Εντολές ενημέρωσης της x
   a.seekp( n*sizeof(T) );
   a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

και όχι με αντιγραφή σε άλλο αρχείο όπως γίνεται στα σειριακά αρχεία.

Με ένα τέτοιο μη μορφοποιημένο αρχείο το πρόβλημα που υπάρχει πάντοτε είναι αυτό της διαχείρισης με προγράμματα που έχουν αναπτυχθεί σε διαφορετικά περιβάλλοντα ανάπτυξης. Έτσι, το πρώτο πρόβλημα που μπορεί να προκύψει ξεκινάει από το ότι το `sizeof(T)` –πιθανότατα– δεν θα είναι παντού ίδιο. Αυτό θα προσπαθήσουμε να το ξεπεράσουμε ως εξής: αντί να φυλάγουμε μια τιμή-δομή με μια εντολή όπως η

```
   a.write( reinterpret_cast<const char*>(&x), sizeof(T) );
```

θα γράψουμε μια συνάρτηση που θα τη φυλάγει μέλος προς μέλος. Π.χ. για την

```
struct Address
{
   char country[17];
   char city[17];
   int  areaCode;
   char street[17];
   int  number;
}; // Address
```

θα γράψουμε μια:

```
void address_save( const Address& a, ostream& bout )
{
   bout.write( a.country, sizeof(a.country) );
   bout.write( a.city, sizeof(a.city) );
   bout.write( reinterpret_cast<const char*>(&a.areaCode),
               sizeof(a.areaCode) );
   bout.write( a.street, sizeof(a.street) );
   bout.write( reinterpret_cast<const char*>(&a.number),
               sizeof(a.number) );
}; // Address_save
```

Τώρα όμως στη *seekg()* και τη *seekp()* δεν θα βάζουμε το `sizeof(T)` αλλά το άθροισμα των μεγεθών φύλαξης των μελών. Θα μπορούσαμε να ορίσουμε την *Address* ως εξής:

```
struct Address
{
   enum { countrySz = 17, citySz = 17, streetSz = 17,
          saveSize = countrySz + citySz + sizeof(int) +
                     streetSz + sizeof(int) };
   char country[countrySz];
   char city[citySz];
   int  areaCode;
   char street[streetSz];
   int  number;
}; // Address
```

και να χρησιμοποιούμε τη *saveSize*:

```
   a.seekg( n*Address::saveSize );
   address_load( x, a );
// Εντολές ενημέρωσης της x
   a.seekp( n*Address::saveSize );
   address_save( x, a );
```

Παρατηρήσεις: ▶

1. Να τονίσουμε ότι το ρεύμα *a* έχει ανοιχτεί ως:

```
   fstream a( "...",
              ios_base::in|ios_base::out|ios_base::binary );
```

2. Πώς θα γράψουμε την *address_load()*; Με βάση την αρχή «διαβάζουμε όπως γράφουμε»:

```
void address_load( Address& a, istream& bin )
{
   bin.read( a.country, sizeof(a.country) );
   bin.read( a.city, sizeof(a.city) );
   bin.read( reinterpret_cast<char*>(&a.areaCode),
             sizeof(a.areaCode) );
   bin.read( a.street, sizeof(a.street) );
   bin.read( reinterpret_cast<char*>(&a.number),
             sizeof(a.number) );
}; // Address_load
```

Όπως βλέπεις υπάρχει αντιστοιχία 1-1 των *read()* της *address_load()* με τις *write()* της *address_save()*.

3. Για να βρούμε το πλήθος των εγγραφών στο αρχείο θα πρέπει να διαιρούμε δια *Address*`::`*saveSize* και όχι δια `sizeof(`*Address*`)`.

4. Λύσαμε όλα τα προβλήματα μεταφοράς του αρχείου από το ένα περιβάλλον σε οποιοδήποτε άλλο; Όχι, αλλά σε πολλές περιπτώσεις θα δουλεύει.◀

15.13.1 * Να Προτιμήσουμε τον Τύπο *string*;

Το ότι ορίσαμε τις σταθερές *countrySz*, *citySz*, *streetSz* μας απαλλάσσει και από τις τρεις «μαγικές σταθερές», τα μήκη των τριών πινάκων `char`. Μπορούμε όμως να τις «αξιοποιήσουμε» περισσότερο. Μπορούμε να χρησιμοποιήσουμε τον –πολύ πιο εύχρηστο– τύπο *string* για τα τρία μέλη και να τα μετατρέπουμε σε πίνακες μόνον για τη φύλαξη. Δηλαδή, ορίζουμε:

```
struct Address
{
   enum { countrySz = 17, citySz = 17, streetSz = 17,
          maxSize = 17,
          saveSize = countrySz + citySz + sizeof(int) +
                     streetSz + sizeof(int) };
   string country;
   string city;
   int    areaCode;
   string street;
   int    number;
}; // Address
```

Τώρα, η *address_save()* γίνεται:

```
void address_save( const Address& a, ostream& bout )
{
   char tmp[Address::maxSize];
   strncpy( tmp, a.country.c_str(), Address::countrySz-1 );
                                 tmp[Address::countrySz-1] = '\0';
   bout.write( tmp, Address::countrySz );
   strncpy( tmp, a.city.c_str(), Address::citySz-1 );
                                    tmp[Address::citySz-1] = '\0';
   bout.write( tmp, Address::citySz );
   bout.write( reinterpret_cast<const char*>(&a.areaCode),
```

```
                sizeof(a.areaCode) );
   strncpy( tmp, a.street.c_str(), Address::streetSz-1 );
                                   tmp[Address::streetSz-1] = '\0';
   bout.write( tmp, Address::streetSz );
   bout.write( reinterpret_cast<const char*>(&a.number),
                sizeof(a.number) );
}; // address_save
```

Πώς γίνεται η φύλαξη του *a.country*;

- Κατ' αρχάς θα φυλάξουμε *countrySz* ψηφιολέξεις το πολύ.
- Αντιγράφουμε *countrySz*–1 (το πολύ) χαρακτήρες στον tmp και σιγουρεύουμε ότι στο τέλος (`tmp[countrySz-1]`) θα υπάρχει ο φρουρός· στη συνέχεια θα καταλάβεις πού χρειάζεται.
- Φυλάγουμε τον tmp στο αρχείο (*countrySz* χαρακτήρες).

Δεν θα μπορούσαμε να φυλάξουμε *countrySz* χαρακτήρες κατ' ευθείαν το `a.country.c_str()`; Αν έχει μήκος μεγαλύτερο από *countrySz* δεν θα φυλαχθεί ο φρουρός.[12]

Με τον ίδιο τρόπο φυλάγουμε και τα μέλη *city*, *street*.

Η φόρτωση θα γίνει με την:

```
void address_load( Address& a, istream& bin )
{
   char tmp[Address::maxSize];
   bin.read( tmp, Address::countrySz );   a.country = tmp;
   bin.read( tmp, Address::citySz );   a.city = tmp;
   bin.read( reinterpret_cast<char*>(&a.areaCode),
             sizeof(a.areaCode) );
   bin.read( tmp, Address::streetSz );   a.street = tmp;
   bin.read( reinterpret_cast<char*>(&a.number),
             sizeof(a.number) );
}; // address_load
```

Για να εκτελεσθεί σωστά η `a.country = tmp` (και οι παρόμοιες) ο tmp θα πρέπει να έχει τον φρουρό (`'\0'`).

Δεν θα μπορούσαμε να φυλάγουμε τις τιμές των τριών μελών τύπου *string* με το ακριβές περιεχόμενο που έχουν κάθε φορά οποιοδήποτε μήκος και αν έχει; Ναι, αρκεί να φυλάγουμε και την τιμή του (κάθε) μήκους. Α, και κάτι άλλο: χάνουμε όλα εκείνα τα πλεονεκτήματα που λέγαμε ότι βασίζονται στο σταθερό μήκος εγγραφής· η διαχείριση γίνεται πολύπλοκη...

15.14 Ένα Παράδειγμα

Το πρόβλημα:

> *Μας δίνεται ένα αρχείο, με όνομα στο δίσκο* `elements.dta`*, που περιέχει πληροφορίες για τα χημικά στοιχεία. Το αρχείο έχει εγγραφές τύπου:*

```
struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn
```

> *που έχουν φυλαχθεί με τη συνάρτηση:*

```
void Elmn_save( const Elmn& a, ostream& bout )
{
   bout.write( reinterpret_cast<const char*>(&a.eANumber),
               sizeof(a.eANumber) );
   bout.write( reinterpret_cast<const char*>(&a.eAWeight),
```

[12] Μπορείς να σκεφτείς άλλη λύση του προβλήματος;

```
                sizeof(a.eAWeight) );
   bout.write( a.eSymbol, sizeof(a.eSymbol) );
   bout.write( a.eName, sizeof(a.eName) );
} // Elmn_save
```

Στην 1η εγγραφή υπάρχουν πληροφορίες για το Υδρογόνο (ατομικός αριθμός: 1), στη δεύτερη πληροφορίες για το Ήλιο (α.α.: 2) και γενικώς στην k-οστή εγγραφή πληροφορίες για το στοιχείο με α.α.= k.
Στο μέλος eName υπάρχει το όνομα του στοιχείου στα αγγλικά.
Θέλουμε να αντιγράψουμε το αρχείο σε ένα άλλο, με όνομα: **elementsGr.dta**, *που θα έχει εγγραφές του εξής τύπου:*

```
struct GrElmn
{
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[4];
   char               geName[14];
   char               geGrName[14];
}; // GrElmn
```

Στο μέλος geGrName θα βάλουμε το ελληνικό όνομα του στοιχείου. Η συμπλήρωση του νέου μέλους θα πρέπει να μπορεί να γίνεται τμηματικά, όποτε θέλει ο χρήστης.

Είναι φανερό ότι εδώ έχουμε δύο εντελώς ξεχωριστές δουλειές:

- Αντιγραφή του παλιού αρχείου στο νέο· αυτή η δουλειά θα γίνει μια φορά μόνο.
- Συμπλήρωση του ελληνικού ονόματος. Αυτή η δουλειά, όπως μας λέει και η διατύπωση του προβλήματος «*θα πρέπει να μπορεί να γίνεται τμηματικά, όποτε θέλει ο χρήστης.*»

Έχουμε λοιπόν να γράψουμε δύο προγράμματα.

15.14.1 Το Πρώτο Πρόγραμμα

Το πρώτο πρόγραμμα είναι απλό: Δουλεύουμε το αρχείο σειριακά, αφού θα πρέπει να το αντιγράψουμε ολόκληρο:

```
Άνοιξε ρεύμα bin από το παλιό αρχείο
Άνοιξε ρεύμα bout προς το νέο αρχείο
Διάβασε μια εγγραφή
while ( !bin.eof() )
{
   Αντίγραψε την εγγραφή
   Γράψε τη νέα εγγραφή
   Διάβασε την επόμενη εγγραφή
} // while
Κλείσε τα ρεύματα
```

Ανοίγουμε τα ρεύματα με τη δήλωσή τους και ελέγχουμε αν άνοιξαν:

```
      ifstream bin( "elements.dta", ios_base::binary );
      if ( bin.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotOpen,
                           "elements.dta" );
      ofstream bout( "elementsGr.dta", ios_base::binary );
      if ( bout.fail() )
      {
         bin.close();
         throw ApplicXptn( "main", ApplicXptn::cannotCreate,
                           "elementsGr.dta" );
      }
// ...
```

Πρόσεξε τι κάνουμε με το δεύτερο ρεύμα: Αν αποτύχει το άνοιγμα, πριν ρίξουμε τη σχετική εξαίρεση, *κλείνουμε το πρώτο ρεύμα που είναι ήδη ανοιχτό*!

Και τώρα το διάβασμα: Θα γράψουμε μια συνάρτηση *Elmn_load()* για να διαβάζουμε τα δεδομένα ενός στοιχείου με αποκλειστικό οδηγό την *Elmn_save()* («διαβάζουμε όπως γράψαμε»).

```
void Elmn_load( Elmn& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.eANumber),
             sizeof(a.eANumber) );
   bin.read( reinterpret_cast<char*>(&a.eAWeight),
             sizeof(a.eAWeight) );
   bin.read( a.eSymbol, sizeof(a.eSymbol) );
   bin.read( a.eName, sizeof(a.eName) );
   if ( bin.fail() && !bin.eof() )
      throw ApplicXptn( "Elmn_load", ApplicXptn::cannotRead );
} // Elmn_load
```

Πρόσεξε ότι εδώ ρίχνουμε και εξαίρεση αν δεν μπορέσουμε να διαβάσουμε από το ρεύμα εισόδου *bin* για οποιονδήποτε λόγο εκτός από "`bin.eof()`".

Αν λοιπόν έχουμε δηλώσει:

```
   Elmn oneElmn;
```

η «*Διάβασε μια εγγραφή*» γίνεται:

```
   Elmn_load( oneElmn, bin );
```

Πριν αρχίσουμε να ασχολούμαστε με τη *GrElmn* να την προσαρμόσουμε σε αυτά που είπαμε στην προηγούμενη παράγραφο:

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

Γυρνώντας στο σχέδιό μας, έχουμε να υλοποιήσουμε την «*Αντίγραψε την εγγραφή*». Μπορούμε να λύσουμε αυτό το πρόβλημα με μια συνάρτηση που θα τροφοδοτείται με ένα αντικείμενο τύπου *Elmn* και θα υπολογίζει και θα επιστρέφει ένα αντικείμενο *GrElmn*. Οι κανόνες μας λένε ότι πρέπει να γράψουμε συνάρτηση με τύπο:

```
GrElmn GrElmn_copyFromElmn( const Elmn& a )
{
   GrElmn fv;
   fv.geANumber = a.eANumber;
   fv.geAWeight = a.eAWeight;
   strcpy( fv.geSymbol, a.eSymbol );
   strcpy( fv.geName, a.eName );
   fv.geGrName[0] = '\0';
   return fv;
} // GrElmn_copyFromElmn
```

Αυτή συνάρτηση καλείται ως εξής:

```
         GrElmn oneGrElmn;
         oneGrElmn = GrElmn_copyFromElmn( oneElmn );
```

Αργότερα θα μάθουμε πώς μπορούμε να δώσουμε καλύτερη λύση.

Η «*Γράψε τη νέα εγγραφή*» υλοποιείται με την:

```
void GrElmn_save( const GrElmn& a, ostream& bout )
{
   bout.write( reinterpret_cast<const char*>(&a.geANumber),
               sizeof(a.geANumber) );
   bout.write( reinterpret_cast<const char*>(&a.geAWeight),
               sizeof(a.geAWeight) );
```

```
   bout.write( a.geSymbol, sizeof(a.geSymbol) );
   bout.write( a.geName, sizeof(a.geName) );
   bout.write( a.geGrName, sizeof(a.geGrName) );
   if ( bout.fail() )
      throw ApplicXptn( "GrElmn_save", ApplicXptn::cannotWrite );
} // GrElmn_save
```

Όπως βλέπεις, ρίχνουμε και εδώ εξαίρεση αν για κάποιον λόγο αποτύχει το γράψιμο. Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct ApplicXptn
{
   enum { cannotOpen, cannotCreate, cannotClose,
          cannotWrite, cannotRead };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ApplicXptn

struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn

void Elmn_load( Elmn& a, istream& bin );

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

void GrElmn_save( const GrElmn& a, ostream& bout );
GrElmn GrElmn_copyFromElmn( const Elmn& a );

int main()
{
   try
   {
   // Άνοιξε ρεύμα bin από το παλιό αρχείο
      ifstream bin( "elements.dta", ios_base::binary );
      if ( bin.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotOpen,
                           "elements.dta" );
   // Άνοιξε ρεύμα bout προς το νέο αρχείο
      ofstream bout( "elementsGr.dta", ios_base::binary );
      if ( bout.fail() )
      {
         bin.close();
```

```
         throw ApplicXptn( "main", ApplicXptn::cannotCreate,
                           "elementsGr.dta" );
      }
   // Διάβασε μια εγγραφή
      Elmn oneElmn;
      int k( 0 );
      Elmn_load( oneElmn, bin );
      while ( !bin.eof() )
      {
      // Αντίγραψε την εγγραφή
         GrElmn oneGrElmn;
         oneGrElmn = GrElmn_copyFromElmn( oneElmn );
      // Γράψε τη νέα εγγραφή
         GrElmn_save( oneGrElmn, bout );
         ++k;
      // Διάβασε την επόμενη εγγραφή
         Elmn_load( oneElmn, bin );
      }
   // Κλείσε τα ρεύματα
      bout.close();
      if ( bout.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           "elementsGr.dta" );
      bin.close();
      cout << "Αντίγραψα " << k << " εγγραφές" << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotCreate:
            cout << "cannot create file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "cannot close file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn ...
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Πρόσεξε πώς ορίζουμε την κλάση εξαιρέσεων, πώς ρίχνουμε εξαιρέσεις και πώς τις συλλαμβάνουμε.

Ειδικώς για τις περιπτώσεις ανοίγματος αρχείων, μπορείς να πεις ότι τα παρατάμε πολύ εύκολα. Θα μπορούσαμε, σε περίπτωση που κάτι δεν πάει καλά, να δώσουμε την ευκαι-

ρία στον χρήστη να κάνει κάτι. Δεν το κάνουμε διότι αυτό είναι ένα πρόγραμμα που θα δουλέψει μόνον μια φορά!

Θα πεις: «Πολύ κακό για το τίποτε! Θα μπορούσαμε να γράψουμε το πρόγραμμα σε δέκα γραμμές!» Σωστό! Το πρόγραμμα γράφτηκε έτσι για διδακτικούς λόγους (και είναι ένα καλό πρόγραμμα).

15.14.2 Το Δεύτερο Πρόγραμμα

Ας έρθουμε τώρα στο δεύτερο πρόγραμμα. Τώρα έχουμε να δουλέψουμε μόνο με ένα αρχείο, το **elementsGr.dta**, του οποίου τις εγγραφές ενημερώνουμε (διαβάζουμε – αλλάζουμε – ξαναγράφουμε). Δηλώνουμε λοιπόν:

```
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
```

και το ανοίγουμε ως εξής:

```
   bool ok;
   do {
      bInOut.open( flNm.c_str(),
                   ios_base::in|ios_base::out|ios_base::binary );
      if ( bInOut.fail() )
      {
         ok = false;
         cout << "cannot open " << flNm << endl;
         cout << "file name ('x' to exit): ";
         getline( cin, flNm, '\n' );
      }
      else
         ok = true;
   } while ( !ok && flNm != "x" && flNm != "X" );
```

Καλό είναι να βάλουμε τα παραπάνω σε μια συνάρτηση:

```
void openFile( string& flNm, fstream& bInOut )
{
   string prevFlNm;
   bool ok;
   do {
      bInOut.open( flNm.c_str(),
                   ios_base::in|ios_base::out|ios_base::binary );
      if ( bInOut.fail() )
      {
         ok = false;
         prevFlNm = flNm;
         cout << "cannot open " << flNm << endl;
         cout << "file name ('x' to exit): ";
         getline( cin, flNm, '\n' );
      }
      else
         ok = true;
   } while ( !ok && flNm != "x" && flNm != "X" );
   if ( !ok )
      throw ApplicXptn( "openFile", ApplicXptn::cannotOpen,
                        prevFlNm.c_str() );
} // openFile
```

Στην περίπτωση που ο χρήστης τα παρατήσει η *flNm* θα έχει τιμή **"x"** ή **"X"**. Φυσικά, ένα τέτοιο όνομα στην εξαίρεση δεν έχει νόημα. Για τον λόγο αυτόν φυλάγουμε στην *prevFlNm* το τελευταίο όνομα που δοκιμάσαμε στην *open()*.

Ας κάνουμε τώρα ένα σχέδιο για το πρόγραμμά μας. Μια απλή ιδέα είναι η εξής: ο χρήστης δίνει τον ατομικό αριθμό (α.α.) του στοιχείου. Ποιες είναι οι δεκτές τιμές για τον α.α.; Από 1 μέχρι το πλήθος των εγγραφών του αρχείου. Διαβάζουμε από το αρχείο την εγγραφή που αντιστοιχεί στο στοιχείο και την εμφανίζουμε στο χρήστη. Ο χρήστης μας δίνει το

ελληνικό όνομα που εισάγουμε στο μέλος *grName*. Τέλος, ξαναγράφουμε την εγγραφή στο αρχείο, στην παλιά της θέση. Αυτή η δουλειά θα γίνεται ξανά και ξανά, μέχρι... Μέχρι να τελειώσει τη δουλειά του ο χρήστης (ή να βαρεθεί). Αφού δεν υπάρχει στοιχείο με ατομικό αριθμό 0, ας κάνουμε τη συμφωνία: όταν δώσει 0 ως α.α. θα τελειώνουμε. Θα έχουμε δηλαδή επανάληψη με φρουρό.

Ωραίο σενάριο, ας το γράψουμε σε διακριτά βήματα:

```
Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
Διάβασε τον α.α.
while ( α.α. != 0 )
{
   Διάβασε από το αρχείο την αντίστοιχη εγγραφή
   Δείξε το περιεχόμενο της εγγραφής στο χρήστη
   Διάβασε το ελληνικό όνομα
   Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
   Διάβασε τον α.α.
}
```

Για την «*Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)*» γράφουμε τη συνάρτηση

```
void countRecords( fstream& bInOut, unsigned int& noOfRecords )
{
   unsigned long int initialPos;
   unsigned long int flSize;

   initialPos = bInOut.tellg();
   bInOut.seekg( 0, ios_base::end );  flSize = bInOut.tellg();
   noOfRecords = flSize / GrElmn::saveSize;
   bInOut.seekg( initialPos );
} // countRecords
```

δηλώνουμε μια μεταβλητή:

```
   unsigned int maxAtNo;
```

και αμέσως μετά το άνοιγμα του ρεύματος γράφουμε:

```
         countRecords( bInOut, maxAtNo );
```

Πρόσεξε τον ρόλο της μεταβλητής *initialPos*: φυλάγουμε την αρχική θέση του ρεύματος (`initialPos = bInOut.tellg()`) και στο τέλος, αφού υπολογίσουμε το πλήθος εγγραφών, το ξαναφέρνουμε στην αρχική του κατάσταση (`bInOut.seekg( initialPos )`).

Η «Διάβασε τον ατομικό αριθμό» μεταφράζεται εύκολα σε C++:

```
void readAtNo( int maxAtNo, int& aa )
{
   string aastr;

   do {
      cout << " Atomic Number (1.." << maxAtNo
           << ", 0 to exit): ";
      getline( cin, aastr, '\n' );
      aa = atol( aastr.c_str() );
   } while( aa < 0 || maxAtNo < aa );
} // readAtNo
```

Για τη «Διάβασε από το αρχείο την αντίστοιχη εγγραφή» θα γράψουμε μια συνάρτηση σαν την *Elmn_load()*, ας την πούμε *GrElmn_load*:

```
void GrElmn_load( GrElmn& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.geANumber),
             sizeof(a.geANumber) );
   bin.read( reinterpret_cast<char*>(&a.geAWeight),
             sizeof(a.geAWeight) );
   bin.read( a.geSymbol, sizeof(a.geSymbol) );
   bin.read( a.geName, sizeof(a.geName) );
   bin.read( a.geGrName, sizeof(a.geGrName) );
   if ( bin.fail() )
```

```
      throw ApplicXptn( "GrElmn_load", ApplicXptn::cannotRead );
} // GrElmn_load
```

Αλλά αυτή δεν είναι αρκετή: τώρα θα πρέπει να έχουμε τυχαία πρόσβαση, που θα γίνεται *με βάση τον ατομικό αριθμό που έδωσε ο χρήστης*. Και από ποια θέση θα πάμε να διαβάσουμε; Στην υπ' αριθμό 0 (μηδέν) εγγραφή του αρχείου βρίσκονται οι πληροφορίες για το Υδρογόνο, που έχει α.α. 1, στην υπ' αριθμό 1 εγγραφή βρίσκονται οι πληροφορίες για το Ήλιο, που έχει α.α. 2 και γενικώς: στην υπ' αριθμό *k*–1 εγγραφή υπάρχουν οι πληροφορίες για το στοιχείο που έχει α.α. *k*. Στην περίπτωσή μας, το μήκος της εγγραφής, σε ψηφιολέξεις, είναι *GrElmn*::*saveSize*· άρα η *aa*–1 εγγραφή, στην οποία υπάρχουν τα στοιχεία για το στοιχείο με α.α. *aa*, αρχίζει στην ψηφιολέξη `(`*aa*`-1)*`*GrElmn*`::`saveSize`.

Ας ξεχωρίσουμε τα προβλήματά μας: Ας γράψουμε πρώτα μια μέθοδο που διαβάζει μια εγγραφή παίρνοντας τον αριθμό της:

```
void readRandom( GrElmn& a, istream& bin, int atNo )
{
   if ( bin.fail() )
      throw ApplicXptn( "readRandom", ApplicXptn::streamNotOpen );
   if ( atNo <= 0 )
      throw ApplicXptn( "readRandom",
                        ApplicXptn::rangeErr, atNo );
   bin.seekg( (atNo-1)*GrElmn::saveSize );
   GrElmn_load( a, bin );
} // readRandom
```

Μέσα στη **main** βάζουμε τη δήλωση:

```
            GrElmn a;
```

και την κλήση:

```
            readRandom( a, bInOut, aa );
```

Για την υλοποίηση της «Δείξε το περιεχόμενο της εγγραφής στο χρήστη» γράφουμε μια συνάρτηση:

```
void GrElmn_display( const GrElmn& a, ostream& tout )
{
   tout << "atomic number: " << a.geANumber << endl
        << "atomic weight: " << a.geAWeight << endl
        << "symbol: " << a.geSymbol << endl
        << "name: " << a.geName << endl
        << "greek name: " << a.geGrName << endl;
} // GrElmn_display
```

Γιατί δείχνουμε το *geGrName*; Δεν θα είναι «κενό»; Μπορεί να είναι· αλλά αυτό το πρόγραμμα θα μπορεί να χρησιμοποιηθεί και για διόρθωση λαθεμένων τιμών, οπότε το παλιό όνομα χρειάζεται.

Η *GrElmn_display* καλείται από τη **main** ως εξής:

```
            GrElmn_display( a, cout );
```

Θα έλεγε κανείς ότι η «Διάβασε το ελληνικό όνομα» είναι απλή:

```
         cin >> newGrName;
         strcpy( a.geGrName, newGrName.c_str() );
```

όπου *newGrName* μεταβλητή τύπου *string*. Αλλά, αφού, όπως είπαμε, το πρόγραμμα θα μπορεί να χρησιμοποιηθεί και για διόρθωση τιμών, θα πρέπει να δίνουμε τη δυνατότητα στον χρήστη να αφήσει το ελληνικό όνομα αναλλοίωτο.

- Ας πούμε λοιπόν ότι αν ο χρήστης πιέσει απλώς το <enter> δεν θα αλλάξει το ελληνικό όνομα. Αυτό δεν μπορεί να γίνει με τη "`cin >> newName`", που, όπως ξέρεις, επιμένει να διαβάσει κάτι. Μπορεί να γίνει με τη "`getline( cin, newGrName, '\n' )`".
- Αν δεν δοθεί νέο ελληνικό όνομα δεν υπάρχει λόγος να ξαναφυλάξουμε την εγγραφή στο αρχείο.

Αλλάζουμε λοιπόν το σχέδιό μας· αντί για τις:

```
Διάβασε από το αρχείο την αντίστοιχη εγγραφή
Δείξε το περιεχόμενο της εγγραφής στο χρήστη
Διάβασε το ελληνικό όνομα
Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
```

θα έχουμε:

```
Διάβασε από το αρχείο την αντίστοιχη εγγραφή
Δείξε το περιεχόμενο της εγγραφής στο χρήστη
Διάβασε το ελληνικό όνομα
if ( άλλαξε το ελληνικό όνομα )
   Γράψε την εγγραφή στο αρχείο στην παλιά της θέση
```

Ενσωματώνουμε τα παραπάνω στην:

```
void editGrName( fstream& bInOut, int atNo )
{
   GrElmn a;
   readRandom( a, bInOut, atNo );
   GrElmn_display( a, cout );
   string newGrName;
   cout << "new greek name: "; getline( cin, newGrName, '\n' );
   if ( !newGrName.empty() )
   {
      GrElmn_setGrName( a, newGrName );
      writeRandom( a, bInOut );
   }
} // editGrName
```

και τη χρησιμοποιούμε στη **main** ως εξής:

```
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
            editGrName( bInOut, aa );
```

Η *GrElmn_setGrName()* είναι απλή:

```
void GrElmn_setGrName( GrElmn& a, string newGrName )
{
   strncpy( a.geGrName, newGrName.c_str(), GrElmn::grNameSz-1 );
   a.geGrName[GrElmn::grNameSz-1] = '\0';
} // GrElmn_setGrName
```

Η *writeRandom()* είναι η «δίδυμη» της *readRandom()* αλλά εδώ έχουμε μια ασυμμετρία: δεν υπάρχει παράμετρος για τον ατομικό αριθμό, διότι αυτός υπάρχει ως μέλος της δομής *a*. Κατά τα άλλα:

```
void writeRandom( const GrElmn& a, ostream& bout )
{
   if ( bout.fail() )
      throw ApplicXptn( "writeRandom",
                        ApplicXptn::streamNotOpen );
   bout.seekp( (a.geANumber-1)*GrElmn::saveSize );
   GrElmn_save( a, bout );
} // writeRandom
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct ApplicXptn
{
   enum { cannotOpen, streamNotOpen, rangeErr, cannotClose,
          cannotWrite, cannotRead };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   int  errIntVal;
```

```
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
}; // ApplicXptn

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

void GrElmn_save( const GrElmn& a, ostream& bout );
void GrElmn_load( GrElmn& a, istream& bin );
void GrElmn_display( const GrElmn& a, ostream& tout );
void GrElmn_setGrName( GrElmn& a, string newGrName );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrName( fstream& bInOut, int atNo );

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrName( bInOut, aa );
         }
      } while ( aa != 0 );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "GrElmn_load", ApplicXptn::cannotClose,
                           flNm.c_str() );
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::streamNotOpen:
```

```
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::rangeErr:
            cout << "no element with such atomic number ("
                 << x.errIntVal << ") in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "cannot close file " << x.errIntVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Να και ένα παράδειγμα εκτέλεσης:

```
 Atomic Number (1..103, 0 to exit): 22
atomic number: 22
atomic weight: 47.9
symbol: Ti
name: Titanium
greek name:
new greek name: Τιτάνιο
 Atomic Number (1..103, 0 to exit): 55
atomic number: 55
atomic weight: 132.906
symbol: Cs
name: Cesium
greek name:
new greek name: Κέσιο
 Atomic Number (1..103, 0 to exit): 35
atomic number: 35
atomic weight: 79.904
symbol: Br
name: Bromine
greek name:
new greek name: Βρώμιο
 Atomic Number (1..103, 0 to exit): 0
```

15.14.3 Για το Παράδειγμά μας

Στο παράδειγμα αυτό είδαμε:

- Αρχεία εγγραφών (τιμές τύπου δομής),
- Αρχεία τυχαίας πρόσβασης.

Κατ' αρχάς να παρατηρήσουμε ότι:

- Το πρώτο πρόγραμμα δημιούργησε το αρχείο ως *σειριακό*.

α.α	Όνομα	σύμβ.	ατομικό βάρος
1	Υδρογόνο (Hydrogen)	H	1.0008
2	Ήλιο (Helium)	He	4.0026
3	Λίθιο (Lithium)	Li	6.941
4	Βηρύλλιο (Beryllium)	Be	9.01218
	. . .		

Πίν. 15-1 Τα δεδομένα για το κάθε στοιχείο βρίσκονται σε θέση (στο αρχείο) που σχετίζεται με τον ατομικό αριθμό του.

- Το δεύτερο πρόγραμμα επεξεργάζεται το αρχείο με *τυχαία πρόσβαση*. Μπορεί όμως να το χειριστεί *και ως σειριακό*. Για παράδειγμα, ας πούμε ότι θέλουμε να βγάλουμε έναν πίνακα σαν αυτόν που βλέπεις στον Πίν. 15-1. Γράφοντας την

```
void GrElmn_writeToTable( const GrElmn& a, ostream& tout )
{
   tout << a.geANumber << '\t' << a.geGrName
        << " (" << a.geName << ")\t" << a.geSymbol << '\t'
        << a.geAWeight << endl;
} // GrElmn_writeToTable
```

μπορούμε με τις εντολές:

```
      ofstream tout( "elementsTbl.txt" );
      GrElmn a;
      bInOut.seekg( 0 );
      GrElmn_load( a, bInOut );
      while ( !bInOut.eof() )
      {
         GrElmn_writeToTable( a, tout );
         GrElmn_load( a, bInOut );
      } // while
      tout.close();
```

να πάρουμε το αρχείο **elementsTbl.txt** με περιεχόμενο της μορφής:[13]

```
1\tΥδρογόνο (Hydrogen)\tH\t1.008
2\tΗλιο (Helium)\tHe\t4.0026
3\tΛίθιο (Lithium)\tLi\t6.941
4\tΒηρύλλιο (Beryllium)\tBe\t9.01218
 . . .
```

Όπως βλέπεις, έχουμε βάλει σε μια ομάδα τις συναρτήσεις που αρχίζουν με "**GrElmn_**". Αυτές έχουν το εξής κοινό χαρακτηριστικό: κάθε μια κάνει μια συγκεκριμένη δουλειά με μια μεταβλητή (ένα αντικείμενο) τύπου *GrElmn*. Αργότερα θα δεις ότι θα τις ενσωματώσουμε στο αντικείμενο και θα τις ονομάσουμε **μεθόδους** (methods) για τον χειρισμό του.

Ένα άλλο πράγμα που δείχνουμε είναι η διαχείριση εξαιρέσεων. Μεταξύ μας: μερικές από τις εξαιρέσεις που ετοιμαζόμαστε να συλλάβουμε δεν υπάρχει περίπτωση να ριχτούν. Η διαχείριση μπήκε για διδακτικούς λόγους.

Το παράδειγμά μας επιλέχτηκε για τον εξής λόγο: Ο *Ατομικός Αριθμός* (α.α.) είναι το σημαντικότερο χαρακτηριστικό ενός στοιχείου, έχει δηλαδή πολύ νόημα να λέμε «το χημικό στοιχείο με α.α. 6». Από την άλλη μεριά, ο α.α. παίρνει τιμές από 1 μέχρι 112 (ή 113 ή 114). Έτσι, ήταν πολύ απλό και φυσικό να πούμε ότι: θα βάλουμε τα δεδομένα για το στοιχείο με α.α. *k* στην (*k*-1)-οστή εγγραφή του αρχείου.

Ας πούμε όμως ότι θέλει να χρησιμοποιήσει το αρχείο –με το πρόγραμμά μας– κάποιος όχι και τόσο ειδικός. Για έναν τέτοιον χρήστη έχει περισσότερο νόημα το όνομα «άνθρακας» –ή ο αγγλικός όρος «carbon»– από το «α.α. 6». Αυτός πώς θα βρίσκει τα στοιχεία που θέλει; Αν έχει ένα **ευρετήριο** (index) σαν αυτό του Πίν. 15-2 μπορεί να κάνει τη δουλειά του.

Φυσικά, το σωστό είναι να έχουμε ένα ευρετήριο που θα το χειρίζεται ο υπολογιστής. Στην περίπτωση αυτή βέβαια δεν έχει νόημα να δίνουμε τον ατομικό αριθμό· μπορούμε να δίνουμε κατ' ευθείαν τον αριθμό της ψηφιολέξης από την οποίαν αρχίζει η αποθήκευση των δεδομένων για το συγκεκριμένο στοιχείο, Και πραγματικά, τέτοια ευρετήρια χρησιμοποιούνται στα **Συστήματα Διαχείρισης Βάσεων Στοιχείων** (Data Base Management Systems, DBMS) όπως και σε απλά συστήματα διαχείρισης αρχείων. Φυσικά, η υλοποίηση των ευρετηρίων γίνεται με μεθόδους πιο πολύπλοκες αλλά και πιο αποδοτικές από άποψη ταχύτητας και χρήσης μνήμης. Αργότερα θα δούμε μια πολύ απλή μορφή ευρετηρίου.

---

[13] Φυσικά, οι στηλοθέτες (tabs) δεν θα φαίνονται. Αν εισαγάγεις το αρχείο σε ένα εργαλείο σαν το MS Excel ή το MS Word παίρνεις τον πίνακα.

Εδώ να παρατηρήσουμε και το εξής: αν στη δεύτερη στηλη του πίνακα δεν έχουμε τον α.α. αλλά την ψηφιολέξη του αρχείου στην οποίαν αρχίζει η αποθήκευση των δεδομένων για το συγκεκριμένο στοιχείο τότε αλλάζουν πολλά πράγματα: μπορείς να ψάχνεις και εγγραφές μεταβλητού μήκους (μπορείς να ψάχνεις και μορφοποιημένα αρχεία). Αλλά να ψάχνεις μόνο· η ενημέρωση είναι πολύπλοκη.

Όνομα	Ατομικός Αριθμός
Actinium	89
Aluminum	13
Americium	95
Antimony	51
Argon	18
Arsenic	33
Astatine	85
Barium	56
. . .	. . .

Πίν. 15-2 Τα δεδομένα για το κάθε στοιχείο βρίσκονται σε θέση (στο αρχείο) που σχετίζεται με τον ατομικό αριθμό του.

15.15 Ανακεφαλαίωση

Οι τύποι-δομές μας επιτρέπουν να παριστάνουμε και να διαχειριζόμαστε τιμές-αντικείμενα που απαρτίζονται από άλλες τιμές, άλλων τύπων και αναφέρονται στην ίδια φυσική οντότητα. Μπορούμε να διαχειριζόμαστε ένα αντικείμενο είτε ολόκληρο είτε κατά μέλη.

Εκτός από μορφοποιημένα αρχεία (text) μπορούμε να χρησιμοποιούμε και *μη μορφοποιημένα* στα οποία τα δεδομένα αντιγράφονται από τη μνήμη χωρίς να μετατραπούν σε χαρακτήρες. Κυριότερο πλεονέκτημα: η *ταχύτητα*, κυριότερο μειονέκτημα: *περιορίζεται η δυνατότητα μεταφοράς*.

Με τη μέθοδο *seekp()* (των *ofstream*, *fstream*) μπορείς να μετακινηθείς για να γράψεις σε οποιαδήποτε θέση του αρχείου. Παρομοίως, για να διαβάσεις από οποιαδήποτε θέση του αρχείου χρησιμοποίησε τη *seekg()* (των *ifstream*, *fstream*).

Στο παράδειγμα με τα χημικά στοιχεία πήραμε μια πρώτη γεύση για το πώς δουλεύουμε με *εξαιρέσεις*. Έτσι περίπου θα γράφουμε τα προγράμματά μας από εδώ και πέρα.

Το κεφάλαιο αυτό συνοδεύεται και από δύο projects. Να τα διαβάσεις οπωσδήποτε και να γυρίσεις στα προηγούμενα κεφάλαια για να ξαναδείς ό,τι δεν θυμάσαι. Το πρώτο (ξανα)δίνεται κυρίως για να το δούμε με εξαιρέσεις. Το δεύτερο θα ξαναδοθεί με άλλον τρόπο στη συνέχεια.

Ασκήσεις

Β Ομάδα

15-1 Μας δίνεται μη μορφοποιημένο (binary) αρχείο με όνομα `dates.dta` και άγνωστο πλήθος τιμών τύπου:

```
struct Date
{
   unsigned int  year;
   unsigned char month;
   unsigned char day;
}; // Date
```

που έχουν φυλαχθεί με τη

```
void Date_save( const Date& a, ostream& bout )
{
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&a.year),
               sizeof(a.year) );
   bout.write( reinterpret_cast<const char*>(&a.month),
               sizeof(a.month) );
   bout.write( reinterpret_cast<const char*>(&a.day),
               sizeof(a.day) );
```

```
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::cannotWrite );
}; // Date_save
```

όπου `XXX_Xptn` κάποια κλάση εξαιρέσεων.

Γράψε πρόγραμμα που θα ζητάει από τον χρήστη μια ημερομηνία –ας την πούμε *limDate*– και διαβάζοντας το αρχείο θα δημιουργεί δύο νέα μορφοποιημένα (text) αρχεία στα οποία θα φυλάγει:

- στο ένα, με όνομα `odates.txt`, τις ημερομηνίες που είναι παλιότερες από τη *limDate* και
- στο άλλο, με όνομα `ndates.txt`, τις ημερομηνίες που είναι ίδιες με ή νεότερες από τη *limDate*.

Το πρόγραμμα θα υπολογίζει και θα μας λέει στο τέλος:

- το πλήθος των τιμών που έβαλε σε κάθε αρχείο,
- την πιο παλιά ημερομηνία που βρήκε και
- την πιο νέα ημερομηνία που βρήκε.

Γ Ομάδα

15-2 Ένας πίνακας που έχει πολλά στοιχεία του ίσα με 0 (μηδέν) λέγεται **αραιός** (sparse). Για έναν τέτοιο πίνακα μπορούμε, για οικονομία μνήμης, να μην φυλάγουμε όλα τα στοιχεία αλλά μόνον τα μη μηδενικά, το καθένα με τη θέση του. Ας πούμε ότι έχουμε έναν δισδιάστατο πίνακα τύπου `double` με *nR* γραμμές και *nC* στήλες. Για την αποθήκευσή του απαιτούνται

nR `*` *nC* `* sizeof(double)`

ψηφιολέξεις. Αν έχει *nNZ* μη μηδενικά στοιχεία και απόθηκεύσουμε το καθένα από αυτά σε ένα στοιχείο τύπου

```
struct ArrElmn
{
   unsigned int row;
   unsigned int col;
   double       value;
}; // ArrElmn
```

έχουμε οικονομία όταν

nNZ `* sizeof(`*ArrElmn*`) <` *nR* `*` *nC* `* sizeof(double)`

Σε ένα μη μορφοποιημένο αρχείο –με όνομα στον δίσκο `sprs017.dta`– έχουμε αποθηκευμένα τα στοιχεία ενός αραιού δισδιάστατου πίνακα ως εξής:

- Στην αρχή υπάρχει μια τιμή `unsigned int` που μας δίνει το πλήθος γραμμών (*nR*) του πίνακα.
- Ακολουθεί άλλη μια τιμή `unsigned int` που μας δίνει το πλήθος στηλών (*nC*) του πίνακα.
- Ακολουθούν *nR*`*`*nC* τιμές τύπου `double` που είναι οι τιμές των στοιχείων κατά γραμμές: πρώτα τα στοιχεία της γραμμής 0, μετά τα στοιχεία της γραμμής 1 κ.ο.κ.

Γράψε πρόγραμμα που θα αντιγράφει το αρχείο σε ένα άλλο, μη μορφοποιημένο, με όνομα στον δίσκο `sprs017cn.dta`, με το εξής περιεχόμενο:

- Στην αρχή θα υπάρχει μια τιμή `unsigned int`, το πλήθος γραμμών (*nR*) του πίνακα.
- Στη συνέχεια άλλη μια τιμή `unsigned int`, μας δίνει το πλήθος στηλών (*nC*) του πίνακα.
- Στη συνέχεια τιμές τύπου *ArrElmn*, μια για κάθε μη μηδενικό στοιχείο του αρχικού πίνακα.

project

1

Αυτοκίνητα στον Δρόμο

Περιεχόμενα:

Prj01.1 Το Πρόβλημα **477**
Prj01.2 Η Δομή Εξαιρέσεων **478**
Prj01.3 Η Συνάρτηση *openWrNoReplace()* **478**
Prj01.4 Η Συνάρτηση *openFiles()* **479**
Prj01.5 Η Συνάρτηση *copyTitle()* **481**
Prj01.6 Η Συνάρτηση *closeFiles()* **481**
Prj01.7 Η *ApplicXptn* (τελικώς) **482**
Prj01.8 Και η `main` **482**
Prj01.9 Η *openFiles()* Αλλιώς **483**

Prj01.1 Το Πρόβλημα

Θα ξαναλύσουμε το πρόβλημα που λύσαμε στην §13.11 αλλά αυτήν τη φορά με χρήση εξαιρέσεων. Για διευκόλυνσή σου ξαναδίνουμε το πρόβλημα:

> *Ένα συνεργείο έκανε μέτρηση ροής οχημάτων σε κάποιο δρόμο. Σε αρχείο, text – με το όνομα στο δίσκο* **`autoflow.txt`***– καταγράφηκαν οι τιμές ροής σε οχήματα/min κάθε λεπτό. Το πλήθος των τιμών στο αρχείο είναι άγνωστο, αλλά σίγουρα θετικό.*
>
> *Οι πρώτες τρεις γραμμές του αρχείου έχουν την «ταυτότητα» της μέτρησης ως εξής:*

```
Υπεύθυνος:\t<όνομα>\t<επώνυμο>
Σημείο Μετρήσεων:\t<οδός-αριθμός>\t<περιοχή>\t<δήμος>
Αρχή:\tdd.mm.yyyy\thh:mm
```

> *Στις υπόλοιπες γραμμές δίνονται οι τιμές από τη μέτρηση, μια σε κάθε γραμμή. Κάθε τιμή είναι γραμμένη στις πέντε πρώτες θέσεις. Πέρα από τη δέκατη θέση μπορεί να υπάρχουν σχόλια που περιγράφουν συμβάντα κατά τη διάρκεια της μέτρησης. Να ένα παράδειγμα:*

```
Υπεύθυνος:    Ανδρέας       Νικολόπουλος
Σημείο Μετρήσεων: Τζαβέλλα 48    Νεάπολις      Αθήνα
Αρχή:  15.06.2009    05:00
   26
   30
    8
   28
. . .
   17
```

```
   19
    4
   12      / Αρχή λειτουργίας σηματοδότησης
   28
   17
. . .
   17
   31      / Βλάβη σηματοδότη Τζαβέλλα & Γιωργάκου
   23
   24
. . .
```

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει:
1. Το πλήθος τιμών του αρχείου καθώς και τη διάρκεια των μετρήσεων σε h και min.
2. Το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων.
3. Τη μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min.
Όλα αυτά θα γράφονται στην τελική έκθεση, που θα γραφεί σε αρχείο με το όνομα `report.txt`. *Στις πρώτες τρεις γραμμές του αρχείου θα αντιγραφούν οι τρεις πρώτες γραμμές του* `autoflow.txt`.
Ένα από τα ζητούμενα της δουλειάς είναι και η μελέτη της μεταβολής της ροής οχημάτων. Ως πρώτο βήμα μας ζητείται να δημιουργήσουμε ένα άλλο αρχείο με τις μεταβολές ροής ανά min.

Prj01.2 Η Δομή Εξαιρέσεων

Στην αρχική λύση μεταφέραμε την πληροφορία «κάτι δεν πάει καλά» με παραμέτρους "`bool& ok`". Τώρα θα τη μεταφέρουμε με εξαιρέσεις. Θα ρίχνουμε εξαιρέσεις τύπου:

```
struct ApplicXptn
{
   enum { . . . };
   char funcName[100];
   int  errCode;
// . . .
   ApplicXptn( const char* fn, int ec, . . . )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
   // . . .
   }
}; // ApplicXptn
```

όπως τον είδαμε στην §15.10.

Θα συμπληρώνουμε τη δομή όπως θα (ξανα)γράφουμε το πρόγραμμά μας και θα αντιμετωπίζουμε τις ανάγκες του.

Προφανώς θα ξεκινήσουμε από τις συναρτήσεις που έχουν παράμετρο *ok*.

Prj01.3 Η Συνάρτηση *openWrNoReplace()*

Η πρώτη συνάρτηση που θα δούμε είναι η *openWrNoReplace()*. Τη γράφουμε ως:

```
void openWrNoReplace( ofstream& newStream, string flNm )
{
   ifstream test( flNm.c_str() );                        // άνοιξε για διάβασμα
   if ( !test.fail() )                                    // υπάρχει το αρχείο,
   {
      test.close();                            // κλείσε το και μην το πειράξεις
      throw ApplicXptn( "openWrNoReplace", ApplicXptn::fileExists,
```

```
                        flNm.c_str() );
   }
   newStream.open( flNm.c_str() );                          // δημιούργησέ το
   if ( newStream.fail() )
      throw ApplicXptn( "openWrNoReplace", ApplicXptn::cannotCreate,
                        flNm.c_str() );
} // openWrNoReplace
```

Πρόσεξε ότι *η λογική της συνάρτησης είναι ίδια με αυτήν της αρχικής*. Γράφουμε σε σχόλιο «*κλείσε το και μην το πειράξεις*». Αυτό πώς διασφαλίζεται; Με το ότι ακολουθεί εντολή **throw** και εκεί θα διακοπεί η εκτέλεση της συνάρτησης και η "`newStream.open( flNm.c_str() )`" δεν θα εκτελεσθεί!

Εδώ χρησιμοποιούμε δύο κωδικούς σφάλματος που πρέπει να τους ορίσουμε στη δομή εξαιρέσεων:

```
   enum { fileExists, cannotCreate, . . . };
```

Ακόμη προκύπτει ανάγκη για δημιουργό με τρίτη παράμετρο πίνακα χαρακτήρων (ορμαθό της C):

```
   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
```

Το *errStrVal* θα δηλωθεί ως μέλος της δομής:

```
   char errStrVal[100];
```

Prj01.4 Η Συνάρτηση *openFiles()*

Η *openWrNoReplace()* καλείται από την *openFiles()* που –και αυτή– έχει παράμετρο *ok*.

Ξαναδές στην §13.11.1 τη λογική της αρχικής συνάρτησης. Ανοίγουμε τα ρεύματα το ένα μετά το άλλο (*autoflow*, *differences*, *report*) εφόσον δεν υπήρξε πρόβλημα με το άνοιγμα των προηγουμένων. Και εδώ, αν γράψουμε:

```
   autoflow.open( autoFlNm.c_str() );
   if ( autoflow.fail() )
      throw ApplicXptn( "openFiles",
                        ApplicXptn::cannotOpen, autoFlNm.c_str() );
// . . .
```

πετυχαίνουμε το εξής: Αν δεν ανοίξει το *autoflow* ρίχνεται εξαίρεση και δεν προχωρούμε παρακάτω.

Το επόμενο ρεύμα (*differences*) ανοίγεται με την *openWrNoReplace()*. Τώρα, το «δεν άνοιξε το *differences*» σημαίνει ότι ρίχτηκε και μια εξαίρεση. Θα πρέπει να είμαστε έτοιμοι να την πιάσουμε:

```
   try
   {
      openWrNoReplace( differences, difFlNm );
   }
   catch( ApplicXptn& x )
   {
      autoflow.close();
      throw;
   } // catch
```

Τι κάνουμε εδώ; Αν πιάσουμε εξαίρεση κλείνουμε πρώτα το *autoflow* που είναι ήδη ανοικτό και ξαναρίχνουμε την εξαίρεση.

Παρομοίως θα μπορούσαμε να χειριστούμε και το άνοιγμα του *report*:

```
   try
   {
      openWrNoReplace( report, reprtFlNm );
   }
```

```
   catch( ApplicXptn& x )
   {
      autoflow.close();
      differences.close();
      throw;
   } // catch
```

Εδώ φυσικά θα πρέπει να κλείσουμε και το *differences*.

Αφού όμως η εξαίρεση κουβαλάει μαζί της και το όνομα του αρχείου που είχε το πρόβλημα (*x.errStrVal*), μπορούμε να γράψουμε:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm )
{
   autoflow.open( autoFlNm.c_str() );
   if ( autoflow.fail() )
      throw ApplicXptn( "openFiles",
                        ApplicXptn::cannotOpen, autoFlNm.c_str() );
// το autoflow ανοικτό
   try
   {
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
   }
   catch( ApplicXptn& x )
   {
      autoflow.close();
      if ( x.errStrVal == reprtFlNm )      // δεν άνοιξε το report
         differences.close();
      throw;
   } // catch
} // openFiles
```

Αν πάρουμε εξαίρεση από την *openWrNoReplace()* κλείνουμε οπωσδήποτε το *autoflow*. Αν προβληματικό αρχείο ήταν το `report.txt` τότε κλείνουμε και το ρεύμα *differences* που έχει ανοίξει πιο πριν.

Για να μην έχουμε τη συνάρτηση «κομμένη στα δύο», δεν θα μπορούσαμε να γράψουμε:

```
   try
   {
      autoflow.open( autoFlNm.c_str() );
      if ( autoflow.fail() )
         throw ApplicXptn( "openFiles",
                           ApplicXptn::cannotOpen,
                           autoFlNm.c_str() );
   // το autoflow ανοικτό
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
   }
   catch( ApplicXptn& x )
   { . . . }
```

Ναι, θα μπορούσαμε. Αλλά προσοχή: αν αποτύχει το άνοιγμα του *autoflow* η εξαίρεση θα συλληφθεί από την **catch** που έχουμε εδώ και η διαχείριση θα είναι πιο πολύπλοκη:

```
   catch( ApplicXptn& x )
   {
      if ( x.errStrVal == difFlNm )   // δεν άνοιξε το differences
         autoflow.close();
      else if ( x.errStrVal == reprtFlNm ) // δεν άνοιξε το report
      {
         autoflow.close();
         differences.close();
      }
      throw;
   } // catch
```

Αν το προτιμάς...

Από τη μετατροπή της *openFiles()* προκύπτει ανάγκη για έναν ακόμη κωδικό σφάλματος:

```
   enum { fileExists, cannotCreate, cannotOpen };
```

Prj01.5 Η Συνάρτηση *copyTitle()*

Στην *copyTitle()* η αντικατάσταση της *ok* με **throw** είναι απλή:

```
void copyTitle( ifstream& autoflow, ofstream& report )
{
   int  lineCount( 0 );   // μετρητής γραμμών
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
   if ( lineCount != 3 )
      throw ApplicXptn( "copyTitle", ApplicXptn::incomplete, lineCount );
} // copyTitle
```

Εδώ όμως, πέρα από την ανάγκη για νέο κωδικό σφάλματος (*incomplete*) προκύπτει η ανάγκη και για νέο δημιουργό εξαιρέσεων που να έχει τρίτη παράμετρο τύπου `int`:

```
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
```

Το *errIntVal* είναι μέλος της *ApplicXptn*:

```
   int  errIntVal;
```

Φυσικά, τώρα έχουμε:

```
   enum { fileExists, cannotCreate, cannotOpen, incomplete };
```

Prj01.6 Η Συνάρτηση *closeFiles()*

Η τελευταία συνάρτηση με παράμετρο *ok* είναι η *closeFiles()*. Την ξαναγράφουμε ως εξής:

```
void closeFiles( ifstream& autoflow, ofstream& differences, ofstream& report )
{
   autoflow.close();

   differences.close();
   if ( differences.fail() )
      throw ApplicXptn( "closeFiles", ApplicXptn::cannotClose, "διαφορές" );
   report.close();
   if ( report.fail() )
      throw ApplicXptn( "closeFiles", ApplicXptn::cannotClose, "έκθεση" );
} // closeFiles
```

Αυτή *δεν είναι λειτουργικώς ισοδύναμη με την αρχική*. Αν δεν μπορέσει να κλείσει το *differences* θα ρίξει εξαίρεση και δεν θα προσπαθήσει να κλείσει το *report*. Η αρχική θα προσπαθήσει να κλείσει και τα δύο. Δηλαδή είναι προτιμότερη η αρχική μορφή της συνάρτησης; Ναι, αν έχει νόημα το να πάρουμε το ένα μόνον αρχείο. Αν θέλουμε και τα δύο αρχεία θα πρέπει να ζητήσουμε να ξαναεκτελεσθεί το πρόγραμμα και στις δύο περιπτώσεις.

Πρόσεξε ακόμη ότι επειδή στη συνάρτηση αυτή δεν περνούν τα ονόματα των αρχείων *βάζουμε απλό κείμενο ως τρίτο όρισμα του δημιουργού*.

Prj01.7 Η *ApplicXptn* (τελικώς)

Ας δούμε τώρα πώς έγινε δομή εξαιρέσεων:

```
struct ApplicXptn
{
   enum { fileExists, cannotCreate, cannotOpen, incomplete, cannotClose };
   char funcName[100];
   int  errCode;
   char errStrVal[100];
   int  errIntVal;

   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   ApplicXptn( const char* fn, int ec, int iv )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;   errIntVal = iv;  }
}; // ApplicXptn
```

Prj01.8 Και η `main`

Τώρα, η `main` θα είναι ως εξής:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.dta
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.txt
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   try
   {
      openFiles( autoflow, autoFlNm, differences, difFlNm,
                 report, reprtFlNm );
      process( autoflow, differences, report );
      closeFiles( autoflow, differences, report );
      cout << "Τέλος καλό, όλα καλά..." << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::fileExists:
            cout << "από την " << x.funcName << ": το αρχείο "
                 << x.errStrVal << " υπάρχει" << endl;
            break;
         case ApplicXptn::cannotCreate:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να δημιουργήσω το " << x.errStrVal << endl;
            break;
         case ApplicXptn::cannotOpen:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να ανοίξω το " << x.errStrVal << endl;
            break;
         case ApplicXptn::incomplete:
            cout << "από την " << x.funcName
                 << ": ελλιπή δεδομένα. Διαβάστηκαν "
                 << x.errIntVal << " γραμμές " << endl;
            break;
         case ApplicXptn::cannotClose:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να κλείσω το αρχείο " << x.errStrVal << endl;
            break;
```

```
         default:
            cout << "unexpected ApplicXptn from " << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Η **main** απλουστεύθηκε σε εντυπωσιακό βαθμό:

```
      openFiles( autoflow, autoFlNm, differences, difFlNm,
                 report, reprtFlNm );
      process( autoflow, differences, report );
      closeFiles( autoflow, differences, report );
      cout << "Τέλος καλό, όλα καλά..." << endl;
```

Αλλα τίποτε δεν είναι δωρεάν στον κόσμο αυτόν: Έφυγαν οι "`if (ok)…`" και μας ήλθε αυτή η μακροσκελής **catch**!

Prj01.9 Η *openFiles()* Αλλιώς

Στην §14.9 γράφαμε: «*Θα διαχειριζόμαστε πάντοτε τις εξαιρέσεις στη* **main***; Όχι. Στη συνέχεια, θα δούμε πώς αποφασίζουμε που και πώς μπορεί να γίνει η καλύτερη διαχείριση της κάθε εξαίρεσης.*» Εδώ διαχειριζόμαστε όλες τις εξαιρέσεις στη **main** αλλά, αν το ξανασκεφτούμε, αυτή δεν είναι η καλύτερη λύση.

Ας πάρουμε τις εξαιρέσεις που ρίχνουμε αν δεν μπορέσουμε να ανοίξουμε κάποιο αρχείο ή κάποια αρχεία· δεν υπάρχει λόγος να τις αφήσουμε να φτάσουν μέχρι τη **main**. Εκεί, αρκεί να φτάσει τον μήνυμα «δεν ανοίγουν τα αρχεία». Καλύτερα λοιπόν να (ξανα)χρησιμοποιήσουμε την *ok* ως εξής:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm,
                bool& ok )
{
   ok = false;
   try
   {
      autoflow.open( autoFlNm.c_str() );
      if ( autoflow.fail() )
         throw ApplicXptn( "openFiles", ApplicXptn::cannotOpen,
                           autoFlNm.c_str() );
   // το autoflow ανοικτό
      openWrNoReplace( differences, difFlNm );
      openWrNoReplace( report, reprtFlNm );
      ok = true;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::fileExists:
         case ApplicXptn::cannotCreate:
            autoflow.close();
            if ( x.errStrVal == reprtFlNm )// δεν άνοιξε το report
               differences.close();
            cout << "από την " << x.funcName;
            if ( x.errCode == ApplicXptn::fileExists )
               cout << ": το αρχείο " << x.errStrVal << " υπάρχει" << endl;
            else // x.errCode == ApplicXptn::cannotCreate
               cout << ": δεν μπορώ να δημιουργήσω το " << x.errStrVal
                    << endl;
            break;
```

```
         case ApplicXptn::cannotOpen:
            cout << "από την " << x.funcName
                 << ": δεν μπορώ να ανοίξω το " << x.errStrVal << endl;
            break;
         default:
            throw;
      } // switch
   } // catch
} // openFiles
```

Αυτή η *openFiles()* πιάνει τις εξαιρέσεις που έχουν σχέση με το άνοιγμα των αρχείων: *ApplicXptn* με κωδικούς *fileExists*, *cannotCreate* και *cannotOpen* ενώ αφήνει να περνούν όλες οι άλλες.

Παρατήρηση: ▶

Στην §14.9 γράφαμε ακόμη: «*Οι εξαιρέσεις ρίχνονται μόνο από συναρτήσεις που καλούμε στην ομάδα* ***try****; Όχι! Αργότερα θα δεις παραδείγματα που θα έχουμε εντολές* `throw` *μέσα στην ομάδα της* ***try****.*» Εδώ έχουμε ένα τέτοιο παράδειγμα: περιμένουμε βέβαια εξαιρέσεις από τις δύο κλήσεις στην *openWrNoReplace()* αλλά υπάρχει και **throw** ακριβώς πάνω από αυτές.◀

Η αποτυχία στο άνοιγμα των αρχείων γνωστοποιείται προς τα έξω με τιμή `false` της *ok*. Το τι θα κάνει η **main** σε μια τέτοια περίπτωση είναι άλλη ιστορία. Για παράδειγμα, θα μπορούσε να γίνει κάτι σαν:

```
      do {
         openFiles( autoflow, autoFlNm, differences, difFlNm,
                    report, reprtFlNm, ok );
         if ( !ok )
            getFileNames( autoFlNm, difFlNm, reprtFlNm, quit );
      } while ( !ok && !quit );
      if ( ok )
      {
         process( autoflow, differences, report );
         closeFiles( autoflow, differences, report );
         cout << "Τέλος καλό, όλα καλά..." << endl;
      }
```

Η

```
void getFileNames( string& autoFlNm, string& difFlNm,
                   string& reprtFlNm, bool& quit )
```

(άσκηση για σένα) ζητάει από τον χρήστη να αλλάξει τα ονόματα (κάποιων) αρχείων ή να τα παρατήσει. Στην τελευταία περίπτωση η *quit* επιστρέφει τιμή `true`.

Με τον ίδιο τρόπο μπορούμε να χειριστούμε και την *closeFiles()*.

project

2

Διανύσματα στις 3 Διαστάσεις

Περιεχόμενα:

Prj02.1 Το Πρόβλημα **485**
Prj02.2 Ο Τύπος *Vector3* και οι Δημιουργοί **486**
Prj02.3 Οι Τελεστές Σύγκρισης **487**
Prj02.4 Οι Τελεστές "+", "-", "*", "^" **488**
Prj02.5 Ο Ενικός Τελεστής "-" **489**
Prj02.6 Οι Τελεστές Εκχώρησης **489**
Prj02.7 Ο Τελεστής "<<" **490**
Prj02.8 ... και το Ευκλείδιο Μέτρο **490**
Prj02.9 Το Πρόγραμμα **490**

Prj02.1 Το Πρόβλημα

Το Project αυτό είναι ένα μεγάλο παράδειγμα επιφόρτωσης τελεστών. Είναι καλό να το διαβάσεις προσεκτικά διότι *αργότερα θα το ξαναδούμε και θα αλλάξουμε μερικά από αυτά πόυ θα δούμε εδώ.*

Ένα τρισδιάστατο (ελεύθερο) ***διάνυσμα*** *(vector) είναι μια διαταγμένη τριάδα πραγματικών αριθμών (x, y, z).*

Όρισε έναν τύπο δομής Vector3 που τα αντικείμενά της θα είναι διανύσματα του χώρου τριών διαστάσεων. Όρισε έναν ερήμην δημιουργό που θα μας δίνει ελεύθερο διάνυσμα με συνιστώσες (0, 0, 0) αλλά θα μπορεί να πάρει και αρχικές τιμές των συνιστωσών.

Α. Έστω ότι έχουμε δύο διανύσματα $\mathbf{v}_1 = (x_1, y_1, z_1)$ *και* $\mathbf{v}_2 = (x_2, y_2, z_2)$.

1. Τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *μπορεί να συγκριθούν για ισότητα και μόνον (δηλαδή έχουν νόημα μόνον οι "==" και "!="). Είναι ίσα αν και μόνον αν* $x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2$. *Επιφόρτωσε τους τελεστές σύγκρισης "==" και "!=" για τον Vector3.*

2. Μπορούμε ακόμη να τα προσθέσουμε ή τα αφαιρέσουμε και να πάρουμε ένα διάνυσμα $\mathbf{v}_1 \pm \mathbf{v}_2$ *με συνιστώσες* $(x_1 \pm x_2, y_1 \pm y_2, z_1 \pm z2)$.
Επιφόρτωσε τους τελεστές "+" και "-" για τον Vector3.
Επιφόρτωσε τους τελεστές "+=" και "-=" για τον Vector3.

3. Πολλαπλασιάζοντας ένα διάνυσμα $\mathbf{v}_1$ *επί έναν πραγματικό αριθμό α (*$\alpha\mathbf{v}_1$ *ή* $\mathbf{v}_1\alpha$*) παίρνουμε ένα νέο διάνυσμα με συνιστώσες* $(\alpha x_1, \alpha y_1, \alpha z_1)$.
Επιφόρτωσε καταλλήλως τον τελεστή "" για τον Vector3.*
Επιφόρτωσε τον τελεστή "=" για τον Vector3.*

4. Ειδική περίπτωση: το $(-1)\mathbf{v}_1$ *μπορεί να γραφεί και ως* $-\mathbf{v}_1$.
Επιφόρτωσε καταλλήλως τον ενικό τελεστή "-" για τον Vector3.

5. Μπορούμε να πολλαπλασιάσουμε τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *και να πάρουμε τον πραγματικό αριθμό:* $\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1x_2 + y_1y_2 + z_1z_2$ *που λέγεται* ***εσωτερικό γινόμενο*** *(inner ή dot product).*
*Επιφόρτωσε (άλλη μια φορά) καταλλήλως τον τελεστή "*****" για τον Vector3.*
6. Μπορούμε όμως να πάρουμε και το ***εξωτερικό γινόμενο*** *(outer ή cross product)* $\mathbf{v}_1 \times \mathbf{v}_2$ *που είναι διάνυσμα:* $\mathbf{v}_1 \times \mathbf{v}_2 = (y_1z_2-z_1y_2,\ x_2z_1-z_2x_1,\ y_2x_1-x_2y_1)$.
*Επιφόρτωσε καταλλήλως τον τελεστή "**^**"για τον Vector3 ώστε να υπολογίζει το εξωτερικό γινόμενο.*
*7. Καλό θα είναι να μπορούμε να γράψουμε στην οθόνη ή σε ένα αρχείο text την τιμή του διανύσματος με τον "**<<**".*
*Επιφόρτωσε τον "**<<**" για αντικείμενα τύπου Vector3.*

8. Τέλος, για κάθε διάνυσμα ορίζεται ένα ευκλείδιο μέτρο: $|\mathbf{v}| = \sqrt{x^2+y^2+z^2}$ *(=* $\sqrt{\mathbf{v}\cdot\mathbf{v}}$ *) που είναι το μήκος του διανύσματος.*
Γράψε συνάρτηση Vector3_abs που να δίνει το ευκλείδιο μέτρο ενός αντικειμένου τύπου Vector3.
Β. Για μια απλή εφαρμογή των παραπάνω, γράψε πρόγραμμα που θα υπολογίζει τη δύναμη Laplace $\mathbf{F}_L = q(\mathbf{v} \times \mathbf{B})$ *που ασκείται σε φορτίο* $q = 1\ \mu Cb$ *που εισέρχεται με ταχύτητα* $\mathbf{v} = (10^8, 0, 0)$ *σε μαγνητικό πεδίο* $\mathbf{B} = (0, 0, 10)$. *Επιβεβαίωσε ότι η δύναμη είναι κάθετη στα* $\mathbf{v}$ *και* $\mathbf{B}$ *(δηλαδή:* $\mathbf{F}_L \cdot \mathbf{v} = 0$ *και* $\mathbf{F}_L \cdot \mathbf{B} = 0$*).*
Το εμβαδόν παραλληλογράμμου που οι πλευρές του είναι παράλληλες προς τα ελεύθερα διανύσματα $\mathbf{a}$ *και* $\mathbf{b}$ *είναι το διάνυσμα* $\mathbf{S} = \mathbf{a} \times \mathbf{b}$. *Αν έχουμε το παραλληλόγραμμο μέσα σε ένα μαγνητικό πεδίο τότε η μαγνητική ροή που διέρχεται από αυτό είναι ίση με* $\mathbf{B} \cdot \mathbf{S}$. *Συμπλήρωσε το πρόγραμμά σου με εντολές που θα υπολογίζουν τη μαγνητική ροή που διέρχεται από παραλληλόγραμμο με πλευρές* $\mathbf{a} = (0.1, 0.2, 0.3)$ *και* $\mathbf{b} = (0.2, 0.3, 0.4)$ *όταν βρίσκεται στο μαγνητικό πεδίο που δώσαμε παραπάνω.*
Με τα $\mathbf{a}$ *και* $\mathbf{b}$ *επιβεβαίωσε την ταυτότητα:* $|\mathbf{a}|^2|\mathbf{b}|^2 = (\mathbf{a}\cdot\mathbf{b})^2 + |\mathbf{a}\times\mathbf{b}|^2$.

Prj02.2 Ο Τύπος *Vector3* και οι Δημιουργοί

Ο τύπος αυτός είναι σαν τον *complex* αλλά με τρία μέλη:

```
struct Vector3
{
   double x;
   double y;
   double z;
}; // Vector3
```

Γράφουμε τον ερήμην δημιουργό με αρχικές τιμές όπως κάναμε και στην *complex*:

```
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
```

Πρόσεξε ότι με αυτόν τον δημιουργό μπορούμε να κάνουμε τις εξής δηλώσεις:

```
   Vector3 v0;                                 // (0, 0, 0)
   Vector3 v1( 1 );                            // (1, 0, 0)
   Vector3 v2( sqrt(2), 3 );                   // (1.41421, 3, 0)
   Vector3 v3( sqrt(3), sqrt(5), 2*sqrt(2) ); // (1.73205, 2.23607, 2.82843)
```

Να λοιπόν ο ορισμός του τύπου μας:

```
struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
```

```
   {  x = ax;  y = ay;  z = az;  }
}; // Vector3
```

Prj02.3 Οι Τελεστές Σύγκρισης

Σύμφωνα με όσα είπαμε στην §14.6.4, ένας δυαδικός τελεστής ("`@`") επιφορτώνεται με μια συνάρτηση της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Για έναν τελεστή σύγκρισης, όπως είναι ο "`==`", *Trv* είναι ο `bool`. Για την περίπτωσή μας *Tl* και *Tr* είναι, κατ' αρχήν, ο *Vector3*. Θα μπορούσαμε λοιπόν να γράψουμε:

```
bool operator==( Vector3 lhs, Vector3 rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Αυτή είναι απολύτως σωστή, αλλά, παρ' όλα αυτά, θα την αλλάξουμε λιγάκι, ως προς τις παραμέτρους:

```
bool operator==( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Ποιο είναι το πλεονέκτημα της δεύτερης μορφής; Κάθε φορά που την καλούμε θα περάσουν, ως παράμετροι, δύο βέλη ενώ στην πρώτη μορφή θα περάσουν δυο τιμές *Vector3*. Σε ψηφιολέξεις αυτό θα μπορούσε να σημαίνει (ενδεικτικώς) 8:48· εξαπλάσιο! Ναι, αλλά το 48 είναι πολύ μικρό για να μας δημιουργήσει πρόβλημα. Υπάρχουν όμως και περιπτώσεις αντικειμένων που μπορεί να είναι πολύ μεγάλα· σκέψου, για παράδειγμα, ένα αντικείμενο τύπου *string* με τιμή το κείμενο ενός βιβλίου 1000 σελίδων. *Τα πολύ μεγάλα αντικείμενα δεν τα περνάμε ως παραμέτρους τιμής αλλά ως παραμέτρους αναφοράς με "`const`"*. Θα χρησιμοποιούμε λοιπόν αυτόν τον τρόπο παντού στις επιφορτώσεις τελεστών για να τον συνηθίσεις(!)

Ας έλθουμε τώρα στον "`!=`"· χρειάζεται να τον γράψουμε; Αφού έχουμε τον "`==`" θα μπορούμε να γράφουμε `!(a == b)` αντί για `a != b`. Φυσικά, αλλά γενικώς θα τηρούμε την εξής προγραμματιστική πρακτική:[1]

- ***Όταν επιφορτώνουμε έναν τελεστή σύγκρισης "@" επιφορτώνουμε και τον αντίθετό του με χρήση του "@".***

Έχουμε λοιπόν:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return !(lhs == rhs);
} // operator!=( Vector3
```

Έτσι, έχουμε και τους δύο τελεστές χωρίς ασυμβατότητες (σίγουρα). Αν θέλεις μπορείς να πας ένα βήμα παραπέρα και να αποφύγεις μια κλήση συνάρτησης:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x != rhs.x) ||
          (lhs.y != rhs.y) || (lhs.z != rhs.z);
} // operator!=( Vector3
```

Για τελεστές που αυτό το τελευταίο βήμα είναι πιο πολύπλοκο –και υπάρχει μεγάλη πιθανότητα λάθους– μην το κάνεις.

---

[1] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposite (such as "==" and "!="), it is appropriate to define both.*»

Prj02.4 Οι Τελεστές "+", "-", "*", "^"

Οι δύο προσθετικοί (δυαδικοί) τελεστές και ο "^" (εξωτερικό γινόμενο) θα επιφορτωθούν με συναρτήσεις της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Γι ά τις πράξεις αυτές έχουμε:

+: *Vector3* × *Vector3* ↛ *Vector3*

-: *Vector3* × *Vector3* ↛ *Vector3*

^: *Vector3* × *Vector3* ↛ *Vector3*

Τις θεωρούμε μερικές συναρτήσεις επειδή παίρνουμε υπόψη την περίπτωση υπερχείλησης. Στη συνέχεια, στην υλοποίηση, θα τις χειριστούμε σαν ολικές συναρτήσεις· δηλαδή δεν θα ρίχνουμε εξαιρέσεις.

Στις περιπτώσεις αυτές *Trv*, *Tl* και *Tr* είναι ο *Vector3*. Σύμφωνα όμως με αυτά που είπαμε παραπάνω, θα βάλουμε τους *Tl* και *Tr* `const Vector3&`:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x + rhs.x;
   fv.y = lhs.y + rhs.y;
   fv.z = lhs.z + rhs.z;
   return fv;
} // operator+( const Vector3&

Vector3 operator-( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x - rhs.x;
   fv.y = lhs.y - rhs.y;
   fv.z = lhs.z - rhs.z;
   return fv;
} // operator-( const Vector3&

Vector3 operator^( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.y*rhs.z - lhs.z*rhs.y;
   fv.y = rhs.x*lhs.z - rhs.z*lhs.x;
   fv.z = rhs.y*lhs.x - rhs.x*lhs.y;
   return fv;
} // operator^( const Vector3
```

Σημείωση:▶

Υπάρχει δυαδικός τελεστής "^" στη C++; Ναι, αλλά θα τον μάθουμε αργότερα. Στην §14.6 λέγαμε «*Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.*» Αυτό το σεβόμαστε με την επιφόρτωση που κάνουμε; Όχι, αλλά όπως θα καταλάβεις δεν δημιουργεί οποιαδήποτε σύγχυση.◀

Ο "*" θέλει να σκεφτούμε κάτι παραπάνω. Αν έχουμε δηλώσει:

```
Vector3 v1, v2;
double  a;
```

θα πρέπει να μπορούμε να γράψουμε είτε:

```
v2 = a*v1;
```

είτε:

```
v2 = v1*a;
```

Έχουμε δηλαδή δύο περιπτώσεις:

*: `double` × *Vector3* ↛ *Vector3*

*: *Vector3* × `double` ↛ *Vector3*

Θα κάνουμε λοιπόν διπλή επιφόρτωση του "*":

```
Vector3 operator*( double lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs * rhs.x;
   fv.y = lhs * rhs.y;
   fv.z = lhs * rhs.z;
   return fv;
} // operator*

Vector3 operator*( const Vector3& lhs, double rhs )
{
   Vector3 fv;
   fv.x = rhs * lhs.x;
   fv.y = rhs * lhs.y;
   fv.z = rhs * lhs.z;
   return fv;
} // operator*
```

Ο "`*`" πρέπει να επιφορτωθεί άλλη μια φορά για το εσωτερικό γινόμενο:

`*`: *Vector3* × *Vector3* ↛ `double`

που υλοποιείται με την

```
double operator*( const Vector3& lhs, const Vector3& rhs )
{
   return lhs.x*rhs.x + lhs.y*rhs.y + lhs.z*rhs.z;
} // operator*( const Vector3&
```

Prj02.5 Ο Ενικός Τελεστής "-"

Αν έχουμε δηλώσει:

```
   Vector3 v1, v2;
```

και δώσουμε:

```
   v2 = -v1;
```

το διάνυσμα $\mathbf{v}_2$ είναι το αντίθετο του $\mathbf{v}_1$, δηλαδή: $\mathbf{v}_1 + \mathbf{v}_2 = \mathbf{0}$.

Για τον "`-`" έχουμε:

`-`: *Vector3* → *Vector3*

Στην §14.6.4 λέγαμε ότι επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή "`@`" με μια συνάρτηση

```
   Trv operator@( T rhs )
```

και είδαμε ήδη μια εφαρμογή αυτού του κανόνα στον "`-`" για τον *complex* (§15.5). Τώρα, *Trv* και *T* είναι ο *Vector3*:

```
Vector3 operator-( const Vector3& rhs )
{
   return Vector3( -rhs.x, -rhs.y, -rhs.z );
} // operator-( const Vector3&
```

Prj02.6 Οι Τελεστές Εκχώρησης[2]

Τώρα θα επιφορτώσουμε τους τρεις τελεστές εκχώρησης "`+=`", "`-=`", "`*=`" όπως είπαμε στην §14.6.3. Εκεί είδαμε ότι ο "+=" για τον τύπο *T* επιφορτώνεται ως:

```
T& operator+=( T& lhs, const T& rhs )
```

Προσεξε ότι ο τύπος της πρώτης παραμέτρου δεν έχει `const`.

Στην περίπτωσή μας *T* είναι ο *Vector3* οπότε έχουμε:

```
Vector3& operator+=( Vector3& lhs, const Vector3& rhs )
{
```

[2] Αργότερα θα μάθουμε ότι ο πάγιος τρόπος επιφόρτωσης αυτών των τελεστών είναι διαφορετικός.

```
   lhs.x += rhs.x;
   lhs.y += rhs.y;
   lhs.z += rhs.z;
   return lhs;
} // operator+=( Vector3&
```

Παρομοίως γίνεται η επιφόρτωση και των άλλων δύο τελεστών:

```
Vector3& operator-=( Vector3& lhs, const Vector3& rhs )
{
   lhs.x -= rhs.x;
   lhs.y -= rhs.y;
   lhs.z -= rhs.z;
   return lhs;
} // operator-=( const Vector3&

Vector3& operator*=( Vector3& lhs, double rhs )
{
   lhs.x *= rhs;
   lhs.y *= rhs;
   lhs.z *= rhs;
   return lhs;
} // operator*=( const Vector3&
```

Prj02.7 Ο Τελεστής "<<"

Έχουμε ήδη επιφορτώσει τον "<<" για αρκετους τύπους. Αντιγράφοντας σχεδόν την επιφόρτωση για τον τύπο *complex* (§15.5) έχουμε:

```
ostream& operator<<( ostream& tout, const Vector3& rhs )
{
   return tout << "(" << rhs.x << ", " << rhs.y << ", "
               << rhs.z << ")";
} // operator<<
```

Prj02.8 ... και το Ευκλείδιο Μέτρο

Για το ευκλείδιο μέτρο δεν έχουμε κάποιον βολικό (από οπτική άποψη) τελεστή. Θα γράψουμε λοιπόν μια:

```
double Vector3_abs( const Vector3& lhs )
{
   return sqrt( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs
```

Prj02.9 Το Πρόγραμμα

Έχοντας αυτά τα εργαλεία το πρόγραμμα είναι τετριμμένο. Για να διευκολύνουμε το γράψιμο (και για να κάνουμε οικονομία στις πράξεις) της τελευταίας ερώτησης ορίζουμε μια επιπλέον συνάρτηση:

```
double Vector3_abs2( const Vector3& lhs )
{
   return ( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs2
```

που για ένα διάνυσμα **a** μας δίνει το $|\mathbf{a}|^2$.

Αυτό που περιμένουμε να δούμε είναι ο μηδενισμός της παράστασης:

```
Vector3_abs2(a)*Vector3_abs2(b) - ((a*b)*(a*b)+Vector3_abs2(a^b))
```

Γράφουμε λοιπόν το πρόγραμμα:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
   Vector3( const Vector3& rhs  )
   {  x = rhs.x;  y = rhs.y;  z = rhs.z;  }
}; // Vector3

bool operator==( const Vector3& lhs, const Vector3& rhs );
bool operator!=( const Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& rhs );
Vector3 operator+( const Vector3& lhs, const Vector3& rhs );
Vector3& operator+=( Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& lhs, const Vector3& rhs );
Vector3& operator-=( Vector3& lhs, const Vector3& rhs );
Vector3 operator*( double lhs, const Vector3& rhs );
Vector3 operator*( const Vector3& lhs, double rhs );
Vector3& operator*=( Vector3& lhs, double rhs );
double operator*( const Vector3& lhs, const Vector3& rhs );
Vector3 operator^( const Vector3& lhs, const Vector3& rhs );
ostream& operator<<( ostream& tout, const Vector3& rhs );
double Vector3_abs( const Vector3& lhs );
double Vector3_abs2( const Vector3& lhs );

int main()
{
   double q( 1e-6 ); // Cb
   Vector3 v( 1e8, 0, 0 );
   Vector3 B( 0, 0, 10 );
   Vector3 FL;

   FL = q*( v ^ B );
   cout << FL << endl;
   cout << FL*v << " " << FL*B << endl;

   Vector3 a( 0.1, 0.2, 0.3 ), b( 0.2, 0.3, 0.4 );

   cout << a << "  " << b << endl;
   cout << B*( a ^ b) << endl;

   cout << Vector3_abs2(a)*Vector3_abs2(b) -
           ((a*b)*(a*b)+Vector3_abs2(a^b)) << endl;
   cout << Vector3_abs2(a)*Vector3_abs2(b) << endl;
} // main
```

Αποτέλεσμα:

```
(0, -1000, 0)
0 0
(0.1, 0.2, 0.3)  (0.2, 0.3, 0.4)
-0.1
-3.27294e-018
0.0406
```

Οι τελευταίες δύο γραμμές χρειάζονται ένα σχόλιο. Η διαφορά που περιμένουμε μηδέν βγαίνει περίπου $-3.3 \cdot 10^{-18}$. Αλλά το γινόμενο $|\mathbf{a}|^2|\mathbf{b}|^2$ έχει τιμή $0.0406 \approx 4 \cdot 10^{-2}$. Όπως βλέπεις, κατ' απόλυτη τιμή, η διαφορά είναι 10^{16} φορές μικρότερη. Μπορούμε λοιπόν να τη θεωρήσουμε μηδέν (0).

κεφάλαιο

16

Δυναμική Παραχώρηση Μνήμης

Ο στόχος μας σε αυτό το κεφάλαιο:

Να γνωρίσεις το σύστημα διαχείρισης δυναμικής μνήμης της C++. Το σύστημα αυτό είναι εξαιρετικώς ευέλικτο και δίνει τη δυνατότητα στον προγραμματιστή να χρησιμοποιεί τη δυναμική μνήμη –με πίνακες ή δυναμικές δομές δεδομένων– με πολύ αποδοτικό τρόπο. Το βέλος είναι το βασικό εργαλείο.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς δυναμικούς πίνακες και δυναμικές δομές δεδομένων στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *τελεστές "`new`", "`new[]`"*
- *τελεστές "`delete`", "`delete[]`"*
- *δυναμική μεταβλητή*
- *δυναμικός πίνακας*
- *μεταβλητή-βέλος (pointer)*
- *RAII*

Περιεχόμενα:

16.1 Οι Τελεστές "`new`" και "`delete`" 494
16.2 Συντακτικά και Βασικές Έννοιες 496
16.3 Τιμές Βελών και Δυναμικών Μεταβλητών 498
16.4 Δυναμικοί Πίνακες 500
16.5 * Η Τρίτη Μορφή του "`new`" 504
16.6 Η Εξαίρεση *bad_alloc* 504
16.6.1 Μια Εξήγηση για τις Εξαιρέσεις μας 505
16.7 Τα Προβλήματα της Δυναμικής Μνήμης 505
16.7.1 RAII: Μια Καλύτερη Λύση 509
16.8 Προβλήματα και στις Δομές 511
16.9 Δισδιάστατοι Δυναμικοί Πίνακες 511
16.10 * Τύπος Βέλους: "`void*`" 515
16.11 * Αναμνήσεις από τη C: *malloc()*, *free()*, *realloc()* 517
16.12 Για να Μη Ζηλεύουμε τη *realloc()* 519
16.13 Παραδείγματα 520
16.13.1 Το Περίγραμμα *linSearch()* 532
16.13.2 Χωρίς τη *linSearch()* 533
16.13.3 "*reserved* + *incr*" ή "2reserved*" 535**
16.14 Προβλήματα Ασφάλειας 535
16.15 Ανακεφαλαίωση 537
Ασκήσεις 538

Α Ομάδα.....538

Εισαγωγικές Παρατηρήσεις:

Όπως λέγαμε στην §11.1 «*σε κάθε πρόγραμμα παραχωρούνται τρεις περιοχές μνήμης:*

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

Η ποσότητα *στατικής μνήμης* που θα χρησιμοποιηθεί καθορίζεται όταν γράφουμε το πρόγραμμά μας, όταν δηλώνουμε τις καθολικές και τις στατικές μεταβλητές που χρησιμοποιεί.

Η ποσότητα *μνήμης στοίβας* που θα χρησιμοποιηθεί εξαρτάται

- από τις μεταβλητές που δηλώνουμε στις διάφορες συναρτήσεις και
- από τον τρόπο που θα κληθούν οι συναρτήσεις και πόσες φορές.

Αν, ας πούμε, έχουμε δύο συναρτήσεις *f()* και *g()* και τις καλέσουμε από τη `main` με τις εντολές:

```
f( . . . );
. . .
g( . . . );
```

τότε η μέγιστη ποσότητα αυτόματης μνήμης που θα απαιτηθεί είναι αυτή της `main` και η μέγιστη από τις απαιτούμενες για τις *f()* και *g()*. Αν όμως η `main` καλεί την *f()* και αυτή καλεί τη *g()*, τότε θα χρειαστούμε μνήμη για τη `main` και μνήμη για την *f()* και μνήμη για τη *g()* ταυτοχρόνως.

Αν η *f()* είναι αναδρομική και καλέσει *n* φορές τον εαυτόν της τότε θα χρειαστούμε μνήμη για τη `main` και *n*+1 φορές μνήμη για την *f()*.

Η C++, όπως και άλλες γλώσσες προγραμματισμού, μας επιτρέπει «να παίρνουμε» πρόσθετη μνήμη –τη **δυναμική** μνήμη– σύμφωνα με ανάγκες που παρουσιάζονται *όταν το πρόγραμμά μας εκτελείται*. Επιτρέπει δηλαδή **δυναμική παραχώρηση μνήμης** (dynamic memory allocation).

Το πρότυπο της C++ αναφέρεται στη δυναμική μνήμη με δύο διαφορετικά ονόματα: Αποκαλεί

- **μνήμη σωρού** (heap memory) αυτήν που χειριζόμαστε με τις συναρτήσεις της C (*malloc*, *calloc*, *realloc*, *free*) και
- **free store** αυτήν που χειριζόμαστε με τους τελεστές `new` και `delete` της C++.

Έτσι, αν κάνεις ένα πείραμα σαν αυτό της §14.8, με αυτά που θα δούμε στο κεφάλαιο αυτό, μπορεί να δεις διαφορετικές περιοχές διευθύνσεων. Αυτό δεν έχει και μεγάλη σημασία αρκεί να τηρείς τον πολύ βασικό κανόνα, που θα επαναλάβουμε και στη συνέχεια: *Μνήμη που παίρνεις με* `new` *θα απελευθερώνεται με* `delete` *και μνήμη που παίρνεις με malloc, calloc, realloc θα απελευθερώνεται με free.*

Θα πρέπει πάντως να επισημάνουμε μια διαφορά της C++ από τις «αδελφές» της απογόνους της C, τη Java και τη C#: Η C++ κράτησε τον τρόπο χειρισμού της δυναμικής μνήμης με *μεταβλητές-βέλη* (pointers) αν και έχει (και) νέα εργαλεία πέρα από αυτά της C.

16.1 Οι Τελεστές "new" και "delete"

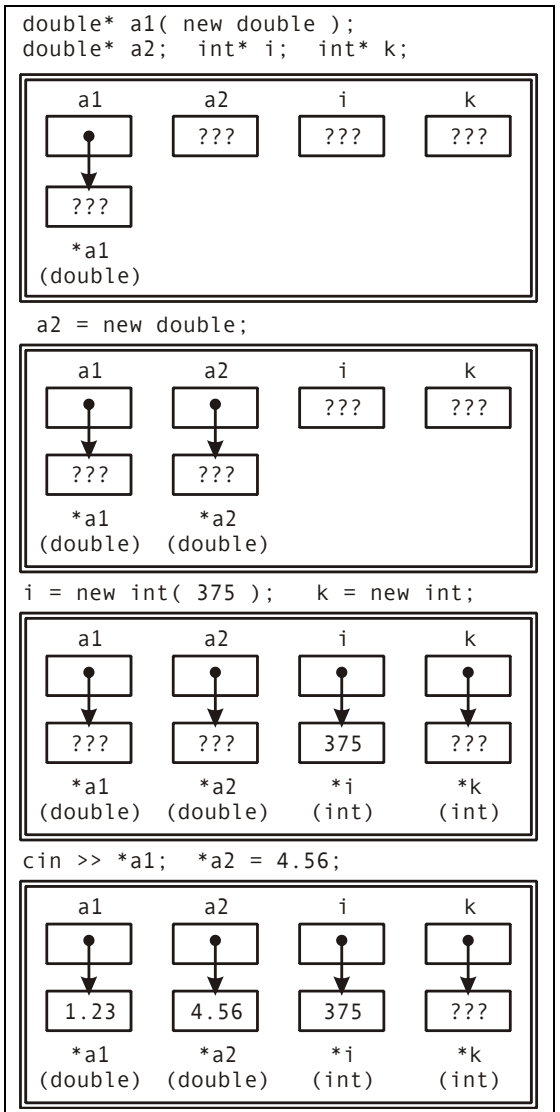
Η δυναμική παραχώρηση μνήμης γίνεται με τους τελεστές "`new`" και "`delete`" και με δύο κατηγορίες μεταβλητών:

- τις **μεταβλητές-βέλη** (pointer variables) –που ήδη ξέρουμε– και

- τις **δυναμικές** (dynamic) **μεταβλητές**.

Στις τιμές-βέλη αναφερθήκαμε για πρώτη φορά στο Κεφ. 2 (§2.8.3) λέγαμε ότι: «*γράφοντας* **&number** *παίρνουμε τη διεύθυνση μιας θέσης μνήμης όπου υπάρχει η πληροφορία που θέλουμε· παίρνουμε δηλαδή μια παραπομπή προς αυτό που μας ενδιαφέρει. Λέμε λοιπόν ότι η* **&number** *είναι μια* ***παραπέμπουσα*** *ή* ***αναφερόμενη*** (*referencing*) *τιμή –αφού παραπέμπει ή αναφέρεται σε κάτι– ή τιμή-**βέλος*** (*pointer*) *– αφού δείχνει κάτι.*» Στην ίδια παράγραφο λέγαμε ακόμη: «*Ας πούμε ότι έχουμε μια τιμή-βέλος p, δηλαδή μια διεύθυνση· πώς μπορούμε να δούμε την τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p; Ή, με άλλα λόγια, ποια είναι αντίστροφη πράξη της "&"; Η C++ τη συμβολίζει με "*****": η τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p παριστάνεται με "***p**". [...] Αν πάρουμε την* ***(&number)** *είναι σαν να παίρνουμε τη* **number**. *Λέμε ότι ο τελεστής "*****" **απο-παραμέμπει*** (*dereferences*) *την τιμή-βέλος στην οποία δρα.*»

Στο Κεφ. 12 είδαμε ότι «*Για τη C++, ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.*» Και (ενώ ένας πίνακας είναι ένα σταθερό βέλος) είδαμε ότι μπορούμε να έχουμε και μεταβλητές-βέλη. Πάντως, σε όλες τις περιπτώσεις οι τιμές και οι μεταβλητές-βέλη έδειχναν θέσεις συμβατικής (στατικής ή αυτόματης) μνήμης.



Σχ. 16-1 Δημιουργία δυναμικών μεταβλητών και εκχώρηση τιμών σε αυτές. Η δυναμική μεταβλητή δεν υπάρχει όταν το αντίστοιχο βέλος δεν έχει τιμή.

Τώρα ας δούμε μια άλλη δυνατότητα. Αν έχουμε δηλώσει:

```
T*   q;  ή  (T  *q;)
```

αν δηλαδή η *q* είναι μια *μεταβλητή-βέλος*, που δείχνει θέσεις μνήμης τύπου *T*, τότε η εκτέλεση της εντολής

```
q = new T;
```

έχει τα εξής αποτελέσματα:

- Παραχωρείται στο πρόγραμμά μας μια θέση μνήμης τύπου *T*. Αυτή είναι μια *δυναμική μεταβλητή* και μπορούμε να την χρησιμοποιούμε στο πρόγραμμά μας με το «όνομα» ***q**.
- Η *διεύθυνση* της ***q** είναι αποθηκευμένη ως τιμή της *q*.

Αν, αργότερα, δεν χρειαζόμαστε αυτή τη θέση, την «αναλυκλώνουμε» ή την «επιστρέφουμε» με την

```
delete q;
```

Πριν δώσουμε παράδειγματα όπου θα φαίνεται η χρησιμότητα της δυναμικής παραχώρησης μνήμης, θα εισαγάγουμε τα στοιχεία της γλώσσας που απαιτούνται με πιο απλά παραδείγματα.

Πλαίσιο 16.1

Τελεστής new

Αν έχουμε δηλώσει:

```
T*  pv;
```

1) η

```
pv = new T;
```

έχει ως αποτέλεσμα να παραχωρηθεί στο πρόγραμμα μια θέση μνήμης (μεταβλητή) τύπου *T*. Η μεταβλητή είναι η

```
*pv
```

και μπορούμε να τη διαχειριζόμαστε όπως οποιαδήποτε μεταβλητή τύπου *T*. Η διεύθυνση της θέσης που μας παραχωρήθηκε αποθηκεύεται ως τιμή της *pv*.

2) η

```
pv = new T[N];
```

έχει ως αποτέλεσμα να παραχωρηθούν στο πρόγραμμα *N* θέσεις μνήμης τύπου *T*, δηλαδή ένας *πίνακας* με *N* στοιχεία τύπου *T*. Τα στοιχεία του πίνακα είναι:

```
pv[0], pv[1],… pv[N-1]
```

και μπορούμε να τα διαχειριζόμαστε σαν να είχαμε δηλώσει στο πρόγραμμα:

```
T pv[N];
```

16.2 Συντακτικά και Βασικές Έννοιες

Ας πούμε ότι έχουμε κάποιον τύπο *T* και δηλώνουμε τη μεταβλητή:

```
T*  q;
```

Ή, αν θέλεις, ορίζουμε έναν τύπο:

```
typedef  T*   PT;
```

και δηλώνουμε:

```
PT  q;
```

Σε κάθε περίπτωση, η *q* είναι μια *μεταβλητή-βέλος*. Ο *T** ή ο *PT* είναι ένας **τύπος βέλους** με **τύπο στόχο** (target ή domain ή base type) τον *T*.

Όπως ξέρουμε *στην q αποθηκεύονται πληροφορίες που καθορίζουν μια διεύθυνση στην κύρια μνήμη του υπολογιστή*, συνήθως η ίδια η διεύθυνση. Στη διεύθυνση αυτήν, που μας δείχνει η τιμή της *q*, μπορεί να αποθηκευτεί μια τιμή τύπου *T*.

Για παράδειγμα, μετά τα

```
typedef  int*   PInt;
// . . .
   double *a1, *a2;
   PInt   i, j, k;
```

Οι *a*1, *a*2, *i*, *j*, *k* είναι μεταβλητές-βέλη· οι δύο πρώτες προς μεταβλητές τύπου `char`, οι τρεις τελευταίες προς μεταβλητές τύπου `int`.

Παρατήρηση: ▶

Πρόσεξε το εξής: Μετά τον ορισμό «*PInt* είναι ο `int*`», με τη δήλωση που κάνουμε, όλες οι μεταβλητές –*i*, *j*, *k* – είναι τύπου `int*`. Στην πρώτη δήλωση, αν γράφαμε "`double* a1, a2`" η *a*1 θα ήταν τύπου `double*` αλλά η *a*2 θα ήταν τύπου `double`.◀

Θα χρησιμοποιήσουμε αυτές τις (συμβατικές) μεταβλητές για να πάρουμε και να ελέγξουμε δυναμικές μεταβλητές.

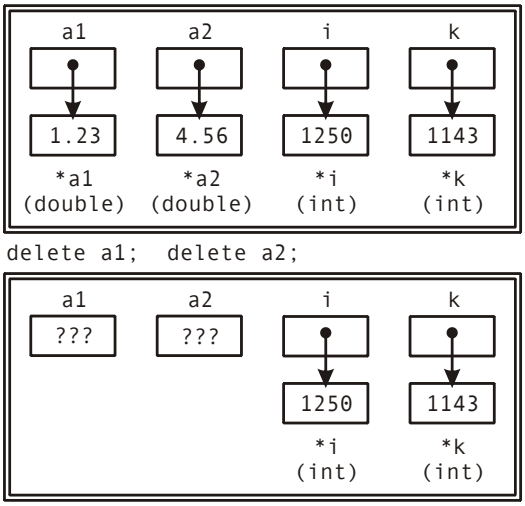
Μετά τη δήλωσή τους οι μεταβλητές αυτές είναι *αόριστες*. Θα τους δώσουμε τιμές με μια παράσταση **new** (Πλ. 16.1). Η πρώτη μορφή της παράστασης είναι γενικώς:

"`new`", *τύπος* ["`(`", *παράσταση*, "`)`"];

και επιστρέφει ένα βέλος προς μια μεταβλητή οργανωμένη όπως καθορίζει ο «*τύπος*». Αν υπάρχει η «*παράσταση*» θα πρέπει να δίνει αποτέλεσμα που μπορεί να μετατραπεί σε τιμή του τύπου της δυναμικής μεταβλητής. Στην πιο απλή περίπτωση μπορείς να δώσεις π.χ.:

```
k = new int;
```

Η μεταβλητη-βέλος *k* ορίζεται και δείχνει μια θέση μνήμης –μια δυναμική μεταβλητή– τύπου `int` που παραχωρήθηκε στο πρόγραμμά μας. Πώς τη χειριζόμαστε; Μα ως "`*k`"! *Μετά την εκτέλεση της εντολής ορίζεται η k αλλά η* `*`*k είναι αόριστη.*



Σχ. 16-2 Μετά τη `delete` η τιμή του βέλους είναι αόριστη και η αντίστοιχη δυναμική μεταβλητή δεν υπάρχει.

Μπορούμε να δώσουμε και

```
i = new int( 375 );
```

Τώρα, ορίζεται η μεταβλητη-βέλος *i* και δείχνει μια δυναμική μεταβλητή (`*`*i*) τύπου `int` αλλά είναι εξ αρχής ορισμένη και η `*`*i* που δημιουργήθηκε με αρχική τιμή "`375`".

Μπορούμε όμως να δώσουμε τη "`new`" και στη δήλωση:

```
double *a1( new double );
```

Τώρα η *a*1 είναι εξ αρχής ορισμένη. Φυσικά, μπορούμε να έχουμε εξ αρχής ορισμένη και τη δυναμική μεταβλητή (`*`*a*1) αν δώσουμε ως αρχική τιμή στο *a*1 "`new double(1.23)`".

Όλα αυτά μπορείς να τα δεις πιο παραστατικά στο Σχ. 16-1. Όπως βλέπεις, ενώ η *k*, ας πούμε, είναι μια *συμβατική μεταβλητή* η `*`*k* είναι πράγματι *δυναμική*. Οι μεταβλητές-βέλη των παραδειγμάτων υπάρχουν από τη στιγμή που αρχίζει να εκτελείται το πρόγραμμά μας (ή η συνάρτηση) όπου έχουν δηλωθεί. Αλλά οι δυναμικές μεταβλητές δεν υπάρχουν μέχρι να παραχωρηθούν από τον τελεστή `new`.

Αν έχουμε τύπο δομής, ας πούμε τον *Date*, μπορούμε να γράψουμε:

```
Date* pd( new Date(Date(2008, 7, 9)) );
```

Δηλαδή: δώσε μου μνήμη για μια δυναμική μεταβλητή τύπου *Date* (που θα τη δείχνει το βέλος *pd*) με αρχική τιμή αυτήν που δημιουργεί ο δημιουργός του τύπου *Date* αν τροφοδοτηθεί με τις τιμές 2008, 7, 9. Πάντως μπορείς να έχεις το ίδιο αποτέλεσμα και με πιο απλό γράψιμο:

```
Date* pd( new Date(2008, 7, 9) );
```

Κατά τα άλλα:

- ***Χειριζόμαστε τις δυναμικές μεταβλητές όπως όλες τις άλλες του ίδιου τύπου.***

Μπορούμε, για παράδειγμα, να τους δίνουμε τιμή ή να αλλάζουμε την τιμή που έχουν με εντολή εκχώρησης

```
*i = (*i)*2 + 500;
```

ή με εντολή εισόδου από αρχείο ή απο την οθόνη:

```
cin >> *k;
```

Για να μιλάμε όμως για δυναμική μνήμη δεν φτάνει μόνον η δυνατότητα παραχώρησης· πρέπει να έχουμε και δυνατότητα επιστροφής όταν δεν τη χρειαζόμαστε. Ο «αντίστροφος» του `new` είναι ο τελεστής `delete` (Πλ. 16.2).

Ας πούμε ότι μετά τη χρήση των δυναμικών μεταβλητών που βλέπουμε στο Σχ. 16-1 βάζουμε τις εντολές

```
delete a1;   delete a2;
```

Πλαίσιο 16.2

Τελεστής `delete`

Αν έχουμε δηλώσει:

```
T*  pv;
```

και έχουμε πάρει δυναμική μνήμη με την

```
pv = new T;
```

η

```
delete pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει η δυναμική μεταβλητή **pv* β) η *pv* να γίνει αόριστη.

Αν έχουμε πάρει δυναμική μνήμη με την:

```
pv = new T[N];
```

η

```
delete[] pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει ο δυναμικός πίνακας που έδειχνε η *pv* β) η *pv* να γίνει αόριστη.

Στο Σχ. 16-2 βλέπεις δυο στιγμιότυπα από τη συνέχεια της εκτέλεσης.

Οι a1 και a2 δεν είναι πια ορισμένες. Η "`delete a1`" «εξαφανίζει» τη θέση μνήμης (μεταβλητή) **a1* και η "`delete a2`" την **a*2. Τί θα πει «εξαφανίζει»; Οι **a1* και **a2* δεν ελέγχονται πια από το πρόγραμμά μας αλλά από το μηχανισμό διαχείρισης της δυναμικής μνήμης. Αυτό σημαίνει ότι η θέση αυτή μπορεί να παραχωρηθεί ξανά με μια επόμενη "`new` …".

Και τώρα προσοχή:

- ***Αμέσως μετά τη "`delete q`" η *q δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.***

Δυστυχώς, η C++ θα αφήσει την τήρηση αυτού του κανόνα στον προγραμματιστή και δεν θα αποπειραθεί να αποτρέψει μια τέτοια παρανομία.

16.3 Τιμές Βελών και Δυναμικών Μεταβλητών

Όπως ήδη ξέρεις, η C++ μας επιτρέπει να τυπώσουμε την τιμή ενός βέλους ή οποιαδήποτε διεύθυνση. Ας πάμε λοιπόν σε αυτά που είδαμε στο Σχ. 16-1 για να δούμε τι θα μπορούσαν να είναι εκείνα τα "???":

```
#include <iostream>
using namespace std;
int main()
{
   double* a1( new double );
   double* a2;  int* i;  int* k;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
```

Αποτέλεσμα; Κάτι σαν

```
0x3e2478  0x34  0x22ffa8  0x7c800000
```

Αν προσπαθήσεις να κάνεις αποπαραπομπές, για παράδειγμα με την:

```
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

είναι πολύ πιθανό να έχεις διακοπή της εκτέλεσης του προγράμματός σου.

Οι αποπαραπομπές γίνονται με ασφάλεια μόνον όταν έχεις ορίσει όλες τις μεταβλητές-βέλη:

```
   a2 = new double;
   i = new int( 375 );   k = new int;
```

```
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

Αποτέλεσμα κάτι σαν

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
0  1.66976e-307  375  4064328
```

Συγκρίνοντας τα τελευταία αποτελέσματα με τα αρχικά βλέπουμε τα εξής: Μετά τις "`new`"

- Οι διευθύνσεις που είναι αποθηκευμένες στα τέσσερα βέλη είναι «κοντινές», στην ίδια περιοχή της μνήμης.
- Οι αποπαραπομπές γίνονται με ασφάλεια αλλά, φυσικά, η μόνη τιμή που έχει νόημα είναι αυτή του `*`*i* (375).

Την πρώτη φορά, η μόνη διεύθυνση που είχε νόημα ήταν η τιμή της *a*1 που ήταν ορισμένη από τη δήλωση. Στις άλλες μεταβλητές-βέλη τι είχαμε; *Το τυχαίο περιεχόμενό τους ερμηνευμένο ως διεύθυνση*. Έτσι, όταν ζητούσαμε να γίνει μια αποπαραπομπή αν η υποτιθέμενη διεύθυνση ήταν έξω από τον *χώρο διευθύνσεων* του προγράμματος είχαμε διακοπή της εκτέλεσής του.

Σημείωση: ▶

Χώρος διευθύνσεων (address space) ενός προγράμματος είναι το σύνολο των διευθύνσεων που μπορεί «νομίμως» να χρησιμοποιήσει όταν εκτελείται. Στα σύγχρονα ΛΣ, που χρησιμοποιούν τεχνολογίες **εικονικής** ή **υπερβατικής** (virtual) μνήμης αυτές οι διευθύνσεις δεν είναι φυσικές αλλά εικονικές που απεικονίζονται στις φυσικές με κάποιον τρόπο[1]. Αυτό εξηγεί και το ότι σε διαφορετικές εκτελέσεις του ίδιου προγράμματος μπορεί να βλέπεις μερικές (εικονικές) διευθύνσεις –όπως εδώ η τιμής της *a*1– να παραμένουν ίδιες.

Δηλαδή, ολη η μνήμη που χρειάζεται το πρόγραμμά μας υπάρχει στον δίσκο. Στην κύρια μνήμη παραχωρείται μια μικρότερη περιοχή (που μπορεί να είναι και *πολύ μικρότερη*). Κατά την εκτέλεση του προγράμματος, αναλόγως των αναγκών, φορτώνονται στην κύρια μνήμη κομμάτια της μνήμης από τον δίσκο ενώ φορτωμένα κομμάτια φυλάγονται στον δίσκο· έχουμε δηλαδή **ανταλλαγές μνήμης** (memory swapping) .◀

Συνεχίζουμε με το πρόγραμμά μας: Δίνουμε τιμές σε όλες τις δυναμικές μεταβλητές:

```
   cin >> *a1;  *a2 = 4.56;  cin >> *k;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και βλέπουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Και τώρα πρόσεξε: Δίνουμε

```
   delete a1;  delete a2;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και παίρνουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Παρά τις "`delete`" οι *a*1 και *a*2 κρατούν τις τιμές τους και οι αποπαραπομπές τους δεν έχουν το παραμικρό πρόβλημα! Και να ήταν μόνο αυτό... Δίνουμε στη συνέχεια τα εξής:

```
   a1 = new double;
   cout << a1 << "  " << *a1 << endl;
   *a1 = 7.13;
   cout << *a2 << endl;
```

και παίρνουμε:

```
0x3e24e0  4.56
```

[1] Έτσι, μπορεί να δείς και τον όρο «virtual address space»

7.13

Με την "`a1 = new double`" η `*a1` υλοποιείται στην ίδια θέση με την `*a2` (που όμως έχουμε «καταργήσει»). Και φυσικά όταν δίνουμε τιμή στην `*a1` αλλάζουμε την τιμή (της καταργημένης) `*a2`.

Ας τα πάρουμε τώρα από την αρχή για να δούμε μερικά κρίσιμα «πρέπει» και «δεν πρέπει».

- *Δεν πρέπει να προσπαθήσεις να κάνεις αποπαραπομπή σε κάποιο βέλος αν δεν υπάρχει η αντίστοιχη δυναμική μεταβλητή-στόχος*. Πολύ περισσότερο, δεν πρέπει να προσπαθήσεις να αλλάξεις την τιμή της (ανύπαρκτης) δυναμικής μεταβλητής· αν η τυχαία τιμή που υπάρχει στο βέλος είναι «νόμιμη» διεύθυνση μπορεί να κάνεις ζημιά! Πώς αντιμετωπίζουμε πρακτικώς αυτό το πρόβλημα;
- *Σε κάθε μεταβλητή-βέλος που δηλώνεις πρέπει να δίνεις οπωσδήποτε αρχική τιμή. Αν δεν την ξέρεις δώσε τιμή* "`0`" ("`NULL`"). (Να υπενθυμίσουμε ότι στην §12.3.2 είδαμε και άλλους τρόπους –εκτός από μια παράσταση **new**– για να δώσουμε αρχική τιμή σε ένα βέλος. Εκεί είδαμε και την τιμή "`0`".)
- *Αν σε κάποιο σημείο του προγράμματός σου δεν έχεις τη σιγουριά ότι μια μεταβλητή-βέλος p δείχνει «νόμιμο» στόχο, πριν κάνεις αποπαραπομπή, πρέπει να ελέγξεις με μια* "`if( p != 0 ) …`"
- *Μετά από μια* "`delete p`", *αν δεν δίνεις μια άλλη νόμιμη τιμή στο βέλος p, πρέπει να βάλεις οπωσδήποτε μια* "`p = 0`" (για να έχει νόημα ο έλεγχος που υποδεικνύεται πιο πάνω). Η "`delete p`"
  - δεν μηδενίζει αυτομάτως την τιμή της *p*,
  - δεν «καθαρίζεται» η τιμή της δυναμικής μεταβλητής `*p`· στο σημείο αυτό θα επανέλθουμε.[2]

16.4 Δυναμικοί Πίνακες

Το ενδιαφέρον της δυναμικής παραχώρησης μνήμης δεν βρίσκεται στο να πάρουμε μια μεταβλητή τύπου `int` όταν εκτελείται το πρόγραμμα αλλά

1. στη δυνατότητα υλοποίησης δυναμικών δομών στοιχείων (λίστες, δένδρα κλπ)
2. στη διαχείριση μεγάλων πινάκων.

Εδώ θα ασχοληθούμε με τη δεύτερη περίπτωση που χρειάζεται τη δεύτερη μορφή των **new** και **delete**. Έστω ότι στο πρόγραμμά μας δηλώνουμε:

```
double*  p( 0 );
```

Αν στη συνέχεια ζητήσουμε:

```
p = new double[100];
```

θα πάρουμε μια περιοχή μνήμης με 100 θέσεις τύπου `double` που μπορούμε να τις διαχειριστούμε σαν να είχαμε δηλώσει:

```
double p[100];
```

με την εξής διαφορά: αντί για το 100, θα μπορούσαμε να είχαμε βάλει στη `new` οποιαδήποτε παράσταση που θα μας έδινε μια θετική ακέραιη τιμή. Αυτή *η τιμή καθορίζεται τη στιγμή που εκετελείται η εντολή και όχι όταν μεταγλωττίζεται*, όπως γίνεται με τη δήλωση ενός συμβατικού πίνακα.

---

[2] Δηλαδή η "`delete`", παρά το όνομά της, δεν κάνει διαγραφές· απλώς παραδίδει στην *ανακύκλωση* τη δυναμική μνήμη που είχαμε σε χρήση.

Παρατήρηση: ▶
«Θετική ακέραιη τιμή»! Δηλαδή δεν μπορεί να είναι μηδέν; Η C++ δεν έχει αντίρρηση αλλά υπάρχουν κάτι προβληματάκια που μπορεί να μας έλθουν από τη C. Τα συζητούμε παρακάτω.◀

Χειριζόμαστε τον δυναμικό πίνακα όπως τους συμβατικούς. Για παράδειγμα τα στοιχεία του p είναι τα p[0], p[1], …, p[99] και χειριζόμαστε το κάθε ένα από αυτά όπως μια μεταβλητή τύπου `double`.

Όταν δεν μας χρειάζεται η μνήμη, μπορούμε να τη επιστρέψουμε με την εντολή:

```
delete[] p;
```

Αλλά εδώ υπάρχει κάτι που χρειάζεται ιδιαίτερη προσοχή: δεν επιτρέπεται να ανακατεύεις τις δύο μορφές των **new** και **delete**.

- ***Όταν παίρνεις δυναμική μνήμη με "`p = new T`" θα την επιστρέφεις με "`delete p`" και όταν παίρνεις δυναμική μνήμη με "`p = new T[...]`" θα την επιστρέφεις με "`delete[] p`".***

Έτσι, *θα ήταν λάθος αν προσπαθούσαμε να επιστρέψουμε τον δυναμικό πίνακα* p *με μια* "`delete p`". Αν πάλι είχαμε δηλώσει

```
double q( new double );
```

είναι λάθος να προσπαθήσουμε να ανακυκλώσουμε την `*q` *με* "`delete[] q`".

Ιδιαίτερο ενδιαφέρον έχουν οι πολυδιάστατοι δυναμικοί πίνακες. Θα τα πούμε στη συνέχεια. Προς το παρόν ας ξαναδούμε ένα παράδειγμα από τα παλιά.

Παράδειγμα
Θα ξαναγράψουμε το πρόγραμμα που «μετράει» το αρχείο alturing.txt, που είδαμε στο Παράδ. 1 της §8.10, *αλλά*

- αφού φορτώσουμε το περιεχόμενο του αρχείου στη μνήμη (§15.12.1),
- σε δυναμικό πίνακα.

Κατ' αρχάς θα γράψουμε μια

```
void loadText( string flNm, char*& toText, size_t& tLength )
```

που τροφοδοτείται με το όνομα ενός αρχείου (*flNm*), φορτώνει το περιεχόμενό του σε έναν δυναμικό πίνακα (με στοιχεία τύπου `char`) και μας επιστρέφει βέλος προς την αρχή του πίνακα (*toText*) και το πλήθος στοιχείων του πίνακα (*tLength*). Η συνάρτηση θα ρίχνει εξαίρεση αν δεν μπορεί να ανοίξει ή να διαβάσει το αρχείο ή αν δεν μπορεί να πάρει δυναμική μνήμη.

```
void loadText( string flNm, char*& toText, size_t& tLength )
{
   ifstream tin( flNm.c_str(), ios_base::binary|ios_base::ate );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotOpen,
                        flNm.c_str() );
   tLength = tin.tellg();
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
   tin.seekg( 0 );
   tin.read( toText, tLength );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotRead,
                        flNm.c_str() );
   tin.close();
   toText[tLength] = '\0';
} // loadText
```

Σε σχέση με αυτά που είδαμε στην §15.12.1 το νέο στοιχείο είναι η δυναμική μνήμη:

```
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
```

```
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
```

Γιατί παίρνουμε *tLength*+1; Για να χωρέσει και ο φρουρός (`'\0'`) που βάζουμε στην τελευταία (παραπανίσια) θέση.

Σε μια άλλη συνάρτηση βάζουμε την επεξεργασία (μέτρημα):

```
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits )
{
   int pos( 0 );
   char ch;      // χαρακτήρας που διαβάζουμε
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
// Επεξεργάσου το αρχείο
   ch = toText[pos];
   while ( pos < tLength )
   {
      while ( pos < tLength && ch != '\n' )
      {
         if ( isupper(ch) ) ++nUpCase;
         else if ( isdigit(ch) )  ++nDigits;
         ++pos;  ch = toText[pos];
      }
      ++nRows;
      if ( pos < tLength )  {  ++pos;  ch = toText[pos];  }
   } // while
} // countText
```

Πρόσεξε τα εξής:

- Αν *pos* είναι ο δείκτης του χαρακτήρα *ch* που επεξεργαζόμαστε (toText`[`*pos*`]` `==` *ch*) τότε η πρώτη "`t.get(ch)`" γίνεται: "`pos = 0; ch = toText[pos];`" και η επαναλαμβανόμενη: "`++pos; ch = toText[pos]`".
- Η "`!t.eof()`" γίνεται "`pos < tLength`". Θα μπορούσε να γίνει και "`ch != '\0'`"; Ναι, αλλά η πρώτη είναι προτιμότερη αφού δουλεύει και στην περίπτωση που υπάρχουν `'\0'` μέσα στο αρχείο. Να τονίσουμε, βέβαιως, ότι δεν περιμένουμε να βρούμε τέτοιους χαρακτήρες σε ένα αρχείο text.
- Ο έλεγχος τέλους γραμμής παραμένει: "`ch != '\n'`". Θα πρέπει να τον αλλάξουμε αν έχουμε αρχείο text που οι γραμμές του διαχωρίζονται από ακολουθία χαρακτήρων που δεν περιέχει τον `'\n'`.
- Αυτή η συνάρτηση δεν ρίχνει κάποια εξαίρεση; Θα μπορούσαμε να βάλουμε ελέγχους όπως "`toText == 0`" ή "`tLength < 0`" αλλά για ένα τόσο απλό πρόγραμμα δεν έχει νόημα.

Έχοντας δηλώσει:

```
   char*  toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t tLength;     // μήκος κειμένου
   int    nRows;       // μετρητής γραμμών
   int    nUpCase;     // μετρητής κεφαλαίων
   int    nDigits;     // μετρητής ψηφίων
```

καλούμε τις δύο συναρτήσεις ως εξής:

```
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
```

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <new>

using namespace std;
```

```
struct ApplicXptn
{
   enum { cannotOpen, cannotRead, allocFailed };
   char funcName[100];
   int  errCode;
   char errStrVal[100];

   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ApplicXptn

void loadText( string flNm, char*& toText, size_t& tLength );
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits );

int main()
{
   char*     toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t    tLength;     // μήκος κειμένου
   int       nRows;       // μετρητής γραμμών
   int       nUpCase;     // μετρητής κεφαλαίων
   int       nDigits;     // μετρητής ψηφίων

   try
   {
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
      delete[] toText;  toText = 0;
   // Λέγε τα αποτελέσματα
      cout << " Διάβασα " << nRows << " γραμμές" << endl;
      cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
           << nDigits << " ψηφία" << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

16.5 * Η Τρίτη Μορφή του "new"

Υπάρχει και μια τρίτη μορφή χρήσης του τελεστή "**new**" που δεν έχει σχέση με δυναμική παραχώριση μνήμης. Δες ένα παράδειγμα: Οι εντολές

```
   int buf[100];
   double* d( new (buf) double );
   cout << &buf[0] << endl;
   cout << d << endl;
```

δίνουν:

```
0x22fde0
0x22fde0
```

Δηλαδή: η μεταβλητή **d* δημιουργείται στην αρχή του πίνακα buf.

Όπως φαίνεται και από το παράδειγμα, αυτή η μορφή του "**new**" έχει ως αποτέλεσμα *την δημιουργία μιας μεταβλητής σε μια συγκεκριμένη θέση της μνήμης*· δεν υπάρχει παραχώρηση δυναμικής μνήμης στο πρόγραμμά μας και επομένως δεν υπάρχει αντίστοιχη μορφή "**delete**". Αυτή η μορφή λέγεται **new τοποθέτησης** (placement new).

Πρόσεξε ένα άλλο παράδειγμα:

```
   char* buf( new char[100] );
   double* d( new (buf) double );
// . . .
   delete[] buf;
```

Εδώ γιατί βάλαμε **delete**; Για να ανακυκλώσουμε τη buf που πήραμε με **new** *δεύτερης μορφής* ("**new char[100]**"). Φυσικά, ανακυκλώνεται και η **d*, αφού δημιουργήθηκε στην αρχή του πίνακα buf.

Σε μικρά μονοχρηστικά συστήματα μπορείς να χρησιμοποιήσεις αυτήν τη μορφή της "**new**" για να βάλεις μεταβλητές κατάλληλου τύπου σε συγκεκριμένες διευθύνσεις για να διαχειριστείς με το πρόγραμμά σου ενταμιευτές συσκευών ή σχετικούς καταχωρητές. Αν δουλεύεις σε Linux ή Windows ή Unix ή άλλο παρόμοιο σύστημα μη διανοηθείς να χρησιμοποιήσεις απόλυτες διευθύνσεις μνήμης. Όπως είπαμε και πιο πριν οι διευθύνσεις που βλέπει το πρόγραμμά σου είναι **εικονικές**.

16.6 Η Εξαίρεση *bad_alloc*

Οι σημερινοί υπολογιστές διαθέτουν τεράστιες ποσότητες μνήμης (σε σύγκριση με αυτά που ξέραμε μέχρι πριν από λίγα χρόνια). Με τα συστήματα εικονικής μνήμης τα όρια φτάνουν τη χωρητικότητα των δίσκων σου. Παρ' όλα αυτά, ένα καλό πρόγραμμα θα πρέπει να είναι προετοιμασμένο για την περίπτωση που θα αποτύχει μια **new**, δηλαδή δεν θα μας δοθεί η μνήμη που ζητάμε. Στην περίπτωση αυτή ρίχνεται μια εξαίρεση τύπου *bad_alloc*.

Αυτό σημαίνει ότι η **new** θα πρέπει να βρίσκεται πάντοτε μέσα σε μια ομάδα **try** που θα ακολουθείται από μια "**catch(bad_alloc&)**". Για να μπορέσεις να χρησιμοποιήσεις τη *bad_alloc* στο πρόγραμμά σου θα πρέπει να έχεις βάλει μια:

```
#include <new>
```

Κατ' αρχήν λοιπόν έχουμε ένα σχέδιο της μορφής:

```
// . . .
#include <new>
// . . .
   try
   {
      double dr( new double[1000] );
// . . .
   }
// . . .
   catch( bad_alloc& )
   {
```

```
   // διαχείριση της εξαίρεσης
   }
// . . .
```

Στη συνέχεια θα βλέπεις να κάνουμε μια κάπως διαφορετική αντιμετώπιση: Θα πιάνουμε την εξαίρεση *bad_alloc* και θα ρίχνουμε μια δική μας,

```
// . . .
   double dr( 0 );
   try {  dr = new double[1000];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "thisFunc", MyProgXptn::allocFailed );  }
// . . .
```

Και τι θα μπορούσαμε να κάνουμε όταν πιάσουμε μια τέτοια εξαίρεση; Πιθανότατα να ανακυκλώσουμε κάποιον (ή κάποιους) τεράστιο πίνακα (-ες) που δεν μας χρειάζονται πια.

16.6.1 Μια Εξήγηση για τις Εξαιρέσεις μας

Τώρα μπορούμε να εξηγήσουμε και ένα χαρακτηριστικό των δικών μας εξαιρέσεων που μπορεί να σου φαίνεται περίεργο.

Γιατί δηλώνουμε

```
   char        funcName[100];
```

και όχι:

```
   string      funcName;
```

όπως συνήθως; Διότι ένα από τα προβλήματα που έχουμε να διαχειριστούμε είναι η έλλειψη δυναμικής μνήμης (*allocFailed*). Η *string* χρησιμοποιεί *δυναμική* μνήμη για να αποθηκεύσει το κείμενο. Αν λοιπόν το πρόβλημά μας είναι ότι *δεν μπορεί να παραχωρηθεί δυναμική μνήμη* και προσπαθήσουμε να ρίξουμε *εξαίρεση που θα χρειαστεί δυναμική μνήμη* – που πιθανότατα δεν θα μπορεί να πάρει– θα προκαλέσουμε «βίαιη» διακοπή της εκτέλεσης του προγράμματός μας. Χρησιμοποιώντας πίνακα τύπου `char` που υλοποιείται στη στοίβα τα πράγματα είναι ασφαλή.

16.7 Τα Προβλήματα της Δυναμικής Μνήμης

Η παραχώρηση δυναμικής μνήμης είναι ένα πολύ καλό εργαλείο αλλά έχει και ορισμένα προβλήματα η αντιμετώπιση των οποίων χρειάζεται την προσοχή μας όταν γράφουμε το πρόγραμμα.

Ξεκινάμε αντιγράφοντας –με ορισμένες προσαρμογές– ένα σχήμα και μερικά πράγματα από την §12.3.2:

Ας πούμε ότι δηλώνουμε:

```
   int* a( new int(10) );
   int* b( new int(20) );
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 16-3(α): το βέλος *a* δείχνει την `*a`, που έχει τιμή 10, και το βέλος *b* δείχνει την `*b` που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή "`*a = *b`". Η εικόνα της μνήμης είναι αυτή του Σχ. 16-3 (β). Η τιμή της `*a` γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

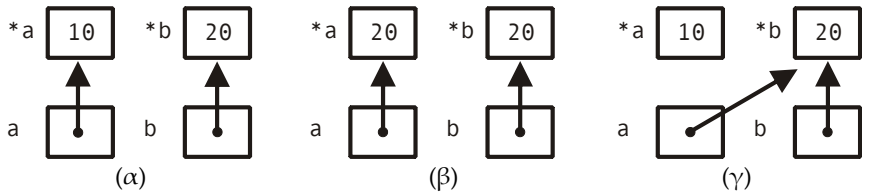
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: "`a = b`". Οι τιμές των μεταβλητών `*a` και `*b` δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *a*· το βέλος *a* δείχνει εκεί που δείχνει και το *b*: τη `*b` (Σχ. 16-3 (γ)).

Φυσικά, και στις δύο περιπτώσεις, αν δώσουμε την εντολή:

```
   cout << *a << "  " << *b << endl;
```

θα πάρουμε αποτέλεσμα:

```
20  20
```



Σχ. 16-3 (τροποποίηση του Σχ. 12-3) Αρχικώς, μετά την εκτέλεση των δηλώσεων (με αρχικές τιμές) `int* a(new int(10)); int* b(new int(20));` έχουμε την εικόνα (α). Η εικόνα (β) προκύπτει αν στη συνέχεια εκτελεσθεί η εκχώρηση: `*a = *b`. Η εικόνα (γ) προκύπτει αν, μετά την (α), εκτελεσθεί η εκχώρηση `a = b`.

αλλά το «παρασκήνιο» είναι διαφορετικό· και στη δεύτερη περίπτωση δημιουργείται το εξής πρόβλημα: *το πρόγραμμά μας έχει μια δυναμική μεταβλητή, την* `*a`*, αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί* (*ούτε να την ανακυκλώσει*).

Αυτό που περιγράψαμε πιο πάνω, είναι γνωστό ως **απώλεια** ή **διαρροή μνήμης** (memory leakage). Μπορεί να εμφανισθεί στις εξής περιπτώσεις:

- Ένα βέλος *p* είναι το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης και χωρίς να την ανακυκλώσουμε, αλλάζουμε την τιμή του *p*, όπως είδαμε παραπάνω.
- Ένα βέλος *p*, όντας το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης, είναι τοπική μεταβλητή σε μια συνάρτηση. Η εκτέλεση της συνάρτησης τελειώνει χωρίς να ανακυκλώσουμε τη δυναμική μνήμη το βέλος *p* «χάνεται». Αυτό φαίνεται στο παρακάτω παράδειγμα.

Έστω ότι έχουμε τη συνάρτηση:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
} // f
```

Ας πούμε λοιπόν ότι

- καλείται η *f*,
- εκτελείται επιτυχώς η "`pi = new int[ni]`" και παίρνει μνήμη για τον pi, αλλά
- αποτυγχάνει να πάρει μνήμη για τον pd και ρίχνεται μια *MyProgXptn* από τη δεύτερη `throw`.
- Διακόπτεται η εκτέλεση της *f* και παύουν να υπάρχουν τα βέλη *pi* και *pd* (επιστρέφονται στη στοίβα).

Έτσι, το πρόγραμμά μας έχει δεσμεύσει μνήμη για *ni* θέσεις τύπου `int` αλλά δεν έχει πρόσβαση σε αυτήν. Τι θα έπρεπε να κάνουμε; Πριν ρίξουμε την εξαίρεση θα έπρεπε να ανακυκλώσουμε τη μνήμη που ήδη πήραμε:

```
// . . .
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
```

Δεν τελειώσαμε όμως· αν μπορεί να εγερθεί εξαίρεση από αυτά που ακολουθούν θα πρέπει να ανακυκλώσουμε και τους δύο πίνακες:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try
   {
   // άλλες εντολές
   }
   catch( ... )
   {
      delete[] pi;
      delete[] pd;
      throw;
   }
} // f
```

Πρόσεξε την τελευταία `catch`:

- πιάνει οποιαδήποτε εξαίρεση μπορεί να προέλθει από τις «*άλλες εντολές*»,
- ανακυκλώνει τους δύο πίνακες και
- την ξαναρίχνει!

Ένα άλλο πρόβλημα, από μια άποψη το «αντίστροφο» αυτού που είδαμε, είναι το πρόβλημα του **μετέωρου βέλους** (pending ή dangling pointer). Ας πούμε ότι είμαστε στην κατάσταση (γ) του Σχ. 16-3 και δίνουμε

```
   delete b;  b = 0;
```

Όπως βλέπεις, ανακυκλώνουμε τη **b* και –τηρώντας τον κανόνα που βάλαμε– μηδενίζουμε το βέλος *b*. Καλά όλα αυτά, αλλά η **b* ήταν ταυτοχρόνως και **a*· έτσι τώρα το βέλος *a* είναι *μετέωρο* αφού δείχνει σε θέση μνήμης που δεν ανήκει στο πρόγραμμά μας.

Και αυτό το πρόβλημα μπορεί να προέλθει από απρόσεκτο γράψιμο συναρτήσεων. Δες την παρακάτω:

```
void g( double* dAr, int n )
{
   int nn;
// . . .
   if ( n < nn )
   {
      double* dArL( new double[nn] );
// . . .
      delete[] dAr;  dAr = dArL;
      n = nn;
   } // if
// . . .
} // g
```

Αυτή η συνάρτηση τροφοδοτείται με έναν δυναμικό πίνακα. Σε κάποιο σημείο μπορεί να ανακαλύψει ότι το μέγεθος του πίνακα δεν είναι κατάλληλο και τον αλλάζει.

Προσοχή! Η παράμετρος-βέλος *dAr* είναι παράμετρος τιμής. Το ότι δεν έχουμε βάλει "`const`" μας επιτρέπει να αλλάζουμε τις τιμές των στοιχείων του πίνακα αλλά το *dAr* είναι αντίγραφο.της πραγματικής παραμέτρου· αν αλλάξουμε την τιμή του βέλους η αλλαγή είναι τοπική μέσα στη συνάρτηση.

Ας πούμε λοιπόν ότι καλούμε τη συνάρτηση ως εξής:

```
   double* bd( 0 );
   int bn( 0 );
// . . .
```

```
g( bd, bn );
```

Όταν αρχίσει η εκτέλεση της συνάρτησης το βέλος *dAr* είναι αντίγραφο του *bd* και έτσι δείχνουν και τα δύο την ίδια θέση της μνήμης. Αν η συνάρτηση βρει ότι το μέγεθος του πίνακα δεν είναι αρκετό παίρνει την κατάλληλη μνήμη (με το βέλος *dArL*) και ανακυκλώνει την παλιά ("`delete[] dAr`") –αφού, πιθανότατα, αντιγράψει το περιεχόμενό της.

- Στην περίπτωση αυτήν το *bd*, που έδειχνε τη μνήμη που ανακυκλώθηκε, είναι πια *μετέωρο*.
- Μετά το τέλος εκτέλεσης της συνάρτησης το *τοπικό* βέλος *dArL* χάνεται (επιστρέφει στη στοίβα) και δεν έχουμε εργαλείο για να χειριστούμε τη δυναμική μνήμη που έδειχνε. Έχουμε δηλαδή *διαρροή μνήμης*.

Το πρόβλημα εξαφανίζεται αν περάσουμε τον δυναμικό πίνακα με *παραμέτρους αναφοράς*:

```
void g( double*& dAr, int& n )
```

Τα επόμενα προβλήματα έχουν σχέση με τον τελεστή "`delete`".

Το πρώτο είναι αυτό που ήδη είπαμε (§16.4): *Η μορφή του "`delete`" με την οποίαν θα ανακυκλώνουμε τη δυναμική μνήμη κάθε φορά πρέπει να είναι αντίστοιχη της μορφής του "`new`" που χρησιμοποιήσαμε για την παραχώρησή της.* Αν παραβιάσεις αυτόν τον κανόνα μπορεί να κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης (heap).

Δεν υπάρχει κάποια «συνταγή» για να διασφαλίσεις την τήρηση αυτού του κανόνα. Η συνηθισμένη σχετική υπόδειξη λέει: *Για κάθε κομμάτι δυναμικής μνήμης που παίρνεις φρόντισε η παράσταση* **new** *και η αντίστοιχη παράσταση* **delete** *να βρίσκονται στην ίδια συνάρτηση.* Αν φροντίζεις να γράφεις και μικρές συναρτήσεις τότε κάτι μπορεί να γίνει... Πάντως, όπως θα καταλάβεις και από την πείρα σου, όταν γράφεις προγράμματα με δυναμική μνήμη, δεν είναι και τόσο απλό να τηρείς αυτόν τον κανόνα.

Ένα άλλο πρόβλημα με τη "`delete`" είναι το εξής: *αν, κατά λάθος φυσικά, αποπειραθείς να ανακυκλώσεις μνήμη που έχεις ήδη ανακυκλώσει* (και επομένως δεν ελέγχεται από το πρόγραμμά σου) *κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης*. Φυσικά, δεν λέει κανείς ότι θα πας να γράψεις στο πρόγραμμά σου

```
delete p;
delete p;
```

Είναι όμως πολύ πιθανό να γράψεις κάτι σαν:

```
if ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
while ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
f( ..., p, ... );
delete p;
```

και μέσα στην *f* εκτελείται άλλη μια "`delete p`".

Αν τηρείς τον κανόνα «*μηδένισε το βέλος μετά τη* `delete`» γλυτώνεις από αυτό το πρόβλημα. Αν έχουμε:

```
delete p;  p = 0;
```

```
delete p;
```

η δεύτερη "`delete p`" είναι στήν πραγματικότητα "`delete 0`" και

- ***Η "`delete 0`" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "`delete[] 0`".***

Το τελευταίο που έχουμε να πούμε είναι το εξής: *Μην προσπαθήσεις να ανακυκλώσεις μνήμη που δεν είναι δυναμική*. Θα δείξουμε πώς μπορεί να συμβεί κάτι τέτοιο με ένα παράδειγμα. Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση που θα τροφοδοτείται με έναν πίνακα a με στοιχεία τύπου `int` και μια τιμή *v* τύπου `int`. Η συνάρτηση θα ψάχνει να βρει τη *v* μέσα στον a και αν δεν τη βρει θα την εισάγει. Τι θα κάνεις αν ο πίνακας είναι γεμάτος;

- Αν *ο πίνακας δεν είναι δυναμικός* η συνάρτηση θα πρέπει να βγάζει ένα μήνυμα – πιθανότατα μια εξαίρεση– για το πρόβλημα.
- Αν *ο πίνακας είναι δυναμικός* η συνάρτηση θα προσπαθήσει να τον «μεγαλώσει». Όπως θα δούμε στη συνέχεια, αυτό σημαίνει αντιγραφή σε έναν μεγαλύτερο πίνακα και ανακύκλωση του παλιού.

Όταν γράφεις τη συνάρτηση δεν ξέρεις με τι είδους πίνακες θα χρησιμοποιείται και αν ξεχάσεις την πρώτη απο τις παραπάνω δυνατότητες είναι πολύ πιθανό να κάνεις προσπάθεια να ανακυκλώσεις μη δυναμικό πίνακα.

Πώς λύνεται το πρόβλημα; Με διαχωρισμό των στόχων:

- Γράψε μια συνάρτηση που αναζητεί μια τιμή σε έναν πίνακα είτε αυτός είναι δυναμικός είτε όχι.
- Γράψε μια συνάρτηση που εισάγει μια τιμή σε έναν πίνακα που *δεν είναι γεμάτος* είτε αυτός είναι δυναμικός είτε όχι.

Αυτές μπορεί να είναι συναρτήσεις γενικής χρήσης· μπορεί να είναι και περιγράμματα.

Ανάμεσα στις κλήσεις αυτών των δύο συναρτήσεων βάλε τις εντολές –που εξαρτώνται από το πρόβλημα– για τις κατάλληλες ενέργειες όταν ο πίνακας είναι γεμάτος.

Αργότερα, όταν θα δούμε την STL, θα δούμε ότι υπάρχουν εργαλεία που μας απαλλάσσουν από αυτά τα προβλήματα. Τέτοια είναι:

- τα **έξυπνα βέλη** (smart pointers) που δεν έχουν αυτά τα προβλήματα και
- το *vector* και τα άλλα περιγράμματα **περιεχόντων** (containers).

16.7.1 RAII: Μια Καλύτερη Λύση

Τώρα, θα ξαναγυρίσούμε στη συνάρτηση *f* που είδαμε ως παράδειγμα για τη διαρροή μνήμης μέσα σε συνάρτηση· ναι αυτήν με τις τρεις **try-catch**! Θα τη χρησιμοποιήσουμε τώρα ως παράδειγμα για να παρουσιάσουμε μια άλλη λύση, σαφώς καλύτερη, με μια πάγια τεχνική της C++ που ονομάζεται «*Resource Acquisition Is Initialization*» (RAII), δηλαδή: *πρόσκτηση πόρων σημαίνει εκκίνηση*.

Πριν προχωρήσεις καλό θα είναι να κάνεις μια επανάληψη σε όσα είπαμε –έστω και ακροθιγώς– για δημιουργούς (§15.3) και καταστροφείς (§15.3.1).

Για το παράδειγμά μας τώρα, ορίζουμε έναν τύπο ως εξής:

```
struct IntDArr
{
   int* da;
   IntDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "IntDArr", MyProgXptn::allocFailed );  }
   }
   ~IntDArr()  {  delete[] da;  }
}; // IntDArr
```

Κάθε μεταβλητή αυτού του τύπου «κρύβει» μέσα της έναν δυναμικό πίνακα με στοιχεία τύπου `int`. Μέσα στη συνάρτηση, αντί για "`int* pi(0)`" βάζουμε τη δήλωση:

```
   IntDArr    pi( ni );
```

Για την εκτέλεσή της καλείται ο δημιουργός που παίρνει τη μνήμη που απαιτείται για τον δυναμικό πίνακα `pi.da`.

Σημείωση: ▶

Εδώ όμως έχουμε και καινούρια πράγματα: *Εξαίρεση από δημιουργό*! Ναι! Αν κάτι «πάει στραβά» (δεν πήραμε μνήμη) ρίχνεται εξαίρεση και *δεν δημιουργείται το αντικείμενο*!

Αυτό βέβαια μας βάζει και ιδέες· δεν θα μπορούσαμε να βάλουμε ελέγχους στον δημιουργό της *Date* –ας πούμε– και να ρίχνουμε εξαίρεση αν μας δώσουν μήνα 37; Ναι και θα το κάνουμε αργότερα. ◀

Στη συνέχεια, μπορούμε να χρησιμοποιούμε τον πίνακα, αλλά για το στοιχείο *k* θα πρέπει να γράφουμε `pi.da[k]` αντί για `pi[k]`. Αν διακοπεί η εκτέλεση της συνάρτησης για κάποιο λόγο –είτε διότι φτάσαμε στο τέλος της είτε λόγω εξαίρεσης– η μεταβλητή `pi` θα καταστραφεί με *αυτόματη κλήση του καταστροφέα*. Ο καταστροφέας, όπως βλέπεις, ανακυκλώνει τον πίνακα και έτσι δεν έχουμε διαρροή μνήμης.

Πολύ ωραία, αλλά για να δούμε την εξής περίπτωση: Ας πούμε ότι κάποια από τις παραμέτρους της συνάρτησης που δεν βλέπουμε είναι "`int*& ipp`" και –αν όλα πάνε καλά– θέλουμε να δείχνει στον νέο πίνακα (που έτσι θα «επιζήσει» μετά την εκτέλεση της συνάρτησης). Κανένα πρόβλημα:

```
   int* psv( pi.da );
   pi.da = ipp;
   ipp = psv;
```

Δηλαδή: αντιμεταθέτουμε τις τιμές των βελών `pi.da` και `ipp` και έτσι το `ipp` δείχνει τον νέο δυναμικό πίνακα ενώ, όταν καταστραφεί η `pi`, θα ανακυκλωθεί ο πίνακας που έδειχνε αρχικώς η `ipp` (αν δεν είχε τιμή 0).

Για να χειριστούμε παρομοίως και τον δυναμικό πίνακα τύπου `double` ορίζουμε:

```
struct DoubleDArr
{
   int* da;
   DoubleDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "DoubleDArr", MyProgXptn::allocFailed ); }
   }
   ~DoubleDArr()  {  delete[] da;  }
}; // DoubleDArr
```

και δες πώς γίνεται η *f*:

```
void f( int ni, int nd, . . . )
{
   IntDArr    pi( ni );
   DoubleDArr pd( nd );

// . . . όπως πριν αλλά
// αντικαθιστώντας το κάθε pi[k] με pi.da[k] και
// το κάθε pd[k] με pd.da[k]
} // f
```

Αγνώριστη και πολύ καλύτερη!

Αργότερα, όταν εξοικειωθείς περισσότερο με δημιουργούς και καταστροφείς θα κατάνοήσεις καλύτερα την τεχνική και θα καταλάβεις ότι είναι πολύτιμη σε πολλές περιπτώσεις.

Οι *IntDArr* και *DoubleDArr* λέγονται **δομές** (ή **κλάσεις**) **περιτυλίγματος** (wrapper structures ή classes). Θα τις ξαναδούμε και για άλλες δουλειές.

16.8 Προβλήματα και στις Δομές

Τώρα θα επισημάνουμε άλλο ένα πρόβλημα αλλά θα αφήσουμε τη λύση του για αργότερα.

Ας πούμε ότι κάνουμε τις εξής αλλαγές στον τύπο *Address*:

```
struct AddressD
{
   char* country;
   char* city;
   int   areaCode;
   char* street;
   int   number;
}; // AddressD
```

με σκοπό να κάνουμε οικονομία και να μην σπαταλούμε τον ίδιο χώρο για την οδό "Κώ" και την οδό "Αγίου Κωνσταντίνου". Όλα δυναμικά!

Πρόσεξε τώρα: ας πούμε ότι έχουμε:

```
   AddressD a1, a2;
```

και –αφού πάρουμε την απαραίτητη δυναμική μνήμη– δίνουμε τιμές σε όλα τα μέλη του *a*1.

Τι αποτέλεσμα θα έχει η εκχώρηση "`a2 = a1`" στη συνεχεια; Τα τρία *βέλη* *a*1.*country*, *a*1.*city*, *a*1.*street* θα αντιγραφούν στα αντίστοιχα *βέλη* του a2 αλλά *δεν θα έχουμε αντιγραφές κειμένων*. Έτσι, θα έχουμε τρεις δυναμικούς πίνακες χαρακτήρων που ο καθένας του στοχεύεται από δύο βέλη. Αλλάζουμε την πόλη στο *a*1 αλλάζει και η πόλη στο *a*2· αλλάζουμε τη χώρα στο *a*2 αλλάζει και η χώρα στο *a*1. Σπανίως θέλουμε να συμβαίνει κάτι τέτοιο.

Η διόρθωση αυτής της συμπεριφοράς γίνεται με τη σωστή επιφόρτωση του τελεστή εκχώρησης. Αυτό θα το δούμε αργότερα.[3] Προς το παρόν, αν θέλεις να μην σπαταλάς μνήμη και να κάνεις εύκολα τη δουλειά σου, τη λύση τη ξέρεις: χρησιμοποίησε τον τύπο *string*.[4]

16.9 Δισδιάστατοι Δυναμικοί Πίνακες

Θα δούμε τώρα πώς μπορούμε να έχουμε δισδιάστατους δυναμικούς πίνακες.

Ας πούμε ότι μας δίνεται ένα μορφοποιημένο αρχείο (όνομα στον δίσκο `egr63e2.txt`) όπου υπάρχουν, σίγουρα, οι τιμές των στοιχείων (πραγματικοί) ενός δισδιάστατου πίνακα. Στην πρώτη γραμμή υπάρχουν δύο θετικοί ακέραιοι που δίνουν το πλήθος γραμμών και το πλήθος στηλών του πίνακα. Σε κάθε μια από τις επόμενες γραμμές του αρχείου υπάρχουν οι τιμές των στοιχείων μιας γραμμής του πίνακα. Θέλουμε να διαβάσουμε το αρχείο και να αποθηκεύσουμε το περιεχόμενό του σε έναν δυναμικό δισιδιάστατο πίνακα.

Αρχίζουμε ανοίγοντας το αρχείο. Αμέσως μετά διαβάζουμε τους αριθμούς γραμμών και στηλών:

```
   ifstream tin( "egr63e2.txt" );
   int      nR, nC;
   tin >> nR >> nC;
```

Ο πίνακάς μας θα είναι *nR*×*nC*.

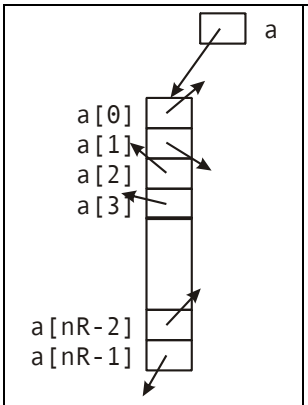
Αν δούμε τον δισδιάστατο δυναμικό πίνακα ως *πίνακα γραμμών –πίνακα πινάκων–* θα έχουμε *nR* μονοδιάστατους δυναμικούς πίνακες που καθένας τους θα έχει *nC* στοιχεία τύπου `double`. Για κάθε έναν από αυτούς τους πίνακες χρειαζόμαστε ένα βέλος τύπου `double*`· επομένως θα χρειαστούμε έναν δυναμικό πίνακα με *nR* στοιχεία τύπου `double*`:

```
   (double*)* a;
```

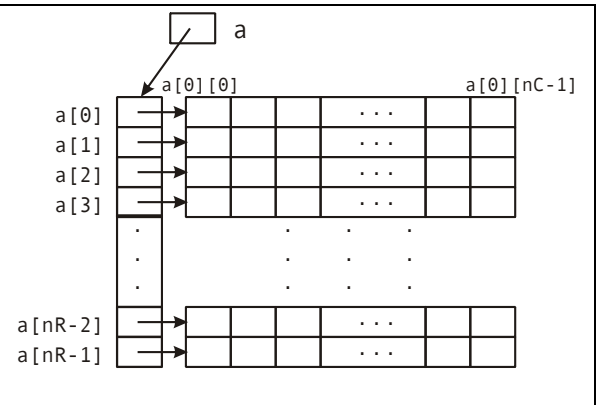
---

[3] Μην προσπαθήσεις να επιφορτώσεις τον "`=`" για τύπο δομής με αυτά που είπαμε στην §14.6.3. Αργότερα θα μάθουμε πώς γίνεται.

[4] Ε, δεν μπορεί να τρόμαξες από αυτά που είπαμε για τη *save()* και τη *load()*! Αφού, όπως είδες, όλα διορθώνονται.



Σχ. 16-4 Τι έχουμε μετά την εκτέλεση της εντολής
`a = new double*[nR]`



Σχ. 16-5 Τι έχουμε μετά την εκτέλεση της εντολής
`for ( int r(0); r < nR; ++r )`
`  a[r] = new double[nC];`

ή απλούστερα:

```
double** a;
```

Έχουμε δηλαδή ένα *διπλό βέλος* ή *βέλος προς βέλος*.

Μόλις μάθουμε το πλήθος γραμμών *nR* του πίνακα μπορούμε να πάρουμε μνήμη για αυτόν τον πίνακα βελών:

```
a = new double*[nR];
```

Στο Σχ. 16-4 βλέπεις μια εικόνα του πίνακα a μετά την εκτέλεση της παραπάνω εντολής. Τα στοιχεία (βέλη) του πίνακα, τα `a[`*r*`]`, *r*: 0..*nR*-1, δεν είναι ορισμένα· αυτό παριστάνεται με τα βέλη που δείχνουν σε τυχαίες διευθύνσεις, νόμιμες ή όχι.

Το τελευταίο βήμα είναι η υλοποίηση των γραμμών με δυναμική μνήμη. Αυτό γίνεται με την:

```
for ( int r(0); r < nR; ++r ) a[r] = new double[nC];
```

Τη μορφή του πίνακα μετά από αυτήν την εντολή τη βλέπεις στο Σχ. 16-5. Το στοιχείο *c* της γραμμής *r* είναι το "`(a[r])[c]`" ή απλούστερα "`a[r][c]`". Αυτόν τον συμβολισμό τον ξέρουμε ήδη από τους μη δυναμικούς δισδιάστατους πίνακες.

Αν θελήσουμε να διαβάσουμε τα στοιχεία του πίνακα από το αρχείο, αυτό θα γίνει κατά τα γνωστά:

```
for ( int r(0); r < nR; ++r )
{
   for ( int c(0); c < nC; ++c ) tin >> a[r][c];
}
tin.close();
```

Σε όλα αυτά που είπαμε δεν βάλαμε ελέγχους για να αναδείξουμε αυτά που μας ενδιαφέρουν εδώ. Στη συνέχεια θα τα ξαναδείς με τους ελέγχους τους (τους ελέγχους για την ανάγνωση από το αρχείο τους ξέρεις καλά.)

Και πώς ανακυκλώνουμε τη μνήμη όταν δεν τη χρειαζόμαστε; Πρώτα ανακυκλώνουμε τις γραμμές:

```
for ( int r(0); r < nR; ++r ) delete[] a[r];
```

και μετά τον πίνακα των βελών:

```
delete[] a;
```

Αφού όλα αυτά διαφέρουν μόνον ως προς τον τύπο των στοιχείων του πίνακα μπορούμε να τα βάλουμε σε δύο περιγράμματα. Εδώ θα βάλουμε και ελέγχους και καλό είναι να τους προσέξεις. Θα χρησιμοποιήσουμε μια δομή εξαιρέσεων που είναι κάτι σαν:

```
struct MyTmpltLibXptn
{
   enum {  domainError, noArray, allocFailed  };

   char funcName[100];
```

```
   int  errCode;
   int  errVal1, errVal2;
   MyTmpltLibXptn( const char* fn, int ec,
                   int erv1 = 0, int erv2 = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errVal1 = erv1;  errVal2 = erv2;  }
}; // MyTmpltLibXptn
```

Θα τη δούμε σε επόμενο κεφάλαιο.

Το πρώτο περίγραμμα –το ονομάζουμε *new2d*– τροφοδοτείται με τα πλήθη γραμμών και στηλών και επιστρέφει βέλος τύπου *T*`**` (*T* ο τύπος-παράμετρος):

```
 0: template< typename T >
 1: T** new2d( int nR, int nC )
 2: {
 3:    if ( nR <= 0 || nC <= 0 )
 4:       throw MyTmpltLibXptn( "new2d",
 5:                             MyTmpltLibXptn::domainError,
 6:                             nR, nC );
 7:    T** fv;
 8:    try {  fv = new T*[nR];  }
 9:    catch( bad_alloc& )
10:    {  throw MyTmpltLibXptn( "new2d",
11:                             MyTmpltLibXptn::allocFailed );  }
12:    for ( int r(0); r < nR; ++r )
13:    {
14:       try {  fv[r] = new T[nC];  }
15:       catch( bad_alloc& )
16:       {
17:          for ( int k(0); k < r; ++k )  delete[] fv[k];
18:          delete[] fv;
19:          throw MyTmpltLibXptn( "new2d",
20:                                MyTmpltLibXptn::allocFailed );
21:       } // catch
22:    } // for ( int r
23:    return fv;
24: } // new2d
```

Στη γρ. 8 προσπαθούμε να πάρουμε μνήμη για τα βέλη των γραμμών· αν δεν τα καταφέρουμε ρίχνουμε εξαίρεση και τελειώσαμε. Στη γρ. 14, που είναι μέσα στην περιοχή επανάληψης της **for** (γρ. 12) προσπαθούμε να πάρουμε μνήμη για τη γραμμή *r*. Μέχρι το σημείο αυτό έχουμε πάρει ήδη μνήμη για τα βέλη των γραμμών αλλά και για τις γραμμές 0 .. *r*-1. Αν δεν μπορέσουμε να πάρουμε μνήμη για τη γραμμή *r* πρέπει –πριν ρίξουμε την εξαίρεση– να ανακυκλώσουμε όλη αυτή τη μνήμη που ήδη πήραμε. Αυτό ακριβώς κάνουμε στις γρ. 17 και 18.

Η *delete2d()* είναι πολύ πιο απλή:

```
template< typename T >
void delete2d( T**& a, int nR )
{
   if ( a == 0 && nR > 0 )
      throw MyTmpltLibXptn( "delete2d", MyTmpltLibXptn::noArray );
   for ( int r(0); r < nR; ++r )  delete[] a[r];
   delete[] a;  a = 0;
} // delete2d
```

Πρόσεξε μόνον τον τύπο της πρώτης παραμέτρου: "`T**&`" διπλό βέλος αναφοράς! Εντυπωσιακό μεν αλλά απολύτως φυσικό: το a είναι ένα διπλό βέλος του οποίου η τιμή αλλάζει μέσα στη συνάρτηση.

Πρόσεξε ότι κάνουμε και αυτό που δεν κάνει η C++: μηδενίζουμε το βέλος μετά την ανακύκλωση του στόχου του.

Με αυτά τα εργαλεία, το παράδειγμα που ξεκινήσαμε γράφεται:

```
   tin >> nR >> nC;
   a = new2d<double>( nR, nC );
```

```
   for ( int r(0); r < nR; ++r )
   {
      for ( int c(0); c < nC; ++c ) tin >> a[r][c];
   }
   tin.close();
// . . .
   delete2d( a, nR );
```

Πρόσεξε ότι η *new2d()* δεν έχει κάποια παράμετρο με βάση την οποία θα μπορούσε να γίνει αυτόματη εξειδίκευση. Έτσι η κλήση θα πρέπει να γίνει υποχρεωτικώς ως: "`new2d<double>`".

Για να συγκρίνεις τους δυναμικούς δισδιάστατους πίνακες με τους συμβατικούς θα γράψουμε για τέταρτη (!) φορά το πρόγραμμα πολλαπλασιασμού πινάκων που πρωτοείδαμε στο Παράδ 4 της §12.4 (χωρίς συναρτήσεις), ξαναείδαμε στο Παράδ. 6 της §13.9.3 (συνάρτηση με παράμετρο τον «γραμμοποιημένο» πίνακα) και τέλος στην §14.7.1, όπου δείξαμε και ένα τέχνασμα με τις παραμέτρους περιγραμμάτων.

Τώρα θα το δούμε με δυναμικούς πίνακες:

```
   const int l( 3 ),  m( 5 ),  n( 2 );

   int** a( new2d<int>(l, m) );
   int** b( new2d<int>(m, n) );
   int** d( new2d<int>(l, n) );
```

Πρόσεξε πώς γράφονται τώρα οι συναρτήσεις εισόδου/εξόδου των πινάκων:

```
void input2DAr( istream& tin, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
} // input2Dar

void output2DAr( ostream& tout, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Ο πίνακας περνάει στη συνάρτηση με: (διπλό) βέλος, πλήθος γραμμών, πλήθος στηλών. Μέσα στις συναρτήσεις χρησιμοποιείται με τον συνήθη συμβολισμό των δισδιάστατων πινάκων (χωρίς «γραμμοποιήσεις»).

Οι συναρτήσεις καλούνται ως εξής: Διαβάζουμε με τις:

```
   input2DAr( atx, a, l, m );
   input2DAr( atx, b, m, n );
```

και γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr( cout, a, l, m );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr( cout, b, m, n );
   cout << " Στοιχεία του πίνακα d" << endl;
   output2DAr( cout, d, l, n );
```

Στο τέλος, ανακυκλώνουμε τη μνήμη που πήραμε με τις:

```
   delete2d( a, l );
   delete2d( b, m );
   delete2d( d, l );
```

Για όποιον δεν το πρόσεξε να το επισημάνουμε: *Οι γραμμές ενός δισδιάστατου δυναμικού πίνακα δεν είναι απαραίτητο να έχουν το ίδιο μήκος!* Ας δούμε ένα

Παράδειγμα

Θέλουμε έναν δυναμικό δισδιάστατο πίνακα που κάθε γραμμή του θα έχει το όνομα ενός μήνα. Δηλώνουμε:

```
   char** monthName( 0 );
```

και παίρνουμε μνήμη ως εξής:

```
   monthName = new char*[12];

   monthName[0] = new char[strlen("January")+1];
   strcpy( monthName[0], "January" );
   monthName[1] = new char[strlen("February")+1];
   strcpy( monthName[1], "February" );
// . . .
   monthName[10] = new char[strlen("November")+1];
   strcpy( monthName[10], "November" );
   monthName[11] = new char[strlen("December")+1];
   strcpy( monthName[11], "December" );
```

Όπως βλέπεις, για κάθε μήνα παίρνουμε ακριβώς τη μνήμη που χρειαζόμαστε (το "`+1`" για τον φρουρό).

Όταν θέλουμε να ανακυκλώσουμε τη μνήμη, η

```
   delete2d( monthName, 12 );
```

κάνει τη δουλειά μια χαρά.

Παρατήρηση:

Αυτά για το παράδειγμα. Σε πραγματικές καταστάσεις. πολύ πιο βολική λύση είναι η

```
   string* sMonthName( new string[12] );
// . . .
   delete[] sMonthName;
```

Αν πάρουμε υπόψη μας ότι σε μια τιμή *string* το κείμενο αποθηκεύεται σε δυναμική μνήμη, έχουμε και πάλι δυναμικό πίνακα δυναμικών πινάκων.

16.10 * Τύπος Βέλους: "`void*`"

Η C δεν υποστηρίζει περιγράμματα συναρτήσεων και για να διευκολύνει τους προγραμματιστές στην προσπάθειά τους να γράψουν συναρτήσεις ανεξάρτητες από τύπο (δίνουμε ένα παράδειγμα στη συνέχεια) δίνει και άλλο ένα εργαλείο: τον τύπο γενικού βέλους "`void*`".

Σε ένα βέλος αυτού του τύπου μπορείς να δώσεις ως τιμή τη διεύθυνση αντικειμένου οποιοδήποτε τύπου. Για παράδειγμα:

```
   int iv;
   void* pv1( &iv );
   double dv;
   pv1 = &dv;
   Date dt;
   pv1 = &dt;
```

Η αντίστροφη εκχώρηση δεν μπορεί να γίνει χωρίς τυποθεώρηση, π.χ.:

```
   int* pInt( reinterpret_cast<int*>(pv1) );
```

Στα βέλη `void*` δεν μπορούμε να κάνουμε αποπαραπομπή αν προηγουμένως δεν κάνουμε ερμηνευτική τυποθεώρηση (`reinterpret_cast`).

Αναφέραμε τα παραπάνω μόνο για προετοιμασία για την επόμενη παράγραφο και δεν θα δεις να τα χρησιμοποιούμε ξανά.

Ας δούμε ένα παράδειγμα γενικής συνάρτησης της C ώστε να δεις και χρήση του "`void*`" αλλιώτικο από αυτό που θα δεις στην επόμενη παράγραφο. Βάζοντας στο πρόγραμμά σου "`#include <cstdlib>`" μπορείς να χρησιμοποιήσεις τη συνάρτηση της C

```
void qsort( void* base, size_t num, size_t size,
            int (*compar)(const void*, const void*) );
```

που ταξινομεί –με τον αλγόριθμο Quicksort– *num* στοιχεία ενός πίνακα ξεκινώντας από αυτό που δείχνει το βέλος *base*. Η παράμετρος *size* περνάει το μέγεθος (σε ψηφιολέξεις) ενός στοιχείου του πίνακα. Η τελευταία παράμετρος είναι (βέλος προς) μια συνάρτηση. Όταν καλείται από την *qsort()* επιστρέφει:

- *Αρνητική* τιμή αν το στοιχείο που δείχνει η *πρώτη* παράμετρος *προηγείται* του στοιχείου που δείχνει η δεύτερη παράμετρος.
- *Μηδέν* αν το στοιχείο που δείχνει η *πρώτη* παράμετρος είναι *ίσο* με το στοιχείο που δείχνει η δεύτερη παράμετρος.
- *Θετική* τιμή αν το στοιχείο που δείχνει η *δεύτερη* παράμετρος *προηγείται* του στοιχείου που δείχνει η πρώτη παράμετρος.

Εδώ βλέπεις έναν άλλον τρόπο να γράφεις γενικές συναρτήσεις: με διευθύνσεις (βέλη) και μεγέθη περιοχών στη μνήμη. Έχουμε δηλαδή προγραμματισμό χαμηλού επιπέδου, που, όπως έχουμε πει, είναι δύσκολος. Πάντως υπάρχει και ένα (μικρό) κέρδος: ενώ, όπως είπαμε, ο μεταγλωττιστής θα δημιουργήσει μια συνάρτηση για κάθε στιγμιότυπο του περιγράμματος που θα βρει στο πρόγραμμά μας, η *qsort()* –και οποιαδήποτε συνάρτηση γραμμενη με αυτόν τον τρόπο– θα μεταγλωττισθεί μια φορά μόνο. Για κάθε κλήση της θα πρέπει να γράψουμε μια (το πολύ) συνάρτηση *compar()*.

Ας πούμε λοιπόν ότι σε κάποιο πρόγραμμα έχουμε:

```
int    intArr[ 50 ];
double dblArr[ 100 ];
```

και θέλουμε να ταξινομήσουμε ολόκληρον τον dblArr, κατ' *αύξουσα* τάξη, με χρήση της *qsort()*. Θα την καλέσουμε ως εξής:

```
qsort( dblArr, 100, sizeof(double), compareDblInc );
```

Το πρώτο όρισμα θα μπορούσε, αντί για "`dblArr`", να είναι "`&dblArr[0]`". Σε κάθε περίπτωση αυτό αντιγράφεται στο *base*. Πώς θα είναι η *compareDblInc()*; Έτσι:[5]

```
int compareDblInc( const void* a, const void* b )
{
   return ( *reinterpret_cast<const double*>(a) -
            *reinterpret_cast<const double*>(b) );
} // compareDblInc
```

Αφού θέλουμε ταξινόμηση κατ' αύξουσα τάξη το "`7.1`" προηγείται του "`11.7`". Έτσι η κλήση "`compareDblInc(&dblArr[35],&dblArr[71])`", όπου τα δύο στοιχεία έχουν αντιστοίχως τις παραπάνω τιμές, θα επιστρέψει τιμή "`-4`" (`== int(-4.6)`). Η *qsort()* θα κάνει πολλές τέτοιες κλήσεις.

Αν θέλουμε να ταξινομήσουμε το κομμάτι `intArr[11]` μέχρι `intArr[20]` κατά *φθίνουσα* τάξη θα πρέπει να γράψουμε:

```
qsort( &intArr[11], 10, sizeof(int), compareIntDec );
```

όπου:

```
int compareIntDec( const void* a, const void* b )
{
   return ( *reinterpret_cast<const int*>(b) -
            *reinterpret_cast<const int*>(a) );
} // compareIntDec
```

Πρόσεξε ότι με αυτήν το "`11`" προηγείται του "`7`".

---

[5] Στη C θα γράφανε: "`return ( *(double*)a - *(double*)b );`".

16.11 * Αναμνήσεις από τη C: *malloc(), free(), realloc()*

Η C++, έχοντας τη C ως υποσύνολό της, δίνει τη δυνατοτητα διαχείρισης της δυναμικής μνήμης και με τις συναρτήσεις *malloc()* (*calloc()*), *free()* και *realloc()*. Επειδή είναι πολύ πιθανό να τις συναντήσεις σε προγράμματα αξίζει να τις δούμε, έστω και εν συντομία. Για να τις χρησιμοποιήσεις θα πρέπει να περιλάβεις στο πρόγραμμά σου το `cstdlib`.

Η επικεφαλίδα της *malloc()* είναι:

```
void* malloc( size_t size );
```

Το "`void*`" λέει ότι η συνάρτηση επιστρέφει βέλος χωρίς τύπο. Μέσω της παραμέτρου πρέπει να περάσουμε την ποσότητα της ζητούμενης μνήμης σε ψηφιολέξεις. Αν έχεις δηλώσει:

```
   int* pInt;
```

και ζητήσεις:

```
   pInt = reinterpret_cast<int*>( malloc(72*sizeof(int) );
```

θα σου παραχωρηθεί που χωράει 72 τιμές τύπου `int` (πίνακας με 72 στοιχεία τύπου `int`) Το βέλος *pInt* δείχνει την αρχή της μνήμης που παραχωρήθηκε. Αν δεν υπάρχει διαθέσιμη η μνήμη που ζητείται η τιμή που επιστρέφεται είναι 0 (μηδέν).

Δεν είναι παράνομο να καλέσεις τη *malloc()* με όρισμα 0 αλλά το αποτέλεσμα εξαρτάται από την εγκατάσταση:

- Μπορεί να σου επιστρέψει βέλος 0.
- Μπορεί να σου επιστρέψει μη μηδενικό βέλος στο οποίο όμως δεν μπορείς να κάνεις αποπαραπομπή.

Παρόμοια με τη *malloc()* είναι η:

```
void* calloc( size_t nItems, size_t size );
```

Αυτή ταιριάζει πιο πολύ σε πίνακες: σου επιτρέπει να ζητήσεις *nItems* θέσεις μνήμης που κάθε μια τους θα πιάνει *size* ψηφιολέξεις. Το παραπάνω παράδειγμα θα μπορούσε να γραφεί:

```
   pInt = reinterpret_cast<int*>( calloc(72, sizeof(int) );
```

Όπως καταλαβαίνεις, αυτές οι δύο συναρτήσεις παίζουν τον ρόλο του τελεστή `new`.

Τον ρόλο του `delete` τον παίζει η συνάρτηση

```
void free( void* block );
```

που επιστρέφει στον σωρό τη μνήμη που δείχνει το βέλος *block*.

Πολύ συχνά, όταν δουλεύουμε με δυναμικούς πίνακες, έχουμε ανάγκη να μεγαλώσουμε (ή και να μικρύνουμε) κάποιον τον πίνακα. Για την περίπτωση αυτήν η C++ (C) μας δίνει τη συνάρτηση

```
void* realloc( void* block, size_t size );
```

που επεκτείνει ή συρρικνώνει το κομμάτι δυναμικής μνήμης που δείχνει το βέλος *block* σε *size* ψηφιολέξεις. Συνήθως αυτό γίνεται με παραχώρηση νέου τμήματος μνήμης στο οποίο η *realloc()* φροντίζει να αντιγράψει το περιεχόμενο του αρχικού τμήματος.

Το παρακάτω παράδειγμα δείχνει τη χρήση των συναρτησεων.

```
 0: #include <iostream>
 1: #include <cstdlib>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    int*    ip( 0 );
 8:    double* dp( 0 );
 9:
10:    ip = reinterpret_cast<int*>( malloc(3*sizeof(int)) );
11:    dp = reinterpret_cast<double*>( calloc(3,sizeof(double)));
12:    for ( int k(0); k <= 2; ++k )
```

```
13:    {
14:       ip[k] = k+1;
15:       dp[k] = 1.1*ip[k];
16:    }
17:    for ( int k(0); k <= 2; ++k )
18:       cout << ip[k] << "  " << dp[k] << endl;
19:    cout << "==========" << endl;
20:    ip = reinterpret_cast<int*>( realloc(ip, 7*sizeof(int)) );
21:    dp = reinterpret_cast<double*>(
22:                               realloc(dp, 7*sizeof(double)) );
23:    for ( int k(3); k <= 6; ++k )
24:    {
25:       ip[k] = k+1;
26:       dp[k] = 1.1*ip[k];
27:    }
28:    for ( int k(0); k <= 6; ++k )
29:       cout << ip[k] << "  " << dp[k] << endl;
30:    free( dp );
31:    free( ip );
32: } // main
```

Αποτέλεσμα:

```
1  1.1
2  2.2
3  3.3
==========
1  1.1
2  2.2
3  3.3
4  4.4
5  5.5
6  6.6
7  7.7
```

Αρχικώς, με τις *malloc()* και *calloc()*, πήραμε μνήμη για πίνακες με τρία στοιχεία (γρ. 10-11). Στη συνέχεια, με τη *realloc* (γρ. 20-22) επεκτείναμε τους πίνακες σε επτά στοιχεία για τον καθένα. Στο τέλος, με τη *free()* (γρ. 30-31), επιστρέφουμε τη μνήμη που πήραμε.

Πριν τελειώσουμε να πούμε ότι είναι *σαφώς προτιμότερη η χρήση των* `new` *και* `delete` και να επαναλάβουμε ότι, σε κάθε περίπτωση:

- ***Μνήμη που παίρνεις με*** `new` ***θα απελευθερώνεται με*** `delete` ***και μνήμη που παίρνεις με malloc(), calloc(), realloc() θα απελευθερώνεται με free().***

και ακόμη περίσσότερο:

- ***Μη χρησιμοποιείς στο ίδιο πρόγραμμα τις συναρτήσεις δυναμικής μνήμης της C και τους*** `new` ***και*** `delete`***.***

Αυτό βέβαια δεν εύκολο όταν μέσα στο πρόγραμμά σου χρησιμοποιείς βιβλιοθήκες ή έτοιμα κομμάτια προγράμματος. Στην περίπτωση αυτή περιορίσου στην τήρηση του πρώτου κανόνα.

Τέλος, να πούμε κάτι που μπορεί να συμβαίνει και να κάνει τα παραπάνω να φαίνονται περίεργα: Σε ορισμένες υλοποιήσεις της C++ οι `new` και `delete` υλοποιούνται με τις *malloc()* και *free()*.

- Αν αναπτύξεις μια εφαρμογή σε ένα τέτοιο περιβάλλον μπορεί να μην έχεις πρόβλημα όσο και αν ανακατέψεις τους τελεστές της C++ με τις συναρτήσεις της C.
- Σε μια τέτοια εγκατάσταση μπορεί το πρόβλημα της κλήσης "`malloc(0)`" να μεταφερθεί και στον τελεστή `new`.

16.12 Για να Μη Ζηλεύουμε τη *realloc()*

Όπως είπαμε και παραπάνω, θα πρέπει να προτιμάς τους **new** και **delete** που είναι πιο βολικοί από τις συναρτήσεις. Βέβαια υπάρχει και η *realloc*, που χρειάζεται αρκετά συχνά. Αργότερα θα μάθουμε ότι η C++ μας δίνει πολύ καλά εργαλεία[7], όπως το *vector*, το *map*, το *list*, που σου επιτρέπουν να έχεις δυναμικές δομές δεδομένων χωρίς να ανησυχείς για "**new**" και "**delete**".

Προς το παρόν όμως το πρόβλημα μπορεί να σου το λύσει το παρακάτω περίγραμμα συνάρτησης, που

- παίρνει το βέλος (*p*) προς έναν δυναμικό πίνακα με στοιχεία τύπου *T* και
- το αλλάζει σε βέλος προς νέον δυναμικό πίνακα με *nf* στοιχεία του ίδιου τύπου. Στα πρώτα *ni* στοιχεία του νέου πίνακα αντιγράφονται τα πρώτα *ni* στοιχεία του παλιού.

Η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*domainError*)[8] αν κληθεί χωρίς να ισχύει η συνθήκη $0 \le ni \le nf$.

Η συνάρτηση θα δεχθεί βέλος 0 (μηδέν) αρκεί να μην έχει θετική τιμή η *ni*. Αλλιώς (*p* == 0 && *ni* > 0) θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*noArray*).

Φυσικά, η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(::*allocFailed*) αν δεν μπορέσει να πάρει την απαραίτητη μνήμη.

```
template< typename T >
void renew( T*& p, int ni, int nf )
{
   if ( ni < 0 || nf < ni )
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::domainError, ni, nf );
// 0 <= ni <= nf
   if ( p == 0 && ni > 0 )
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::noArray );
// (0 <= ni <= nf) && (p != 0 || ni == 0)
   try
   {
      T* temp( new T[nf] );
      for ( int k(0);  k < ni; ++k )  temp[k] = p[k];
      delete[] p;   p = temp;
   }
   catch( bad_alloc& )
   {
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed );
   }
} // void renew
```

Παρατηρήσεις: ▶

1. Οτιδήποτε και να πάει στραβά, αν ριχτεί εξαίρεση, δεν θα πειραχτεί ο αρχικός πίνακας.
2. Η *bad_alloc* μπορεί να προέλθει όχι μόνο από τη "**new T[nf]**" αλλά και από τις αντιγραφές "**temp[k] = p[k]**". Αυτό μπορεί να γίνει αν ο *T* έχει αντικείμενα που χρησιμοποιούν δυναμική μνήμη, όπως λέγαμε στην §16.8. Θα πρέπει να κάνουμε κάτι σαν αυτό που κάναμε στη *new2d*. Θα το μάθουμε αργότερα. ◀

Στη συνέχεια, ξαναδίνουμε το παράδειγμα που είδαμε πιο πάνω αλλά με τα εργαλεία της C++:

```
#include <iostream>
#include <new>

using namespace std;

template< typename T >
void renew( T*& p, int ni, int nf );
```

[7] Πρόκειται για περιγράμματα κλάσεων.

[8] Είδαμε τη *MyTmpltLibXptn* πιο πριν, στην §16.9.

```
int main()
{
   int*    ip( 0 );
   double* dp( 0 );

   ip = new int [3];  dp = new double [3];
   for ( int k = 0; k <= 2; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 2; ++k )
      cout << ip[k] << "  " << dp[k] << endl;
   cout << "==========" << endl;
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
   for ( int k = 3; k <= 6; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 6; ++k )
      cout << ip[k] << "  " << dp[k] << endl;
   delete [] dp;
   delete [] ip;
} // main
```

Η `renew(ip, 3, 7)` αλλάζει την τιμή του βέλους προς έναν δυναμικό πίνακα με 7 στοιχεία τύπου `int`. Στα 3 πρώτα στοιχεία αυτού του πίνακα αντιγράφονται τα στοιχεία του πίνακα που έδειχνε το *ip* πριν από την κλήση της συνάρτησης. Παρόμοια κάνει και η `renew(dp, 3, 7)` στον dp.

Η C++ ελαχιστοποιεί την ανάγκη για χρήση της *realloc* με το περίγραμμα κλάσης *vector* που υπάρχει στην πάγια βιβλιοθήκη περιγραμμάτων (Standard Template Library, STL).

16.13 Παραδείγματα

Θα δώσουμε τώρα τρία παραδείγματα χρήσης δυναμικής μνήμης που είναι τρεις παραλλαγές του δεύτερου προγράμματος (§15.14.2) της §15.14. Πριν προχωρήσουμε όμως θα πρέπει να τονίσουμε ότι *τα παραδείγματα δίνονται επειδή κρίνονται κατάλληλα για την επίδειξη χρήσης όσων είπαμε παραπάνω. Σε καμιά περίπτωση δεν εννοούμε ότι αυτές οι λύσεις είναι καλύτερες από την αρχική.*

Παράδειγμα 1

Αντί να διαβάζουμε τα δεδομένα του κάθε χημικού στοιχείου που μας ζητείται και να τα φυλάγουμε μετά τη συμπλήρωση/διόρθωση του στοιχείου αυτού

- *διαβάζουμε ολόκληρο το αρχείο σε έναν δυναμικό πίνακα,*
- *κάνουμε όλες τις ενημερώσεις στον πίνακα και*
- *φυλάγουμε στο αρχείο τον ενημερωμένο πίνακα.*

Αυτό το πρόγραμμα είναι απλούστερο από το αρχικό. Δηλώνουμε:

```
   GrElmn* grElmnTbl( 0 );
```

και μόλις μάθουμε τον μέγιστο ατομικό αριθμό –που είναι ίσος με το πλήθος των στοιχείων– ζητούμε την απαραίτητη μνήμη:

```
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
```

Για την περίπτωση αποτυχίας προσθέτουμε, μετά την ομάδα **try**, άλλη μια:

```
   catch ( bad_alloc& )
   {
      cout << "not enough memeory to load data" << endl;
   }
```

Στη συνέχεια φορτώνουμε το περιεχόμενο του αρχείου στον πίνακα με την:

```
      loadAllData( bInOut, grElmnTbl, maxAtNo );
```

όπου:

```
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekg( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_load( grElmnTbl[k], bInOut );
} // loadAllData
```

Η *editGrName* δεν μας βολεύει πια. Την τροποποιούμε:

```
void editGrNameMM( GrElmn& a )
{
   GrElmn_display( a, cout );
   string newGrName;
   cout << "new greek name: "; getline( cin, newGrName, '\n' );
   if ( !newGrName.empty() )
   {
      GrElmn_setGrName( a, newGrName );
   }
} // editGrNameMM
```

Όπως βλέπεις, δεν έχει πια στις παραμέτρους το ρεύμα απο/προς το αρχείο αφού η δουλειά γίνεται στην κύρια μνήμη (βάλαμε το "*MM*" για να μας θυμίζει τη «Main Memory»).

Την καλούμε, αφού πρώτα διαβάσουμε το ατομικό αριθμο, ως εξής:

```
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
```

(θυμίσου ότι το στοιχείο με ατομικό αριθμό *aa* βρίσκεται στη θέση *aa*-1 του πίνακα.)

Τελικώς, φυλάγουμε τον πίνακα με την:

```
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekp( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_save( grElmnTbl[k], bInOut );
} // saveAllData
```

Οι συναρτήσεις *readRandom()* και *writeRandom()* δεν μας χρειάζονται. Οι *loadAllData()* και *saveAllData()* επεξεργάζονται το αρχείο ως σειριακό.

Το πρόγραμμά μας θα είναι κάπως έτσι:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
struct GrElmn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
// ΝΕΕΣ ΣΥΝΑΡΤΗΣΕΙΣ
void editGrNameMM( GrElmn& a );
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );

int main()
```

```
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );

   try
   {
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
      loadAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
      } while ( aa != 0 );
      openFile( flNm, bInOut );
      saveAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch ( bad_alloc& )
   {
      cout << "not enough memory to load data" << endl;
   }
   catch( ApplicXptn& x )
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Όπως βλέπεις:

- μια φορά παίρνουμε μνήμη (`grElmnTbl = new GrElmn[maxAtNo]`),
- μια φορά ανακυκλώνουμε (`delete[] grElmnTbl`).

Έτσι, δεν υπάρχει λόγος να πιάσουμε τη *bad_alloc* και να ρίξουμε δική μας.

Παράδειγμα 2

Θα ξαναλύσουμε το προηγούμενο πρόβλημα με έναν άλλον τρόπο: *Θα κρατούμε σε πίνακα μόνο τις εγγραφές που ζητούνται από τον χρήστη και με το τέλος χρήσης του προγράμματος θα ενημερώνουμε το αρχείο.*

Η ουσιαστική διαφορά από την προηγούμενη περίπτωση είναι η εξής:

- Στο προηγούμενο πρόγραμμα με το άνοιγμα του αρχείου μπορούσαμε να υπολογίσουμε το μέγεθος του δυναμικού πίνακα.
- Τώρα το μέγεθος του δυναμικού πίνακα εξαρτάται από το πόσο θα δουλέψει ο χρήστης! Κάθε φορά που ο χρήστης αποφασίζει να διορθώσει ένα στοιχείο το μέγεθος του δυναμικού πίνακα θα αυξάνεται κατά 1.

Αυτό το τελευταίο μπορεί και να μην είναι έτσι ακριβώς. Ας πούμε, ότι ο χρήστης παίρνει το στοιχείο 74 (Tungsten), το μεταφράζει «Τουνγκστένιο» και μετά θυμάται ότι στα ελληνικά το λέμε «Βολφράμιο»! Τα δεδομένα για το στοιχείο αυτό βρίσκονται στον πίνακα

και δεν χρειάζεται να τα ξαναφορτώσει! Άρα μπορεί να κάνει και διορθώσεις χωρίς να βάζει και άλλα στοιχεία στον πίνακα.

Αυτό όμως βάζει μια άλλη απαίτηση: αναζήτηση στα στοιχεία του πίνακα με βάση τον ατομικό αριθμό. Αυτό μπορούμε να το αντιμετωπίσουμε: Θα πάρουμε τη *linSearch()* –όπως την κάναμε στην §12.2.1– και θα τη μετατρέψουμε σε περίγραμμα (§16.13.1). Αλλά, για τη χρησιμοποιήσουμε θα πρέπει να μπορούμε να κάνουμε εκχώρηση (τελεστής "=") και σύγκριση "!=". Θα πρέπει λοιπόν να επιφορτώσουμε τον τελεστή "!=" (και τον αντίθετό του "==") για τον *GrElmn*. Αυτό είναι πολύ απλό αν σκεφτούμε ότι αντικείμενα τύπου *GrElmn* είναι ίσα αν και μόνον αν έχουν ίδιο ατομικό αριθμό:

```
bool operator!=( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber != rhs.geANumber );  }

bool operator==( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber == rhs.geANumber );  }
```

Να σκεφτούμε τώρα πώς θα διαχειριστούμε τον δυναμικό πίνακα. Θα μπορούσαμε να καλούμε τη *renew()* κάθε φορά που ζητείται κάποιο στοιχείο που δεν υπάρχει στον πίνακα. Για σκέψου όμως τι σημαίνει αυτό:

- Όταν βάλουμε το δεύτερο στοιχείο θα κάνουμε μια αντιγραφή (του πρώτου).
- Όταν βάλουμε το τρίτο θα κάνουμε δύο αντιγραφές (του πρώτου και του δεύτερου).

κ.ο.κ. Έτσι, αν ο χρήστης, σε μια εκτέλεση του προγράμματος διορθώσει 12 διαφορετικά στοιχεία η *renew()* θα κάνει:

$$1 + 2 + 3 + \ldots + 11 = 66 \text{ αντιγραφές}$$

Η άλλη ακραία επιλογή είναι να κρατήσουμε χώρο για ολόκληρον τον πίνακα αλλά να χρησιμοποιήσουμε μόνο 12 από τις θέσεις του.

Ας εξετάσουμε μια ενδιάμεση περίπτωση: Να αυξάνουμε το μέγεθος τού πίνακα κατά πεντάδες. Έτσι, όταν πάρουμε τη δεύτερη πεντάδα θα έχουμε 5 αντιγραφές και όταν πάρουμε την τρίτη άλλες 10. Συνολικώς θα κάνουμε 15 αντιγραφές ενώ θα πάρουμε μόνον τρεις παραπάνω θέσεις.

Η λύση στο πρόβλημά μας είναι κάτι τέτοιο. Αλλά, θα αυξάνουμε κατά 5 ή κατά 10 ή κατά 80; Η απάντηση εξαρτάται από το συγκεκριμένο πρόβλημα κάθε φορά. Όπως είδες, με την υπόθεση εργασίας «*σε μια εκτέλεση του προγράμματος διορθώνει 12 διαφορετικά στοιχεία*», το «κβάντο» αύξησης 5 φαίνεται να δουλεύει μια χαρά. Αν λέγαμε ότι θα διορθωσει 35 με 40 στοιχεία ένα «κβάντο» 10 ή 15 θα ήταν πιο κατάλληλο.

Και πριν προχωρήσουμε παρακάτω να υπενθμίσουμε ότι η *linSearch* χρησιμοποιεί και φρουρό. Για να έχουμε, λοιπόν, *n* στοιχεία αποθηκευμένα και να κάνουμε και αναζητήσεις χρειαζόμαστε πίνακα με *n*+1 θέσεις.

Για να ανταποκριθούμε σε όλες τις απαιτήσεις που βάλαμε πιο πάνω θα πρέπει να δηλώσουμε για τον δυναμικό πίνακα τα εξής:

```
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων
```

και όταν πάρουμε για πρώτη φορά μνήμη:

```
      grElmnTbl = new GrElmn[incr];
      nReserved = incr;
'     nElmn = 0;
```

Κάθε φορά που θα θέλουμε να εισαγάγουμε ένα νέο στοιχείο –με ατομικό αριθμό *aa*– στον πίνακα θα δουλεύουμε ως εξής:

```
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
```

```
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
```

Πρόσεξε τα εξής σημεία:

- Αυξάνουμε το μέγεθος του πίνακα όταν *nElmn*+1 == *nReserved*. Το «**+1**» μας επιτρέπει να έχουμε μια διαθέσιμη θέση στο τέλος για να βάζει η *linSearch* τον φρουρό.
- Καλούμε τη *renew()* με την **renew(grElmnTbl, nElmn, nReserved+incr)**. Το νέο μέγεθος του πίνακα θα είναι **nReserved+incr**. Από τον αρχικό πίνακα θα αντιγραφούν τα **nElmn** στοιχεία.
- Το νέο στοιχείο διαβάζεται από το αρχείο στη θέση **nElmn** του πίνακα. Μετά την αύξηση της τιμής της **nElmn** η θέση θα είναι **nElmn-1**.

Όλα αυτά τα κρύβουμε μέσα σε μια συνάρτηση που σιγουρεύει ότι τα δεδομένα του στοιχείου που μας ενδιαφέρει βρίσκονται στον πίνακα:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos )
{
   GrElmn oneElmn( aa );
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = nElmn - 1;
   }
   else
      pos = lPos;
} // elmntInTable
```

Να δούμε τι γίνεται με αυτήν τη συνάρτηση:

- Κατ' αρχάς έχει πολλές (7) παραμέτρους. Οι τέσσερις από αυτές έχουν να κάνουν με τον δυναμικό πίνακα και οι τρεις από αυτές είναι αναφοράς. Πράγματι:
  - Είναι πολύ πιθανό να κάνει εισαγωγή νέου στοιχείου στον πίνακα οπότε μπορεί να αλλάξει το πλήθος στοιχείων *nElmn*.
  - Μπορεί να χρειαστεί να μεγαλώσει ο πίνακας –με κλήση της *renew*– οπότε αλλάζει το βέλος *grElmnTbl* και το πλήθος των δεσμεμένων στοιχείων *nReserved*.
- Πριν από οτιδήποτε άλλο, καλείται η *linSearch()* για να μας φέρει στην *lPos* τη θέση του στοιχείου, αν υπάρχει στον πίνακα. Η *linSearch()* περιμένει ότι η τιμή που θα αναζητήσει θα είναι του ίδιου τύπου με τα στοιχεία του πίνακα. Για αυτόν ακριβώς τον λόγο δηλώνουμε την *oneElmn*.
- Αφού η σύγκριση "**!=**" γίνεται με χρήση μόνο του ατομικού αριθμού, για να κάνουμε τη δουλειά μας αρκεί να βάλουμε *oneElmn.geANumber* τη τιμή της παραμέτρου *aa*, Αυτό μπορούμε να το κάνουμε:
  - Με την εντολή "**oneElmn.geANumber = aa**".
  - Με τη δήλωση της *oneElmn* αν έχουμε τον κατάλληλο δημιουργό. Προτιμήσαμε αυτή τη λύση και γράψαμε τον

```
struct GrElmn
{
// . . .
   GrElmn( int aAN=0 ) {  geANumber = aAN;  }
}; // GrElmn
```

- Αν έχουμε αυτόν τον δημιουργό δεν χρειάζεται να δηλώσουμε τη μεταβλητή *oneElmn* αλλά μπορούμε να καλέσουμε τη *linSearch()* ως εξής:

```
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, GrElmn(aa)));
```

- Είδαμε παραπάνω τι κάνουμε αν το στοιχείο που ζητάει ο χρήστης δεν υπάρχει στον πίνακα (*lPos* < 0). Στην περίπτωση αυτή τα δεδομένα φορτώνονται από το αρχείο στην τελευταία χρησιμοποιούμενη θέση του πίνακα και επιστρέφουμε ως τιμή της *pos* το `nElmn - 1`.
- Αν τα δεδομένα υπάρχουν στη θέση *lPos* αυτή επιστρέφεται ως τιμή της *pos*.

Όταν ο χρήστης τελειώσει τη δουλειά του το περιεχόμενο του πίνακα φυλάγεται στο αρχείο με την:

```
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
```

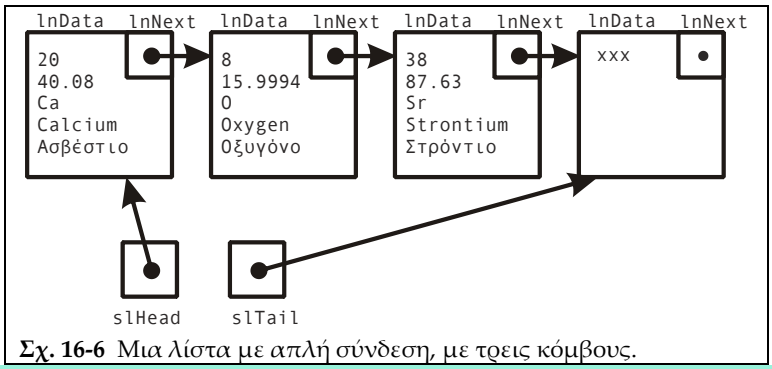
όπου:

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn )
{
   for ( int k(0); k < nElmn; ++k )
      writeRandom( grElmnTbl[k], bInOut );
} // saveUpdateTable
```

Τώρα το πρόγραμμα θα είναι ως εξής:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

template< typename T >
void renew( T*& p, int ni, int nf );
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x );

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
   GrElmn( int aAN=0 ) {  geANumber = aAN;  }
}; // GrElmn

// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
bool operator!=( const GrElmn& lhs, const GrElmn& rhs );
bool operator==( const GrElmn& lhs, const GrElmn& rhs );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
```

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn );
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos );

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων

   try
   {
      try {  grElmnTbl = new GrElmn[incr];  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "main", ApplicXptn::allocFailed );  }
      nReserved = incr;
      nElmn = 0;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            unsigned int pos;
            elmntInTable( grElmnTbl, nElmn, nReserved, incr,
                          bInOut, aa, pos );
            editGrNameMM( grElmnTbl[pos] );
         }
      } while ( aa != 0 );
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
      // ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
         case MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
```



Σχ. 16-6 Μια λίστα με απλή σύνδεση, με τρεις κόμβους.

```
            break;
         case MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στο πρόγραμμα αυτό παίρνουμε δυναμική μνήμη στη **main** (μια φορά) και στη *renew()*. Παρ' όλο που δεν υπάρχει περίπτωση σύγχυσης, αφού από τη *renew()* ρίχνονται εξαιρέσεις τύπου *MyTmpltLibXptn*, προτιμήσαμε να πιάνουμε και στη **main** τη *bad_alloc* και να ρίχνουμε μια δική μας *ApplicXptn*(::*allocFailed*).

Στη διαχείριση των εξαιρέσεων *ApplicXptn* προσθέσαμε μια επι πλέον περίπτωση *ApplicXptn*::*allocFailed*.

Παράδειγμα 3

Τώρα θα ξαναγράψουμε το πρόγραμμα του Παραδ. 2 με την εξής διαφορά: *Θα κρατούμε τις εγγραφές που ζητάει ο χρήστης όχι σε δυναμικό πίνακα αλλά σε μια λίστα με (απλή) σύνδεση.*

Η **λίστα με απλή σύνδεση** (simply linked list) –μια υλοποίηση της *ακολουθίας* (sequence)– είναι μια ευρύτατα χρησιμοποιούμενη δομή δεδομένων. Θα δούμε εδώ ένα τμήμα υλοποίησης μιας τέτοιας λίστας με λογική στοίβας.

Η παράσταση που θα υλοποιήσουμε φαίνεται στο Σχ. 16-6. Όπως βλέπεις, η λίστα είναι μια ακολουθία από *κόμβους* (nodes). Ο «κόμβος-φρουρός» στο τέλος απλουστεύει ορισμένους αλγόριθμους διαχείρισης της λίστας.

Κάθε κόμβος έχει δεδομένα (στην περίπτωσή μας τύπου *GrElmn*) και ένα βέλος-σύνδεσμο προς τον επόμενο κόμβο. Μπορούμε να τον υλοποιήσουμε στο πρόγραμμά μας με αντικείμενα τύπου:

```
struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode
```

Ένα βέλος δείχνει την αρχή της λίστας. Η ύπαρξη και δεύτερου βέλους που δείχνει τον φρουρό στο τέλος της λίστας απλουστεύει –όπως και ο ίδιος ο φρουρός, ορισμένους αλγόριθμους. Μπορούμε να πούμε λοιπόν ότι θα έχουμε:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

Στο Σχ. 16-7 βλέπεις μια κενή λίστα. Από αυτήν την εικόνα μπορούμε εύκολα να βγάλουμε τον ερήμην δημιουργό:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // Slist
```

Εδώ βλέπουμε ξανά να ρίχνεται εξαίρεση από δημιουργό. Βλέπουμε όμως και κάτι ακόμη: πώς παιρνουμε μνήμη για έναν κόμβο της λίστας. Με τον ίδιο τρόπο θα παίρνουμε και για τους υπόλοιπους κόμβους. Αλλά τι θα γίνει όταν θελήσουμε να ανακυκλώσουμε αυτή τη μνήμη; Και *η ανακύκλωση θα γίνει κομβο προς κόμβο*· δεν υπάρχει "`delete`" που να καθαρίζει όλη τη λίστα.

Να δούμε τώρα τι θέλουμε να κάνουμε με μια τέτοια λίστα:

- Εισαγωγή δεδομένων ενός στοιχείου στη λίστα (σε νέο κόμβο).
- Αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου.
- Φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.
- Ανακύκλωση όλων των κόμβων της λίστας.

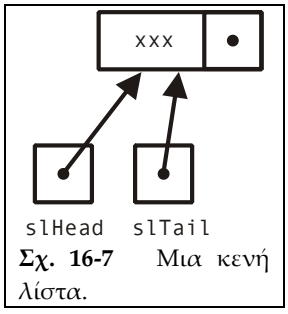
Ξεκινούμε με τη συνάρτηση εισαγωγής δεδομένων στη λίστα που δεν είναι και τόσο μπερδεμένη:[9]

```
void SList_push_front( SList& lst, const GrElmn& aData )
{
   ListNode* pnln( 0 );
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "SList_push_front",
                        ApplicXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = lst.slHead;
   lst.slHead = pnln;
} // SList_push_front
```

Ας δούμε τη λειτουργία της με ένα παράδειγμα από το «κτίσιμο» της λίστας του Σχ. 16-6: Έχουμε μια λίστα στην οποίαν υπάρχει μόνο το Στρόντιο και θέλουμε να εισαγάγουμε και το Οξυγόνο. Τα δεδομένα του Οξυγόνου υπάρχουν στην παράμετρο *aData*.

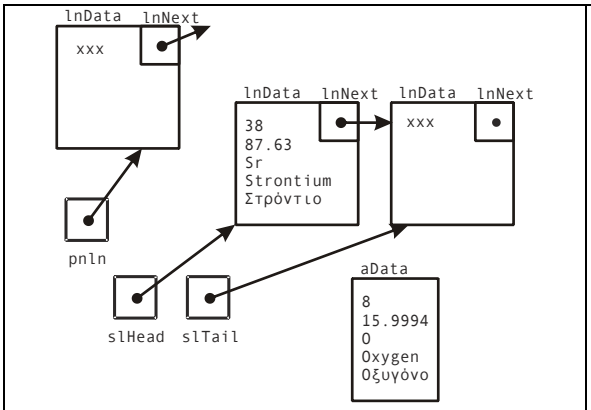
Αν προσπαθήσουμε να πάρουμε δυναμική μνήμη κάπως απρόσεκτα με μια "`lst.slHead = new ListNode`" *χάνουμε την αρχή της λίστας*. Επιλέγουμε λοιπόν να κάνουμε το εξής:

- Παίρνουμε μνήμη χρησιμοποιώντας ένα βοηθητικό βέλος, το *pnln* με την "`pnln = new ListNode`". Στο Σχ. 16-8α βλέπεις την

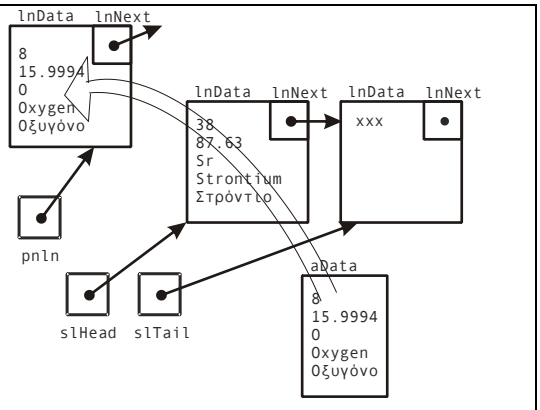


Σχ. 16-7 Μια κενή λίστα.

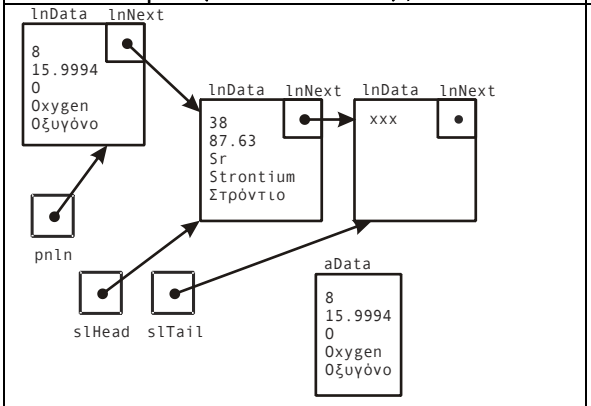
[9] Το όνομα από την STL.



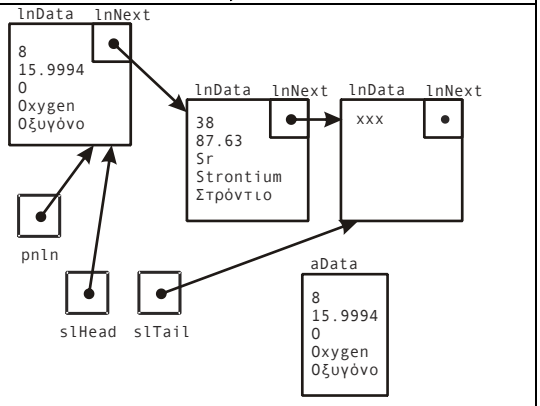
Σχ. 16-8α Μετά την εκτέλεση της εντολής:
`ListNode* pnln( new ListNode );`



Σχ. 16-8β Μετά την εκτέλεση της εντολής:
`pnln->lnData = aData;`



Σχ. 16-8γ Μετά την εκτέλεση της εντολής:
`pnln->lnNext = lst.slHead;`



Σχ. 16-8δ Μετά την εκτέλεση της εντολής:
`lst.slHead = pnln;`

κατάσταση που διαμορφώνεται. Η λίστα δεν έχει πειραχτεί ακόμη.

- Στον νέο κόμβο, στο μέλος *lnData* αντιγράφουμε τα δεδομένα που θέλουμε να εισαγάγουμε στη λίστα με την "`pnln->lnData = aData`". Η λίστα δεν έχει πειραχτεί ακόμη. Η κατάσταση φαίνεται στο Σχ. 16-8β.
- Με την "`pnln->lnNext = lst.slHead`" συμπληρώνεται ο νέος κόμβος. Το βέλος *lnNext* δείχνει τον μέχρι τώρα πρώτο κόμβο της λίστας. Όπως βλέπεις στο Σχ. 16-8γ έχουμε μια «νέα είσοδο» στη λίστα που κατά τα άλλα δεν έχει πειραχτεί.
- Με την "`lst.slHead = pnln`" ο νέος κόμβος ενσωματώνεται στη λίστα (Σχ. 16-8δ): Ξεκινώντας από την *slHead* περνάς από τον νέο κόμβο και από αυτόν πηγαίνεις στον προηγούμενο πρώτο (με το Στρόντιο).

Με το τέλος εκτέλεσης της συνάρτησης το βέλος *pnln* παύει να υπάρχει.

Σε δύο περιπτώσεις, μετά από αντιγραφές βελών, είχαμε δύο βέλη να δείχνουν τον ίδιο στόχο: στα Σχ. 16-8γ και 16-8δ, αλλά δεν υπήρχε οποιοδήποτε πρόβλημα.

Ο τρόπος αυτός δεν είναι ο μοναδικός· να και ένας άλλος το ίδιο σωστός:

```
ListNode* pnln( lst.slHead );
lst.slHead = new ListNode;
(lst.slHead)->lnData = aData;
(lst.slHead)->lnNext = pnln;
```

Η «*αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου*» μπορεί να γίνει στη λίστα όπως περίπου γίνεται στον πίνακα με τη *linSerch*:

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   ListNode* p( lst.slHead );
   while ( p->lnData != aData && p->lnNext != lst.slTail )
      p = p->lnNext;
   return (p->lnData == aData ? p : 0 );
```

```
} // SList_listSearch
```

Το βέλος *p* ξεκινάει δείχνοντας το στοιχείο που δείχνει και το *lst.slHead* και προχωρεί από στοιχείο σε στοιχείο με την "`p = p->lnNext`". Διασχίζει τη λίστα κόμβο προς κόμβο με την "`p = p->lnNext`" μέχρι

- Να βρει τα στοιχεία, αν υπάρχουν ("`p->lnData != aData`") ή
- Να φτάσει στον φρουρό ("`p->lnNext != lst.slTail`").

Αν έχει βρει την τιμή ("`p->lnData == aData`") η τιμή που επιστρέφει η συνάρτηση είναι *p* · αλλιώς επιστρέφει `0`.

Αν στον κόμβο-φρουρό αντιγράψουμε την τιμή που ψάχνουμε δεν θα απαλλαγούμε από τη μια σύγκριση της **while**; Βεβαίως, αυτή η τεχνική είναι η αντίστοιχη αυτής που μάθαμε στη γραμμική αναζήτηση σε πίνακα. Φυσικά, αυτό απαγορεύεται λόγω του "`const SList& lst`" αλλά ξέρουμε πώς θα αντιμετωπίσουμε: με τυποθεώρηση **const**.

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   SList& ncLst( const_cast<SList&>(lst) );
   ncLst.slTail->lnData = aData;
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
   return (p != lst.slTail ? p : 0 );
} // SList_listSearch
```

Η *ncLst* είναι η *lst* αλλά τύπου *SList*`&` –χωρίς "`const`"– και μπορούμε να δώσουμε:

```
   ncLst.slTail->lnData = aData;
```

Έτσι, η **while** γίνεται:

```
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
```

και η **return**:

```
   return (p != lst.slTail ? p : 0 );
```

(επίστρεψε το *p* αν δεν δείχνει τον φρουρό, αλλιώς το 0)

Η *linSearch()* επιστρέφει δείκτη στοιχείου πίνακα. Η *SList_listSearch* επιστρέφει βέλος προς κόμβο που έχει μέλος τύπου *GrElmn*. Έτσι, τώρα θα καλέσουμε την *editGrNameMM* αλλά κάπως διαφορετικά:

```
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
```

Εδώ είδαμε και πώς διασχίζουμε ολόκληρη τη λίστα: «*Το βέλος p ξεκινάει δείχνοντας το στοιχείο που δείχνει και το lst.slHead και προχωρεί από στοιχείο σε στοιχείο με την* "`p = p->lnNext`" *... μέχρι ... να φτάσει στον φρουρό* ("`p->lnNext != lst.slTail`").»

Πώς θα κάνουμε λοιπόν τη «*φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.*» Τη διασχίζουμε και φυλάγουμε το περιεχόμενο του κάθε κόμβου:

```
void saveUpdateList( fstream& bInOut, const SList& lst )
{
   for ( ListNode* p(lst.slHead); p != lst.slTail; p = p->lnNext )
      writeRandom( p->lnData, bInOut );
} // saveUpdateList
```

Παρομοίως γίνεται και η «*ανακύκλωση όλων των κόμβων της λίστας.*»

```
void SList_deleteAll( SList& lst )
{
   while ( lst.slHead != lst.slTail )
   {
      ListNode* p( lst.slHead );
      lst.slHead = (lst.slHead)->lnNext;
      delete p;
   }
   delete lst.slHead;
```

```
   lst.slTail = lst.slHead = 0;
} // SList_deleteAll
```

Πρόσεξε πώς γίνεται η διαγραφή του (εκάστοτε) πρώτου κόμβου:

- Φυλάγουμε στο *p* την τιμή του *lst.slHead* που δείχνει τον πρώτο κόμβο.
- Προχωρούμε το *lst.slHead* στον επόμενο κόμβο.
- Ανακυκλώνουμε αυτόν που δείχνει το *p*.

Αυτή η διαδικασία τελειώνει όταν "`lst.slHead == lst.slTail`": και τα δύο βέλη δείχνουν τον φρουρό. Η "`delete lst.slHead`" ανακυκλώνει και τον φρουρό.

Σε τι κατάσταση βρίσκεται η λίστα τώρα; Είναι άδεια; Όχι! *Δεν υπάρχει λίστα!* Δηλαδή δεν μπορείς να ξανακάνεις εισαγωγή στοιχείου με *SList_push_front()*. Αυτή η κατάσταση είναι ανιχνεύσιμη μετά την εκτέλεση των εκχωρήσεων:

```
   lst.slTail = lst.slHead = 0;
```

Να δούμε τώρα το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

// Συναρτήσεις GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode

struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // SList

void SList_push_front( SList& lst, const GrElmn& aData );
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData );
void SList_deleteAll( SList& lst );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
void saveUpdateList( fstream& bInOut, const SList& lst );
void elmntInList( SList& lst,
                  fstream& bInOut, int aa, ListNode*& pos );
```

```
int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      SList   lst;
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      SList_deleteAll( lst );
   }
   catch( ApplicXptn& x )
   // ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στις λίστες θα ξαναγυρίσουμε.

Πρόσεξε τώρα ένα σημαντικό πρόβλημα που έχουν αυτές οι τρεις λύσεις: *η καθυστέρηση που υπάρχει από τη στιγμή που κάνεις την ενημέρωση μέχρι να περάσει στο αρχείο.* Όταν λέμε «*καθυστέρηση*» την εννοούμε με όρους ανθρώπου και όχι υπολογιστή, π.χ. συμπληρώνω πέντε στοιχεία πάω για καφέ και τσιγάρο και επιστρέφω για τα υπόλοιπα. Αν λοιπόν συμβεί κάποιο είδος «system crash» χάνεις όλη τη δουλειά που έχεις κάνει εν τω μεταξύ (ενώ με το αρχικό πρόγραμμα το πολύ που μπορείς να χάσεις είναι η τελευταία εγγραφή).

Όμως, εκτός από αυτήν την περίπτωση, πρόσεξε ότι η *GrElmn_save()* μπορεί να ρίξει εξαίρεση *ApplicXptn*(::*cannotWrite*). Πώς μπορεί να γίνει αυτό; Για παράδειγμα, έχεις το αρχείο σε memory stick και κατά λάθος βγήκε από τη θέση του! Παρόμοια ζημιά μπορεί να γίνει –στα Παραδ. 2 και 3– και στην περίπτωση που η *GrElmn_load()* ρίξει εξαίρεση *ApplicXptn* (::*cannotRead*). Όπως είναι γραμμένα τα προγράμματα δεν υπάρχει πρόβλεψη για τέτοια καταστροφή. Για σκέψου όμως, κάτι θα μπορέσεις να σκαρφιστείς...

Ειρήσθω εν παρόδω ότι στις περιπτώσεις αυτές έχουμε και «διαρροή μνήμης» αλλά, αν έχεις χάσει όλη τη δουλειά που έκανες, μάλλον δεν σε ενδιαφέρει και τόσο.

16.13.1 Το Περίγραμμα *linSearch()*

Μετατρέπουμε σε περίγραμμα τη *linSearch* όπως την είδαμε στην §12.2.1

```
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
{
   if ( v == 0 && n > 0 )
      throw MyTmpltLibXptn( "linSearch",
                            MyTmpltLibXptn::noArray );
   int fv( -1 );
   if ( v != 0 && (0 <= from && from <= upto && upto < n) )
   {
      T* ncv( const_cast<T*>(v) );
      T  save( v[upto+1] );  // φύλαξε το v[upto+1]
      ncv[upto+1] = x;       // φρουρός
      int k( from );
      while ( v[k] != x )  ++k;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
      ncv[upto+1] = save;    // όπως ήταν στην αρχή
   // (from <= fv <= upto && v[fv] == x) ||
   //                (fv == -1 && (∀j:from..upto • v[j] != x))
   }
   return fv;
} // linSearch
```

Πέρα από τις αλλαγές από "`int`" σε "*T*" πρόσεξε τα εξής σημεία:

- Ελέγχουμε μήπως "`v == 0`". Στην περίπτωση αυτήν, αν το πλήθος στοιχείων του πίνακα *n* είναι 0 επιστρέφουμε τιμή "`-1`" ("δεν το βρήκα") αλλιώς ρίχνουμε εξαίρεση.
- Για να μπορεί να γίνει εξειδίκευση στον τύπο *T*, θα πρέπει να έχουμε για τον τύπο αυτόν:
  - Τελεστή εκχώρησης που να δουλεύει σωστά (αν δεν καταλαβαίνεις τι θα πει αυτό διάβασε, για παράδειγμα, αυτά που λέμε στην §16.8) για τις εκχωρήσεις "`ncv[upto+1] = x`" και "`ncv[upto+1] = save`". Στην περίπτωσή μας αυτός που υπάρχει αυτομάτως δουλεύει σωστά.
  - Τελεστή σύγκρισης "`!=`" για τη σύγκριση "`v[k] != x`" στη **while**.
  - *Δημιουργό αντιγραφής* που να δουλεύει σωστά! Τι είναι πάλι αυτό; Θα το μάθουμε αργότερα. Απλώς να πούμε ότι αυτός είναι που εκτελείται για τη δήλωση "`T save(v[upto+1])`".[10] Στην περίπτωσή μας και αυτός, όπως ο τελεστής εκχώρησης, υπάρχει αυτομάτως και δουλεύει σωστά.

Πάντως θα πρέπει να επιστήσουμε την προσοχή σου στις πολλές αντιγραφές που –αν τα αντικείμενα του τύπου *T* είναι μεγάλα– μπορεί να καθυστερούν την εκτέλεση του προγράμματός σου. Και να σκεφτείς ότι για τη σύγκριση χρειαζόμαστε μόνο το κλειδί, που συνήθως είναι πολύ πιο μικρό...

16.13.2 Χωρίς τη *linSearch()*

Η *linSearch* είναι ένα καλό εργαλείο από εκπαιδευτική άποψη, αλλά η C++ μας προσφέρει ένα περίγραμμα συνάρτησης –με το όνομα (`std::`)*find*– που κάνει την ίδια δουλειά: γραμμική αναζήτηση στα στοιχεία ενός πίνακα (και όχι μόνο). Θα το περιγράψουμε τώρα με τους περιορισμούς που υπάρχουν από αυτά που ξέρουμε μέχρι τώρα. Αργότερα, θα τη δούμε πληρέστερα.

Μπορείς, προς το παρόν, να σκέφτεσαι το περίγραμμα ως εξής:

```
template< typename T >
```

[10] Αν δεν θέλεις να ανησυχείς για δημιουργούς αντιγραφής και άλλα παρόμοια γράψε:

```
      T  save;
      save = v[upto+1];  // φύλαξε το v[upto+1]
```

```
T* find( T* first, T* last, T value )
```

όπου:

- Ο τύπος *T* έχει ορισμένη τη σύγκριση για ισότητα ("`==`").
- Η `[` *first*`..`*last* `)` είναι μια περιοχή βελών που μπορούμε να διασχίσουμε –ξεκινώντας από τη *first*– με μια μεταβλητή-βέλος `T* p` με την πράξη "`++p`". *Η last είναι η πρώτη θέση μετά την περιοχή αναζήτησης και δεν περιλαμβάνεται σε αυτήν.*
- *value* είναι η τιμή που αναζητούμε.

Η *find()* θα επιστρέψει βέλος προς την πρώτη θέση *fPos* για την οποίαν θα ισχύει η `*fPos == value`. Αν δεν βρει τη *value* επιστρέφει *last*.

Αν λοιπόν έχουμε έναν πίνακα ar –συμβατικό ή δυναμικό– με στοιχεία τύπου *T* για να ψάξουμε με τη *linSearch()* θα πρέπει να δώσουμε:

```
   unsigned int size, from, upto;
   int          ndx;
   T            value;
// . . .
   ndx = linSearch( ar, size, from, upto, value );
   if ( ndx >= 0 ) // βρέθηκε
   // . . .
```

ενώ για να ψάξουμε με τη *find* θα πρέπει να δώσουμε:

```
   unsigned int from, upto,;
   T*           pos;
   T            value;
// . . .
   pos = find( ar+from, ar+upto, value );//[11]
   if ( pos != ar+upto ) // βρέθηκε
   // . . .
```

Γυρνώντας στο Παράδ. 2, θα μπορούσαμε να γράψουμε τη *elmntInTable* ως εξής:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, GrElmn*& pos )
{
   GrElmn oneElmn( aa );
   GrElmn* lPos( find(grElmnTbl, grElmnTbl+nElmn, GrElmn(aa)) );
   if ( lPos == grElmnTbl+nElmn ) // αν δεν υπάρχει
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = &grElmnTbl[nElmn-1];
   }
   else
      pos = lPos;
} // elmntInTable
```

Να επισημάνουμε τις διαφορές:

- Η τελευταία παράμετρος είναι τώρα: `GrElmn*& pos`.
- Παρομοίως, η *lPos* είναι τύπου *GrElmn*`*`.
- Πρόσεξε πώς καλούμε τη *find* για να αναζητήσει τη `GrElmn(aa)` σε ολόκληρον τον πίνακα *grElmnTbl*:
  - Πρώτη παράμετρος ο πίνακας (η αρχή του),

[11] Ή, αν προτιμάς:

```
pos = find( &ar[from], &ar[upto], value );
```

- Δεύτερη παράμετρος μια θέση μετά το τελευταίο στοιχείο, Να υπενθυμίσουμε ότι: `grElmnTbl+nElmn` σημαίνει `&grElmnTbl[nElmn]`.

Μετά την κλήση ελέγχουμε το *αν δεν βρέθηκε η τιμή* ελέγχοντας τη συνθήκη *lPos* `==` *grElmnTbl*`+`*nElmn*.

Μπορούμε να χρησιμοποιήσουμε τη *find* για αναζητήσεις στη λίστα; Όχι για δύο λόγους:

- Δεν διασχίζουμε τη λίστα με την "`++p`" αλλά με την "`p = p->lnNext`".
- Αυτό που αναζητούμε δεν είναι ολόκληρος ο κόμβος αλλά μέλος του κόμβου.

Αργότερα θα γνωρίσουμε τα κατάλληλα εργαλεία για να το καταφέρουμε και αυτό.

16.13.3 "*reserved* + *incr*" ή "2**reserved*"

Ο τρόπος διαχείρισης της δυναμικής μνήμης δεν είναι ο καλύτερος. Ας πούμε ότι ξεκινούμε με *reserved* == *incr* στοιχεία στον πίνακα και τελικώς εισάγουμε *n* τιμές. Θα πάρουμε μνήμη *n*/*incr* φορές.[12] Την *k*-οστή φορά θα πάρουμε μνήμη για *k***incr* στοιχεία και θα κάνουμε (*k*-1)**incr* αντιγραφές.

Συνολικώς, το πλήθος των αντιγραφών θα είναι:

$$0*incr + 1*incr + 2*incr + \ldots + ((n/incr)-1)*incr = (n/incr)*((n/incr)-1)*incr/2 = O(n^2)$$

Στη βιβλιοθήκη της C++ (STL), κάθε φορά που χρειάζεται δυναμική μνήμη, παίρνει διπλάσια από αυτήν που ήδη έχει. Έτσι, αν ξεκινήσει με μνήμη για *incr* στοιχεία, την *k*-οστή φορά θα πάρει μνήμη για $2^{k-1}*incr$ στοιχεία και θα κάνει $2^{k-2}*incr$ αντιγραφές ($k \geq 2$: την πρώτη φορά δεν γίνονται αντιγραφές). Την τελευταία φορά θα έχουμε πάρει μνήμη για $n == 2^{p-1}*incr$ στοιχεία. Από αυτήν έχουμε: $p = log_2(n/incr)+1$.

Οι αντιγραφές που θα γίνουν:

$$2^0*incr + 2^1*incr + 2^2*incr + \ldots + 2^{p-2}*incr = incr*(2^{p-1}-1) = O(n)$$

Με αυτόν τον τρόπο δουλεύουν οι *παραχωρητές μνήμης* (memory allocators) των *περιεχόντων* (containers) της βιβλιοθήκης της C++.

Εμείς θα χρησιμοποιούμε τον «αργό τρόπο» μέχρι να γυρίσουμε στην STL, μια και όταν χρησιμοποιούμε δυναμική παραχώρηση μνήμης άλλα είναι αυτά που θα θέλουμε να δείξουμε. Πάντως, εσύ, αν θέλεις, μπορείς να χρησιμοποιείς τον «γρήγορο τρόπο». Πιο πάνω, στο Παράδ. 2, είδαμε τις εντολές:

```
renew( grElmnTbl, nElmn, nReserved+incr );
nReserved += incr;
```

Μπορείς να τις αλλάξεις σε:

```
renew( grElmnTbl, nElmn, 2*nReserved );
nReserved *= 2;
```

16.14 Προβλήματα Ασφάλειας

Στην παράγραφο αυτή θα ασχοληθούμε με δυο συστάσεις του (CERT 2009):

- ***Καθάρισε ευαίσθητες πληροφορίες αποθηκευμένες σε ανακυκλώσιμους πόρους που επιστρέφονται για να ξαναχρησιμοποιηθούν.***[13]

και

- ***Εξασφάλισε ότι δεν γράφονται στον δίσκο ευαίσθητα δεδομένα.***[14]

---

[12] Για να μη μπερδεύεσαι με ατελείς διαιρέσεις θεώρησε ότι $n = 2^N*incr$.

[13] Σύσταση *MEM*03: "Clear sensitive information stored in reusable resources returned for reuse."

[14] Σύσταση *MEM*06: "Ensure that sensitive data is not written out to disk."

Θα ξεκινήσουμε ξαναδίνοντας ένα παράδειγμα που είδαμε στην §16.3. Με τη βοήθεια δύο βελών:

```
   double* a1( new double );
   double* a2;
```

είχαμε υλοποιήσει δύο δυναμικές μεταβλητές *a1, *a2. Αφού τους δώσαμε τιμές, η

```
   cout << *a1 << "  " << *a2 << endl;
```

μας έδωσε:

```
1.23  4.56
```

Στη συνέχεια τις ανακυκλώσαμε:

```
   delete a1;  delete a2;
```

ξαναϋλοποιήσαμε την *a1 και είδαμε την τιμή της πριν προσπαθήσουμε να την ορίσουμε:

```
   a1 = new double;
   cout << *a1 << endl;
```

Αποτέλεσμα:

```
4.56
```

Πώς έγινε αυτό; Η *a1 υλοποιήθηκε τη δεύτερη φορά στη μνήμη που ελευθερώθηκε με την ανακύκλωση της *a2. Φυσικά, αυτό στηρίζεται στο ότι:

- Ο "`delete`", παρά το όνομά του, δεν κάνει οποιαδήποτε διαγραφή.
- Ο "`new`", παρά το όνομά του μπορεί να μας φέρει «παλιά» πράγματα.

Παρόμοια πράγματα μπορεί να συμβούν και με τη μνήμη στοίβας:

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

void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
}

void f2()
{
   char q[5];
   cout << "from f2:  q = " << q << endl;
}

int main()
{
   f1();
   f2();
}
```

Αυτό το πρόγραμμα θα δώσει:

```
from f1:  m = abcd
from f2:  q = abcd
```

Τι έγινε εδώ; Με την κλήση της *f1()* η τοπική μεταβλητή *m* υλοποιείται σε μνήμη που δίνεται από τη στοίβα και παίρνει τιμή `"abcd"`. Με το τέλος της εκτέλεσης της *f1()* η μνήμη αυτή απελευθερώνεται. Εδώ δεν έχουμε ούτε "`new`" ούτε "`delete`"· όλα γίνονται αυτομάτως. Όταν στη συνέχεια ενεργοποιείται η *f2()*, με τον τρόπο που λειτουργεί η στοίβα, η μνήμη που είχε δοθεί στην *m* της *f1()* δίνεται στην *q* της *f2()*.

Τι κοινό έχουν τα παραπάνω παραδείγματα; *Και στις δύο περιπτώσεις πήραμε μνήμη για να κάνουμε τη δουλειά μας και μετά την ελευθερώσαμε. Αλλά τα δεδομένα που αποθηκεύτηκαν έχουν παραμείνει.*

Αν τα δεδομένα είναι «ευαίσθητα», για παράδειγμα κάποια *συνθηματικά πρόσβασης* (passwords), θα θέλαμε να έχουμε τη σιγουριά ότι όταν δεν τα χρησιμοποιούμε στο πρόγραμμά μας δεν υπάρχουν στη μνήμη του υπολογιστή. Ένας απλός τρόπος να το σιγουρέ-

ψουμε είναι να σβήσουμε πριν ανακυκλωθούν. Υπάρχει μια πάγια τεχνική για τη δουλειά αυτή: γεμίζουμε την περιοχή της μνήμης που μας ενδιαφέρει με κάποιον χαρακτήρα –συνήθως τον `'\0'`– με τη συνάρτηση της C *memset()*:[15]

```
   memset( a2, '\0', sizeof(double) );  delete a2;
```

και στην *f*1:

```
void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
   memset( m, '\0', strlen(m) );
}
```

Σημείωση: ▶

Η *memset()* έχει επικεφαλίδα:

```
void* memset( void* s, int c, size_t n );
```

και λειτουργεί ως εξής: βάζει *n* αντίγραφα του χαρακτήρα *c* στην περιοχή της μνήμης που αρχίζει από τη διεύθυνση *s*. Επιστρέφει την τιμή του *s*. ◀

Πάντως τα πράγματα μπορεί και να μη γίνουν όπως τα περιμένεις: Μερικοί μεταγλωττιστές, υπερβολικώς «ευφυείς», αν βρουν εντολές που αλλάζουν το περιεχόμενο μιας περιοχής της μνήμης που στη συνέχεια ανακυκλώνεται –χωρίς να μεσολαβει χρήση του περιεχόμενου– *δεν μεταγλωττίζουν τις εντολές αλλαγής για να κάνουν το πρόγραμμα ταχύτερο!* Το ΛΣ μπορεί να έχει εργαλεία για τη λύση του προβλήματος.[16]

Η δεύτερη σύσταση έχει να κάνει με τη μνήμη συνολικώς και όχι μόνον τη δυναμική:

- Στην §16.3 μιλήσαμε για τη διαδικασία ανταλλαγών μνήμης και είπαμε ότι όλη η μνήμη που απαιτείται για το κάθε πρόγραμμα βρίσκεται κατ' αρχήν στον δίσκο. Αν λοιπόν κάποιος «κακός» ξέρει καλά το ΛΣ αλλα δεν μπορεί να «σπάσει» τις προστασίες του Συστήματος Διαχείρισης Βάσεων Δεδομένων (DBMS) μπορεί να «κλέβει» στιγμιότυπα μνήμης προγραμμάτων που χρησιμοποιούν τα δεδομένα που τον ενδιαφέρουν.
- Ορισμένα ΛΣ, όταν έχουν *μη κανονικό* τερματισμό εκτέλεσης προγράμματος δίνουν μια **απόρριψη** (περιεχόμενου της) **μνήμης** του (memory dump) σε κάποιο αρχείο, για να διευκολύνουν τον εντοπισμό του προβλήματος. Ο «κακός», που λέγαμε παραπάνω, θα μπορούσε να προκαλεί μη κανονικούς τερματισμούς προγραμμάτων και να κλέβει τα αρχεία με τα περιεχόμενα της μνήμης.

  Αυτά τα προβλήματα μπορεί να αντιμετωπισθούν μόνον μέσω του ΛΣ.

16.15 Ανακεφαλαίωση

Στο πρόγραμμά μας μπορούμε να παίρνουμε μνήμη για να υλοποιήσουμε μεταβλητές ή πίνακες σύμφωνα με τις ανάγκες που προκύπτουν κατά τη διάρκεια της εκτέλεσης. Αν η *p* είναι μεταβλητή-βέλος τύπου *T*`*` τότε:

- Με την εντολή "`p = new T`" παίρνουμε μνήμη για την υλοποίηση μιας *δυναμικής μεταβλητής* τύπου *T*. Η *p* δείχνει αυτήν τη μεταβλητή που τη χειριζόμαστε ως "`*p`".
- Η `*`*p* είναι όπως οποιαδήποτε μεταβλητή τύπου *T* και ως τέτοια τη χειριζόμαστε.
- Όταν δεν χρειαζόμαστε την `*`*p* την ανακυκλώνουμε με την "`delete p`". Αμέσως μετά τη "`delete p`" η `*`*p* δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.
- Με την "`p = new T[`*Π*`]`" ζητούμε μνήμη για έναν δυναμικό πίνακα με στοιχεία τύπου *T*. Το πλήθος τους καθορίζεται από την τιμή *n* της παράστασης *Π* που θα πρέπει να είναι ακέραιου τύπου και θετική.

---

[15] Θα πρέπει να έχεις δώσει "`#include <string>`" για να τη χρησιμοποιήσεις.

[16] Για παράδειγμα, στο Windows API υπάρχει η συνάρτηση *SecureZeroMemory()*.

- Τα στοιχεία του πίνακα είναι `p[0]`, `p[1]`, … `p[`*n*`-1]`. Χειριζόμαστε τον πίνακα όπως έναν συνήθη πίνακα με *n* στοιχεία τύπου *T*.
- Όταν δεν χρειαζόμαστε τον δυναμικό πίνακα τον ανακυκλώνουμε με την "`delete p[]`".
- Όταν ένα βέλος δεν δείχνει συγκεκριμένο στόχο –π.χ. μετά από δήλωση χωρίς αρχική τιμή ή μετά από ανακύκλωση του στόχου– βάζουμε στο βέλος τιμή "`0`" (ή "`NULL`" ή (*std*`::`)"*nullptr*").
- Η απόπειρα ανακύκλωσης ήδη ανακυκλωμένης μνήμης ("`delete p; delete p`") είναι σοβαρό λάθος. Η "`delete 0`" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "`delete[] 0`".

Έχεις δυνατότητα να χειριστείς τη δυναμική μνήμη με τα εργαλεία της C. Αλλά:

- Μνήμη που παίρνεις με *malloc(), calloc()* θα πρέπει να ανακυκλώνεται με *free()* και όχι με `delete`.
- Μνήμη που παίρνεις με `new`, `new … […]` θα πρέπει να ανακυκλώνεται με `delete`, `delete[]` αντιστοίχως.

Ακόμη καλύτερα, αν αυτό είναι δυνατόν:

- Μην αναμειγνύεις τους δύο τρόπους στο ίδιο πρόγραμμα.

Εκτός από το «παράπτωμα» της επαναλαμβανόμενης ανακύκλωσης τα σοβαρότερα προβλήματα με τη χρήση δυναμικής μνήμης είναι τα εξής:

- Η *διαρροή μνήμης*, δηλαδή η απώλεια ελέγχου σε τμήμα δυναμικής μνήμης που παραμένει δεσμευμένη από το πρόγραμμά μας αλλά δεν μπορούμε να τη χειριστούμε διότι δεν υπάρχει βέλος που να τη δείχνει.
- Το *μετέωρο βέλος*, δηλαδή βέλος που δείχνει τμήμα δυναμικής μνήμης που έχει ήδη ανακυκλωθεί.

Ασκήσεις

Α Ομάδα

16-1 Στην §14.7.3 γράψαμε τη *swap()* για ορμαθούς της C χρησιμοποιώντας ως ενδιάμεσο ενταμιευτή μια μεταβλητή τύπου *string*. Τότε δεν ξέραμε από δυναμική μνήμη και πήραμε βοήθεια από τον τύπο *string* (που ξέρει). Τώρα, που έμαθες τη χρήση της δυναμικής μνήμης, να την ξαναγράψεις. Προσεκτικά...

16-2 Ας ξαναδούμε το πρόγραμμα με το οποίο δοκιμάζαμε τον αλγόριθμο του Horner για την τιμή πολυωνύμου (§9.4, Παράδ. 1). Στην πρώτη μορφή, είχαμε δηλώσει:

```
const int N = 20;
double a[N];
```

έναν αρκετά μεγάλο πίνακα a που θα μπορούσε να χωρέσει συντελεστές πολυωνύμου μέχρι και 19ου βαθμού. Έτσι όμως, από τη μια δεν μπορούμε να υπολογίσουμε τιμές πολυωνύμου 20ου βαθμού και από την άλλη αν έχουμε πολυώνυμα μικρού βαθμού πολλά στοιχεία του πίνακα είναι άχρηστα.

Τώρα, χρησιμοποιώντας δυναμικό πίνακα συντελεστών, μπορείς να έχεις ακριβώς αυτά που χρειάζεσαι.

16-3 Τροποποιώντας την *SList_deleteAll()*, γράψε μια *SList_clear()* που «καθαρίζει» τη λίστα –ανακυκλώνει όλους τους κόμβους αλλά όχι τον φρουρό– ώστε να μπορεί να χρησιμοποιηθεί ξανά.

κεφάλαιο

17

* Εσωτερική Παράσταση Δεδομένων

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις

- πώς παριστάνονται στη μνήμη του ΗΥ οι τιμές διαφόρων τύπων και
- περιορισμούς και προβλήματα που προκύπτουν από τους τρόπους παράστασης.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις πιο αξιόπιστα προγράμματα.

Έννοιες κλειδιά:

- *παράσταση ακεραίων*
- *παράσταση πραγματικών*
- *διαχείριση δυαδικών ψηφίων*
- *ψηφιοπράξεις*
- *σφάλματα παράστασης*
- *σφάλματα πράξεων*

Περιεχόμενα:

17.1 Παράσταση Φυσικών 541
17.2 Παράσταση Ακεραίων - Αρνητικοί Αριθμοί 543
17.2.1 * Ακέραιοι Τύποι του C99 544
17.3 Οι Ακέραιοι στο Πρόγραμμα 545
17.3.1 Παράμετροι "`unsigned`" 547
17.4 * Απαριθμητοί Τύποι (ξανά) 548
17.5 Ψηφιοπράξεις στη C++ 549
17.6 Ψηφιοχάρτες και Συνηθισμένες Πράξεις 552
17.6.1 Τιμή Δυαδικού Ψηφίου 553
17.6.2 Βάλε Τιμή 1 σε Δυαδικό Ψηφίο 554
17.6.3 Βάλε Τιμή 0 σε Δυαδικό Ψηφίο 555
17.6.4 Πλήθος "1" 556
17.6.5 Μέρος Ψηφιοχάρτη 556
17.7 Τύποι *bitmask* 557
17.8 Αριθμητικές Πράξεις και Ψηφιοπράξεις 560
17.9 Παράσταση και Πράξεις στον Τύπο "`float`" 560
17.9.1 Υπολογισμός Περιοδικής Συνάρτησης 564
17.10 Άλλοι Τύποι Κινητής Υποδιαστολής 564
17.11 Ο Τύπος "`float`" στο Δυαδικό Σύστημα 565
17.11.1 Πόλωση 566
17.11.2 Άλλες Περιπλοκές - Πρότυπο IEEE 567
17.11.3 Οι Τύποι "`double`" και "`long double`" 568
17.11.4 Μερικά Χρήσιμα Εργαλεία από τη C 569
17.12 Σφάλμα από Μετατροπή Τύπου 570

17.13 Τα Σφάλματα και πώς Μεταδίδονται 570
17.13.1 Το Σφάλμα Παράστασης 571
17.13.2 Μετάδοση Σφαλμάτων 571
17.14 Ισότητα στους Τύπους Κινητής Υποδιαστολής 573
17.15 Πρακτικές Συμβουλές 576
Ασκήσεις 578
Α Ομάδα 578
Β Ομάδα 579
Γ Ομάδα 579

Εισαγωγικές Παρατηρήσεις – Αριθμητικά Συστήματα:
Ένας από τους πιό γνωστούς τρόπους κωδικοποίησης αριθμητικών πληροφοριών είναι η κωδικοποίηση στο γνωστό **δεκαδικό σύστημα**, στο οποίο χρησιμοποιούμε δέκα διαφορετικά σύμβολα, τα ψηφία: 0,1,2,3,4,5,6,7,8,9.

Για να παραστήσουμε στο σύστημα αυτό έναν ακέραιο αριθμό (θετικό ή αρνητικό), χρησιμοποιούμε ένδεκα σύμβολα: τα δέκα ψηφία και το πρόσημο "**-**" (μείον). Πολλές φορές χρησιμοποιούμε και το πρόσημο "**+**" (συν), για να ξεχωρίζουμε ευκολότερα τους θετικούς αριθμούς από τους αρνητικούς (π.χ. **-**5109, 2048, +2048). Ακόμη, στην παράσταση ενός κλασματικού αριθμού χρειαζόμαστε και την υποδιαστολή (ευρωπαϊκές χώρες: "**,**", ΗΠΑ. Μεγ. Βρεττανία: "**.**"), που διαχωρίζει το ακέραιο μέρος του αριθμού από το κλασματικό (π.χ. 48,25).

Το κοινό δεκαδικό αριθμητικό σύστημα είναι **θεσιακό** (positional), διότι η τιμή που παριστάνει κάθε ψηφίο ενός αριθμού εξαρτάται από τη θέση που έχει στην παράσταση του αριθμού. Για παράδειγμα, το πρώτο ψηφίο 3 στον αριθμό 6343 σημαίνει την τιμή 300 (3×10^2), ενώ το δεύτερο ψηφίο 3 την τιμή 3 (3×10^0).

Εκτός από τα θεσιακά συστήματα αριθμήσεως υπάρχουν και μη θεσιακά συστήματα, όπως είναι το γνωστό –**προσθετικό** (additive)– ρωμαϊκό σύστημα (π.χ. XXXIII = 10 + 10 + 10 + 1 + 1 + 1 = 33) ή το Ελληνικό (ρβ′ = 100 + 2 = 102), όπου τα σύμβολα που χρησιμοποιούμε έχουν την ίδια τιμή ανεξαρτήτως της θέσης τους στον αριθμό που παριστάνουν.

Σε ένα θεσιακό αριθμητικό σύστημα κάθε φυσικός αριθμός N μπορεί να παρασταθεί με ένα πολυώνυμο της εξής μορφής:

$$N = \psi_{L-1}B^{L-1} + \psi_{L-2}B^{L-2} + \ldots + \psi_0 B^0 \qquad (1)$$

Ο αριθμός B λέγεται **βάση** (base, radix) του αριθμητικού συστήματος και υποθέτουμε γενικώς ότι είναι μεγαλύτερος από το 1 (υπάρχουν όμως και συστήματα με αρνητική βάση). Οι συντελεστές ψ_k (k: $0..L-1$) που μπορεί να πάρουν τιμές στο $[0..B)$ ($0 \le \psi_k < B$), αποτελούν τα διαδοχικά ψηφία του αριθμού N ο οποίος έχει L θέσεις (ή ψηφία). Έτσι, η βάση B καθορίζει το πλήθος των διαφορετικών ψηφίων ενός αριθμητικού συστήματος. Στο δεκαδικό αριθμητικό σύστημα έχουμε βάση $B = 10$ και τα δέκα διαφορετικά ψηφία είναι: 0, 1, 2,...,9.

Δηλαδή επιλέγοντας τη βάση, δημιουργούμε ένα αριθμητικό σύστημα. Όταν π.χ. η βάση B = 2 ή 3, 8, 10, 16, τότε το σύστημα λέγεται **δυαδικό** ή αντιστοίχως **τριαδικό, οκταδικό, δεκαδικό, δεκαεξαδικό**. Έτσι λοιπόν ο αριθμός N = 7635, του δεκαδικού θεσιακού συστήματος, μπορεί να γραφτεί με την εξής μορφή:

$$\mathrm{N} = 7\times10^3 + 6\times10^2 + 3\times10^1 + 5\times10^0 = 7635$$

όπου $\psi_3 = 7$, $\psi_2 = 6$, $\psi_1 = 3$ και $\psi_0 = 5$.

Στο οκταδικό σύστημα η βάση B = 8 και τα διάφορα ψηφία αυτού του συστήματος είναι: 0,1,2,...,7. Επομένως ο αριθμός N = 7635, του οκταδικού συστήματος, παριστάνει την τιμή του δεκαδικού αριθμού 5149, επειδή:

$$N = 7635_8 = 7\times8^3 + 6\times8^2 + 3\times8^1 + 5\times8^0 = 5149_{10}$$

Βλέπουμε λοιπόν ότι η ίδια ακολουθία ψηφίων (π.χ. 7635) αντιπροσωπεύει άλλη τιμή στο οκταδικό σύστημα και άλλη στο δεκαδικό. Έτσι, για να μη γίνεται σύγχυση, συχνά γράφουμε σαν κάτω δείκτη, στο δεξιό μέρος της ακολουθίας ψηφίων ενός αριθμού τη βάση του αριθμητικού συστήματος που χρησιμοποιούμε (πάντα στο δεκαδικό συμβολισμό). Στην

παρουσίαση των αριθμών στα θεσιακά συστήματα σημειώνουμε, για λόγους απλοποίησης, μόνο τους όρους ψ_k (σε κατιούσα σειρά) και παραλείπουμε τις δυνάμεις B.

17.1 Παράσταση Φυσικών

Στους ψηφιακούς ΗΥ χρησιμοποιείται το ***δυαδικό*** (binary) *αριθμητικό σύστημα*, για λόγους τεχνολογικής αξιοπιστίας (αλλά και θεωρητικούς). Η Φυσική και η Ηλεκτρονική έχουν να προτείνουν **συστήματα δύο καταστάσεων** (two state systems), στα οποία μπορούμε

- να ανιχνεύουμε την κατάσταση που βρίσκεται το σύστημα και
- να το βάζουμε στην κατάσταση που θέλουμε.

Σε ένα τέτοιο σύστημα, «βαφτίζουμε» τη μια κατάσταση "`0`" και την άλλην "`1`"· παριστάνουμε δηλαδή τα δύο ψηφία του δυαδικού συστήματος ($B = 2$).

Σύμφωνα με αυτά που είπαμε παραπάνω, στο δυαδικό σύστημα κάθε φυσικός αριθμός παριστάνεται ως άθροισμα δυνάμεων του 2. Για παράδειγμα, ο δεκαδικός αριθμός $N = 12$ ($8 + 4 = 2^3 + 2^2$), μπορεί να παρασταθεί ως εξής:

$$N = 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = 1100_2$$

όπου $\psi_3 = 1$, $\psi_2 = 1$, $\psi_1 = 0$ και $\psi_0 = 0$.

Συχνά στα υπολογιστικά συστήματα χρησιμοποιείται και το **δεκαεξαδικό** (hexadecimal) αιθμητικό σύστημα, για το οποίο $B = 16$ και τα ψηφία του είναι:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`

Τα νέα σύμβολα *A*, *B*, *C*, *D*, *E* και *F* (ή *a*, *b*, *c*, *d*, *e*, *f*) αντιστοιχούν στις τιμές των (δεκαδικών) αριθμών: δέκα, ένδεκα, δώδεκα, δεκατρία, δεκατέσσερα και δεκαπέντε. Κατά συνέπεια ο δεκαεξαδικός αριθμός 153 σημαίνει:

$$153_{16} = 1\times16^2 + 5\times16^1 + 3\times16^0 = 256 + 80 + 3 = 339_{10}$$

και ο δεκαεξαδικός αριθμός $1A3F$ σημαίνει:

$$\begin{aligned} 1A3F_{16} &= 1\times16^3 + A\times16^2 + 3\times16^1 + F\times16^0 \\ &= 1\times16^3 + 10\times16^2 + 3\times16^1 + 15\times16^0 \\ &= 6719_{10} \end{aligned}$$

Ο Πίν. 17-1 δείχνει την παράσταση των ακεραίων αριθμών από 1 μέχρι 20 σε διάφορα θεσιακά αριθμητικά συστήματα.

Από τον Πίν. 17-1 φαίνεται ότι όταν μικραίνει η βάση του αριθμητικού συστήματος μεγαλώνει το πλήθος των ψηφίων που χρειάζονται για να παρασταθεί ένας αριθμός. Για παράδειγμα, ο αριθμός 9 χρειάζεται ένα ψηφίο στο 16-δικό και στο 10-δικό σύστημα, δυό στο 8-δικό, τρία στο 3-δικό και τέσσερα στο 2-δικό σύστημα.

Το πλήθος Π των διαφόρων αριθμών που μπορεί να γραφούν σε L θέσεις ενός αριθμητικού συστήματος βάσεως B δίνεται από τον τύπο:

$$\Pi = B^L \tag{2}$$

Αν, ας πούμε, πάρουμε $B = 2$ και $L = 4$, τότε μπορούμε να ξεχωρίσουμε 16 ($=2^4$) διαφορετικούς δυαδικούς αριθμούς (ή συνδυασμούς), δηλ. τους αριθμούς: 0000, 0001, 0010,... 1111.

Αν τώρα γνωρίζουμε το πλήθος των ακεραίων αριθμών Π, που θέλουμε να παραστήσουμε στο αριθμητικό σύστημα με βάση το B, τότε ο απαιτούμενος αριθμός θέσεων L καθορίζεται από τον τύπο:

$$L = \log_B\Pi \tag{3}$$

Έτσι, για να παραστήσουμε τους πρώτους 16 φυσικούς αριθμούς, από 0 μέχρι 15, στο δυαδικό σύστημα χρειαζόμαστε $\log_2 16 = 4$ θέσεις. Ενώ για την παράσταση των πρώτων 9 αριθμών στο τριαδικό σύστημα χρειαζόμαστε $\log_3 9 = 2$ θέσεις. Μπορείς να ελέγξεις την ορθότητα του τύπου (3) και από τον Πίν. 17-1.

Ακόμη, από τον Πίν. 17-1, μπορείς να δεις ότι το οκταδικό και το δεκαεξαδικό είναι «συμπυκνώσεις» του δυαδικού συστήματος. Π.χ. το "20_{10}" στο δυαδικό γράφεται "10100". Αν

10δικό	16δικό	8δικό	3δικό	2δικό
0	00	00	000	00000
1	01	01	001	00001
2	02	02	002	00010
3	03	03	010	00011
4	04	04	011	00100
5	05	05	012	00101
6	06	06	020	00110
7	07	07	021	00111
8	08	10	022	01000
9	09	11	100	01001
10	0A	12	101	01010
11	0B	13	102	01011
12	0C	14	110	01100
13	0D	15	111	01101
14	0E	16	112	01110
15	0F	17	120	01111
16	10	20	121	10000
17	11	21	122	10001
18	12	22	200	10010
19	13	23	201	10011
20	14	24	202	10100

Πίν. 17-1 Παράσταση αριθμών σε διάφορα αριθμητικά συστήματα.

δεις αυτήν την παράσταση ως δυο τριάδες: "010 | 100" και κάθε μια από αυτές ως ψηφίο του οκταδικού συστήματος, παίρνεις τον "24_8". Παρομοίως, αν τη δεις ως δυο τετράδες "0001 | 0100" και τις γράψεις ως ψηφία του δεκαεξαδικού παίρνεις: "14_{16}". Φυσικά, αυτές οι ιδιότητες ξεκινούν από το ότι $8 = 2^3$ και $16 = 2^4$.

Στην §2.5 (Πίν. 2-1) είδαμε ότι η C++, για την παράσταση φυσικών αριθμών, μας δίνει τους ακέραιους τύπους χωρίς πρόσημο: `unsigned char`, `unsigned int`, `unsigned long`. Ο `unsigned char` αποθηκεύει τιμές σε μια ψηφιολέξη[1] των οκτώ δυαδικών ψηφίων· σύμφωνα με τον τύπο (2), μπορεί να παραστήσει $2^8 = 256$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 255. Ο `unsigned short int` δουλεύει με μια λέξη (δυο ψηφιολέξεις) με 16 δυαδικά ψηφία και μπορεί να παραστήσει $2^{16} = 65\,536$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 65 535. Τέλος, ο `unsigned long` μπορεί να παραστήσει $2^{32} = 4\,294\,967\,296$ τιμές, από 0 μέχρι 4 294 967 295. Και ο `unsigned int`; Μπορεί να σαν τον `unsigned long` ή σαν τον `unsigned short int`.

Στη συνέχεια θα δούμε έναν τρόπο για να «σκαλίζουμε» τις εσωτερικές παραστάσεις.

Στις §1.8.1 και §1.8.2 είδαμε ότι η C++ μας δίνει τη δυνατότητα να γράφουμε στο πρόγραμμά μας φυσικούς αριθμούς στο δεκαεξαδικό σύστημα βάζοντας το πρόθεμα "`0x`" ή "`0X`" και στο οκταδικό σύστημα βάζοντας το πρόθεμα "`0`". Π.χ., οι εντολές:

```
k = 255;   k = 0XFF;   k = 0xff;   k = 0377;
```

κάνουν ακριβώς το ίδιο πράγμα. Το "255" είναι μια αριθμητική σταθερά στο δεκαδικό σύστημα. Η ίδια τιμή στο δεκαεξαδικό σύστημα γράφεται "FF_{16}" και στο οκταδικό "377_8". Στη

[1] Οι τιμές των μεγεθών που δίνουμε εδώ δεν είναι υποχρεωτικές, είναι απλώς συνηθισμένες. Θυμίσου ότι όπως λέγαμε στο Κεφ. 2 το υποχρεωτικό είναι ότι:

$$\texttt{sizeof(char)} \le \texttt{sizeof(short)} \le \texttt{sizeof(int)} \le \texttt{sizeof(long)}$$

C++, η δεκαεξαδική τιμή γράφεται "`0XFF`" και η οκταδική "`0377`". Πρόσεξε καλά την οκταδική γραφή:

- ***Όταν μια αριθμητική σταθερά ακέραιου τύπου αρχίζει με "`0`" (μηδέν) είναι οκταδικός και και όχι δεκαδικός αριθμός.***

17.2 Παράσταση Ακεραίων – Αρνητικοί Αριθμοί

Πώς μπορούμε να παραστήσουμε σε μια ψηφιολέξη ακέραιους θετικούς ή αρνητικούς; Αφού το πρόσημο μπορεί να είναι "`+`" ή "`-`", μπορούμε να το παραστήσουμε με ένα δυαδικό ψηφίο: "`0`" για το "`+`" και "`1`" για το "`-`". Έτσι, θα μας μείνουν άλλα 7 δυαδικά ψηφία για την απόλυτη τιμή, που θα μπορεί να είναι από 0 μέχρι $2^7 - 1 = 127$. Να λοιπόν ένας τρόπος για να παραστήσουμε ακέραιες τιμές από –127 μέχρι +127. Αυτός ο τρόπος παράστασης λέγεται «**πρόσημο - απόλυτη τιμή**». Αλλά πρόσεξε: εδώ έχουμε 255 τιμές, ενώ στον τύπο `unsigned char` παριστάνουμε 256 διαφορετικές τιμές από –128 μέχρι 127! Ναί, χάσαμε μια τιμή, διότι το 0 (μηδέν) παριστάνεται με δυο διαφορετικούς τρόπους: ως "`+0`" (`00000000`) και ως "`-0`" (`10000000`).

Συνήθως, στους υπολογιστές μας θα βρούμε την παράσταση αρνητικών με το "**συμπλήρωμα ως προς 2**" (2's complement). Ας δούμε ένα

Παράδειγμα

Όπως είδαμε, το 12 παριστάνεται σε μια ψηφιολέξη, ως:

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

Για να βρούμε το συμπλήρωμα ως προς 2:

Βήμα 1: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1

(αυτό είναι το **συμπλήρωμα ως προς 1**)

Βήμα 2: προσθέτουμε το 1

```
  1 1 1 1 0 0 1 1
+ 0 0 0 0 0 0 0 1
-----------------
  1 1 1 1 0 0 1 0
  7 6 5 4 3 2 1 0
```

Αυτήν την παράσταση χρησιμοποιούμε για να παραστήσουμε το "`-12`".

☜☜☜

Το ότι η τιμή είναι αρνητική φαίνεται –όπως και στην «πρόσημο-απόλυτη τιμή»– από το "`1`" στο δυαδικό ψηφίο 7. Τί κερδίσαμε από όλα αυτά; Ας του προσθέσουμε την παράσταση του 20 (`00010100`) αγνοώντας το ότι το πρώτο ψηφίο είναι πρόσημο:

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

Αν ξεχάσουμε το πιο σημαντικό ψηφίο, που είναι "`1`", τα υπόλοιπα οκτώ ψηφία παριστάνουν το 8, που είναι σωστό: (-12) + 20 = 8.

Ένας άλλος τρόπος για να πούμε το ίδιο πράγμα είναι ο εξής: Το αποτέλεσμα της πράξης ήταν $2^8+2^3 = 264$. Κρατάμε το υπόλοιπο της διαίρεσής του δια 256 (= 2^8). Γι' αυτό, αυτή η αριθμητική λέγεται **αριθμητική υπολοίπων** (modulo arithmetic).

Δηλαδή:

- ***Όταν παριστάνουμε τους αρνητικούς με συμπλήρωμα ως προς 2 κάνουμε πρόσθεση χωρίς να ενδιαφερόμαστε για τα πρόσημα των προσθεταίων.***

Να λοιπόν πώς δουλεύει ο υπολογιστής στην περίπτωση αυτή:

- οι αρνητικοί παριστάνονται με συμπλήρωμα ως προς 2,
- κατά την πρόσθεση το ψηφίο προσήμου δεν έχει διαφορετική μεταχείριση από τα άλλα,

- η πρόσθεση γίνεται με αριθμητική υπολοίπων ως προς 2^N, όπου N το πλήθος δυαδικών ψηφίων που χρησιμοποιούνται για την παράσταση.

  Ποιος είναι ο μέγιστος αριθμός που μπορούμε να παραστήσουμε; Είναι ο

  7 6 5 4 3 2 1 0
  0 1 1 1 1 1 1 1 = 127_{10}

  Η ελάχιστη τιμή είναι -128 και παριστάνεται ως:

  7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 0 = -128_{10}

Μάλλον θα θέλεις κάποια βοήθεια για να βρεις το "**-128**". Λοιπόν: αφού έχει "**1**" στο ψηφίο 7, είναι αρνητικός. Ας εφαρμόσουμε *αντιστρόφως* αυτά που κάναμε για να υπολογίσουμε το συμπλήρωμα ως προς 2:

Βήμα 1: αφαιρούμε το 1

```
  7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 0
-               1
-----------------
  0 1 1 1 1 1 1 1
  7 6 5 4 3 2 1 0
```

Βήμα 2: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

10000000

που είναι το $2^7 = 128$.

Το μηδέν παριστάνεται με έναν μοναδικό τρόπο: "**00000000**".

Η C++ έχει τρεις τύπους που τους χειρίζεται με τον τρόπο αυτόν:

- (**signed**) **char**, σε 8 ψηφία. Παριστάνει τιμές από **-**128 μέχρι 127 με αριθμητική υπολοίπου ως προς $2^8 = 256$.
- **short int**, σε 16 ψηφία. Παριστάνει τιμές από **-**32 768 μέχρι 32 767 με αριθμητική υπολοίπου ως προς $2^{16} = 65\,536$.
- **int** και **long int**, σε 32 ψηφία. Παριστάνει τιμές από **-**2 147 483 648 μέχρι 2 147 483 647 με αριθμητική υπολοίπου ως προς $2^{32} = 4\,294\,967\,296$.

Να (ξανα)τονίσουμε ότι *τα παραπάνω μεγέθη δεν καθορίζονται από το πρότυπο* της γλώσσας. Ας πούμε ότι είναι συνηθισμένες τιμές.

Στη συνέχεια θα δούμε και μερικούς ακόμη ακέραιους τύπους.

Πού θα μας χρειαστούν όλα αυτά; Το συζητούμε στην συνέχεια.

17.2.1 * Ακέραιοι Τύποι του C99

Στο πρότυπο C99 (ISO/IEC 1999) της C εισάγονται ακέραιοι τύποι με 64 δυαδικά ψηφία: **long long int** και **unsigned long long int**.

- Ελάχιστη τιμή του **long long int**

  LLONG_MIN: **-9223372036854775807** = $-(2^{63} - 1)$

- Μέγιστη τιμή του **long long int**

  LLONG_MAX: **+9223372036854775807** = $2^{63} - 1$

- Μέγιστη τιμή του **unsigned long long int**

  ULLONG_MAX: **18446744073709551615** = $2^{64} - 1$

Οι **long long int** και **unsigned long long int** προβλέπονται και στο C++11.

Πέρα από αυτούς τους ακέραιους 64 δυαδικών ψηφίων, το C99 υποδεικνύει και μερικούς ορισμούς-μεταονομασίες τύπων. Η πρώτη οικογένεια περιλαμβάνει ορισμούς ονομάτων της μορφής **int***N***_t** και **uint***N***_t**. Το *N* υποδηλώνει το πλήθος δυαδικών ψηφίων. Ο gcc (Dev C++), στο **stdint.h**, έχει τους εξής σχετικούς ορισμούς:

```
typedef signed char         int8_t;
typedef unsigned char       uint8_t;
typedef short               int16_t;
typedef unsigned short      uint16_t;
typedef int                 int32_t;
```

```
typedef unsigned           uint32_t;
typedef long long          int64_t;
typedef unsigned long long uint64_t;
```

Τι κερδίζουμε με αυτούς; Όταν θέλουμε μια ακέραιη μεταβλητή με 32 ψηφία δεν χρειάζεται να ψάχνουμε αν θα τη δηλώσουμε υποχρεωτικώς `long int` ή μας κάνει και ο `int`· δηλώνουμε:

```
   int32_t x;
```

και τελειώνουμε.

Μια άλλη οικογένεια περιλαμβάνει ονόματα της μορφής `int_least`*N*`_t` και `uint_least`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Για παράδειγμα, ο τύπος `int_least32_t` είναι ο «μικρότερος« ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Μια τρίτη οικογένεια περιλαμβάνει ονόματα της μορφής `int_fast`*N*`_t` και `uint_fast`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Ο τύπος `int_fast32_t` είναι ο «ταχύτερος» (ό,τι και αν σημαίνει αυτό) ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Τέλος, ο τύπος `intmax_t` είναι αυτός στον οποίον μπορούμε να παραστήσουμε οποιαδήποτε ακέραιη τιμή μπορεί να παρασταθεί στη συγκεκριμένη υλοποίηση. Ο αντίστοιχος για τιμές χωρίς πρόσημο είναι ο `uintmax_t`. Ο gcc, στο `stdint.h`, έχει:

```
typedef long long          intmax_t;
typedef unsigned long long uintmax_t;
```

Όλοι αυτοί οι ορισμοί προβλέπεται να υπάρχουν στο νέο πρότυπο της C++.

17.3 Οι Ακέραιοι στο Πρόγραμμα

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
  int  i( INT_MAX );
  i = 4*i;
  cout << i << endl;
}
```

Αποτέλεσμα:

```
-4
```

Ας δούμε τί έγινε: Η (gcc - Dev) C++ παριστάνει τιμές τύπου `int` σε 32 δυαδικά ψηφία και το *INT_MAX* παριστάνεται ως εξής:

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

Το αποτέλεσμα του πολλαπλασιασμού υπολογίζεται σε:

```
3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0
2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 = 8589934588
```

Αλλά, στον τύπο `int` δεν μπορούμε να παραστήσουμε αυτήν την τιμή· έχουμε **υπερχείλιση** (overflow). Δουλεύοντας με αριθμητική υπολοίπων και συμπλήρωμα ως προς 2, η συνέχεια είναι η εξής: στη μεταβλητή *i* αποθηκεύονται τελικώς τα ψηφία 0 μέχρι 31 του αποτελέσματος. Όταν έρθει η ώρα να εκτελεσθεί η εντολή `cout << i << endl`, πρώτα ελέγχεται το ψηφίο προσήμου (31)· αφού έχει τιμή 1, ο υπολογιστής καταλαβαίνει ότι πρόκειται για *αρνητική τιμή!* Στη συνέχεια, αφαιρεί 1:

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

και αλλάζει το 1 σε 0 και τα 0 σε 1:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

που είναι το 4.

Τι δίδαγμα βγαίνει από αυτά;

- ***Στις πράξεις του `int` (και των άλλων ακεραίων τύπων) είναι δυνατόν να έχουμε υπερχείλιση χωρίς καμιά ειδοποίηση από τον υπολογιστή. Αυτό είναι συχνά από τα δυσκολότερα λάθη, τουλάχιστον στην ανίχνευση.***

Πώς αντιμετωπίζεται αυτό το πρόβλημα; Ας δούμε τί μπορούμε να κάνουμε με την πρόσθεση. Έχουμε δυο μεταβλητές x, y, τύπου `int` και θέλουμε να υπολογίσουμε το $x + y$, αν υπολογίζεται. Για τα x, y έχουμε:

$$INT\_MIN \le x \le INT\_MAX$$
$$INT\_MIN \le y \le INT\_MAX$$

και θα αποπειραθούμε την πρόσθεση μόνον αν ξέρουμε από πριν ότι

$$INT\_MIN \le x + y \le INT\_MAX \text{ ή}$$
$$INT\_MIN - y \le x \le INT\_MAX - y$$

Φυσικά, δεν μπορούμε να γράψουμε:

```
if (INT_MIN-y <= x && (x <= INT_MAX-y)
  s = x + y;
else
  Λάθος
```

διότι οι πράξεις `INT_MIN - y` και `INT_MAX - y` *δεν είναι ασφαλείς!* Π.χ. αν η y έχει αρνητική τιμή, η `INT_MAX - y` μας δίνει σίγουρα υπερχείλιση. Ας τα ξαναδούμε πιο προσεκτικά. Κατ' αρχάς, αν οι x, y έχουν ετερόσημες τιμές δεν υπάρχει περίπτωση υπερχείλισης με την πρόσθεση. Αν οι x, y έχουν τιμές ≥ 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $x \le INT\_MAX - y$ και η πράξη δεξιά είναι ασφαλής. Αν οι x, y έχουν τιμές < 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $INT\_MIN - y \le x$ και η πράξη αριστερά είναι ασφαλής. Μπορούμε λοιπόν να γράψουμε την:

```
int addInt( int x, int y )
{
   int fv;

   if ( x >= 0 )
   {
      if ( y < 0 )  fv = x + y;
      else if ( x <= INT_MAX - y )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   else    // x < 0
   {
      if ( y >= 0 )  fv = x + y;
      else if ( INT_MIN - y <= x )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   return fv;
} // addInt
```

«Δηλαδή», σκέφτεσαι με φρίκη, «θα πρέπει αντί για "`c = a + b`" να γράφω "`c = addInt(a, b)`" και αντί για μια πράξη να καλώ μια συνάρτηση και να διαχειρίζομαι εξαιρέσεις;» Όχι! Σε ένα καλοσχεδιασμένο πρόγραμμα οι περισσότερες πράξεις *είναι* συνήθως ασφαλείς· και αυτές που δεν είναι φαίνονται εύκολα.[2] Φυσικά, δεν υπάρχει συνταγή για το πότε βάζουμε έλεγχο και πότε όχι· αλλά, δες δυο παραδείγματα:

```
#include <iostream>
#include <string>
#include <climits>
```

[2] Και ακόμη: για τις άλλες πράξεις τα πράγματα είναι πολύ πιο απλά.

```
using namespace std;

struct IntOvrflXptn
{
   char funcName[100];
   int  errCode;
   int  errVal1;
   int  errVal2;
   IntOvrflXptn( const char* fn, int ec, int ev1=0, int ev2= 0 )
   {
      strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal1 = ev1;  errVal2 = ev2;
   }
}; // IntOvrflXptn

int addInt( int x, int y );

int main()
{
   int x, y, z;

   cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
   cin >> x >> y;
   try
   {
      z = addInt( x, y );
      cout << z << endl;
   // . . .
   }
   catch ( IntOvrflXptn& xp )
   {
      cout << xp.errVal1 << " + " << xp.errVal2
           <<"??? ΣΟΒΑΡΕΨΟΥ!" << endl;
   }
// . . .
   do
   {
      cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
      cin >> x >> y;
   } while (x <= 0 || 100 < x || y <= 0 || 100 < y);
   z = x + y;
// . . .
} // main
```

Το μήνυμα που δίνεις πριν από την εντολή ανάγνωσης δεν σου εξασφαλίζει οτιδήποτε. Στην πρώτη περίπτωση δεν σε ενδιαφέρει να δεις αν ο χρήστης υπάκουσε στην οδηγία σου και προχωράς. Η χρήση της *addInt()* θα σε προφυλάξει από μια τιμή της *z* χωρίς νόημα. Στη δεύτερη περίπτωση, αφού δεν προχωράς παρά μόνο με «σωστές» τιμές των *x*, *y*, μπορείς να κάνεις την πρόσθεση χωρίς άλλον έλεγχο.

17.3.1 Παράμετροι "unsigned"

Στην §6.8 (και στην §13.3.1) δίναμε τον κανόνα:

- ***Μη βάζεις στις συναρτήσεις σου παραμέτρους τιμής τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

  Στο παράδειγμα παραβίασης του κανόνα που δίναμε καλούσαμε

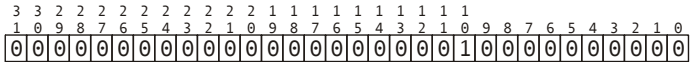
```
   n = -1024;
   cout << n << "  " << intSqrt(n) << endl;
```

τη συνάρτηση με επικεφαλίδα

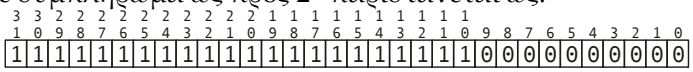
```
unsigned int intSqrt( unsigned int x );
```

Και τι βλέπαμε; Στη συνάρτηση περνούσε στη *x* η τιμή 4294966272 και η συνάρτηση υπολόγιζε την (ακέραιη) τετραγωνική της ρίζα.

Τώρα μπορείς να καταλάβεις τι γίνεται. Σε **int** τεσσάρων ψηφιολέξεων το 1024 (= 2^{10}) παριστάνεται ως:



Το "**-1024**" –σε συμπλήρωμα ως προς 2– παριστάνεται ως:



Αυτή είναι η εσωτερική παράσταση της *n* και αντιγράφεται ως τιμή της *x*. Μέσα στη συνάρτηση αυτή η παράσταση ερμηνεύεται ως **unsigned int**. Έτσι προκύπτει η τιμή 4294966272.

Άσκηση για σένα: η επιβεβαίωση της παράστασης του "**-1024**" και ο υπολογισμός της τιμής 4294966272.

17.4 * Απαριθμητοί Τύποι (ξανά)

Πρωτοείδαμε τους απαριθμητούς τύπους πολύ νωρίς (§4.8) και τους χρησιμοποιούμε όπως θα χρησιμοποιούσαμε τους αντίστοιχους της Pascal –και πολύ καλά κάναμε. Στη συνέχεια, σε ορισμένες περιπτώσεις, θα πρέπει να τους χρησιμοποιήσουμε όπως τους χρησιμοποιεί η C.

Ας ξαναδούμε τον τύπο

```
enum Digit { miden = 48, zero = 48, one, two, three, four,
             five, six, seven, eight, nine };
```

και τη δήλωση:

```
   Digit d1( 49 );
```

Ο μεταγλωττιστής θα την απορρίψει: «invalid conversion from '**int**' to '*Digit*'» (gcc –Dev C++).[3] Αν όμως δώσεις:

```
   Digit d1( static_cast<Digit>(49) );
```

όλα πάνε μια χαρά. Σωστό!

Δοκιμάζουμε όμως και το εξής:

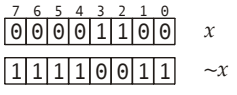
```
   d1 = static_cast<Digit>( 9 );
```

Και αυτό περνάει χωρίς το παραμικρό πρόβλημα, ενώ, με βάση αυτά που ξέρουμε, θα έπρεπε να γίνονται δεκτές τιμές από **static_cast<Digit>(49)** μέχρι και **static_cast <Digit>(57)**.

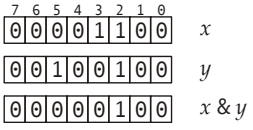
Αυτό που συμβαίνει στην πραγματικότητα είναι το εξής:

- Αν οι σταθερές που έχουμε στην απαρίθμηση είναι μη αρνητικές η C++ θα δεχτεί ως τιμή μιας σταθεράς κάθε ακέραιη τιμή από 0 μέχρι τη μέγιστη τιμή που μπορεί να παρασταθεί στα δυαδικά ψηφία –χωρίς να υπολογίζουμε ψηφίο προσήμου– που απαιτούνται για τη μέγιστη τιμή της απαρίθμησης. Στο παράδειγμά μας μέγιστη τιμή της απαρίθμησης είναι η *nine* που αντιστοιχεί στο $57_{10} = 111001_2$. Η μέγιστη τιμή που μπορεί να παρασταθεί σε έξη δυαδικά ψηφία είναι $111111_2 = 63_{10}$.
- Αν υπάρχουν και αρνητικές τιμές τότε ισχύουν τα ίδια αλλά θα πρέπει να υπολογίζουμε και ψηφίο προσήμου. Για παράδειγμα, αν στην απαρίθμηση του παραδείγματος βάλουμε άλλο ένα στοιχείο **neg = -1**, τότε η περιοχή είναι από -64 μέχρι 63 (παράσταση σε 7 ψηφία). Αν βάλουμε **neg = -100**, τότε η περιοχή είναι από -128 μέχρι 127 (παράσταση σε 8 ψηφία).

[3] Ο δικός σου μεταγλωττιστής, με ή χωρίς διμαρτυρίες (warnings), τη δέχτηκε! Συμβαίνουν και αυτά...



Σχ. 17-1 Η ψηφιοπράξη "**~**" (συμπλήρωμα ως προς 1).



Σχ. 17-2 Ψηφιοπράξη "**&**" (**and**).

Πάντως είναι πολύ πιθανό, ο μεταγλωττιστής σου να δεχτεί χωρίς διαμαρτύρίές ως τιμή μεταβλητής τύπου *Digit* τη `static_cast<Digit>(n)` όπου *n* τυχούσα τιμή τύπου `int`.

17.5 Ψηφιοπράξεις στη C++

Η C++ προσφέρει ορισμένες πράξεις που επιτρέπουν διαχείριση των τιμών όλων των ακέραιων τύπων της (δυαδικό) ψηφίο προς ψηφίο· για τον λόγο αυτόν τις ονομάζουμε **ψηφιοπράξεις** (bitwise operations).

Ας τις δούμε ξεκινώντας από τις πράξεις **ολίσθησης** (shift). Ας πούμε ότι έχουμε:

```
   unsigned char  x, y;
// . . .
   x = 77;
```

Στη μνήμη θα αποθηκευτούν σε μια ψηφιολέξη τα εξής:

7 6 5 4 3 2 1 0
0 1 0 0 1 1 0 1

Αν δώσουμε την εντολή `y = x << 2` (ολίσθησε αριστερά κατά 2 δυαδικά ψηφία), θα συμβούν τα εξής:

- οι τιμές όλων των ψηφίων της *x* θα ολισθήσουν κατά 2 θέσεις προς τα αριστερά,
- τα 2 πρώτα (από αριστερά, 7 και 6) θα χαθούν,
- τα δυο τελευταία θα γίνουν 0.
- Ο τύπος του αποτελέσματος είναι ίδιος με τον τύπο του πρώτου ορίσματος.

Έτσι, στη θέση *y* θα αποθηκευτούν τα εξής:

7 6 5 4 3 2 1 0
0 0 1 1 0 1 0 0

Σχεδόν παρομοίως γίνεται και η ολίσθηση δεξιά. Αν δώσουμε την εντολή `y = x >> 2` (ολίσθησε δεξιά κατά 2 δυαδικά ψηφία), θα πάρουμε στη *y*:

7 6 5 4 3 2 1 0
0 0 0 1 0 0 1 1

Το «σχεδόν» τι αφορά; Το «γέμισμα» με μηδενικά:

- *Αν ο τύπος της x είναι* `unsigned` *τότε οι πρώτες θέσεις που «αδειάζουν» θα «γεμίσουν» με μηδενικά.*
- *Αν ο τύπος της x δεν είναι* `unsigned` *και το ψηφίο προσήμου είναι* "1" *το πώς θα γεμίσουν οι θέσεις που αδειάζουν μπορεί να αλλάζει από τον έναν μεταγλωττιστή στον άλλον.*

Ορίζονται και οι σχετικές συντομογραφίες για την εκχώρηση:

- Αντί για "`x = x << N`" μπορείς να γράφεις "`x <<= N`" και
- αντί για "`x = x >> N`" μπορείς να γράφεις "`x >>= N`".

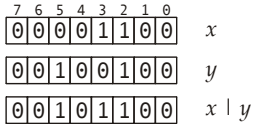
Στα παραδείγματα αυτά με την ολίσθηση χάνονται "`1`". Για να δούμε τι συμβαίνει αν δεν έχουμε τέτοιες απώλειες. Ας πούμε ότι στο *x* έχουμε βάλει την τιμή 12, οπότε, όπως είπαμε, η εσωτερική παράσταση είναι: 7 6 5 4 3 2 1 0 **0 0 0 0 1 1 0 0**.

Μετά τη "`y = x << 2`" η *y* γίνεται: 7 6 5 4 3 2 1 0 **0 0 1 1 0 0 0 0** .που είναι $2^5+2^4 = 48 = 12\times4$, ενώ μετά τη "`y = x >> 2`" η *y* γίνεται: 7 6 5 4 3 2 1 0 **0 0 0 0 0 0 1 1** .που είναι $2^1+2^0 = 3 = 12/4$,

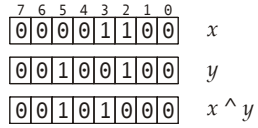
Δηλαδή, μπορούμε να πούμε:

`x << N` σημαίνει $x \times 2^N$ και `x >> N` σημαίνει x / 2^N ;

Ναι,



Σχ. 17-3 Ψηφιοπράξη "`|`" (**or**).



Σχ. 17-4 Ψηφιοπράξη "`^`" (**xor**).

- αν δεν έχουμε υπερχείλιση (για τη "`<<`") και
- αν δεν έχουμε περιπλοκές από τα πρόσημα (`char`, `short`, `int`, `long`).

Αυτές οι δυο πράξεις μας δίνουν τη δυνατότητα να γράφουμε από την C++ δυο εντολές του επεξεργαστή. Είναι λοιπόν πολύ ταχύτερες από τις αντίστοιχες αριθμητικές, αλλά *μην αρχίσεις να σκέφτεσαι να κάνεις έτσι πολλαπλασιασμούς και διαιρέσεις ακεραίων*: χρειάζονται προσοχή στη χρήση και οι πιθανότητες για λάθη είναι πάρα πολλές.

Παρατήρηση: ▶

Ο προσεκτικός αναγνώστης, διαβάζοντας για τους τελεστές ολίσθησης, "`<<`" και "`>>`", θα πρέπει να ανησύχησε: πώς δεν γίνεται μπέρδεμα με τους ίδιους (οπτικώς) τελεστές που χρησιμοποιούμε για γράψιμο και διάβασμα τιμών; Οι τελεστές ολίσθησης περιμένουν δύο ορίσματα που είναι ακέραιοι αριθμοί, ενώ οι τελεστές εισόδου/εξόδου χρειάζονται αριστερά κάποιο ρεύμα. Βεβαίως, σε ορισμένες περιπτώσεις χρειάζεται λίγη προσοχή. Αν γράψεις:

```
int x = 12;
cout << x << 2 << endl;
cout << (x << 2) << endl;
```

θα πάρεις αποτέλεσμα:

```
122
48
```

Το 122 προέρχεται από την πρώτη εντολή εκτύπωσης, που λέει: γράψε την τιμή της x, που είναι 12 και στη συνέχεια γράψε και το 2· έτσι παίρνουμε αυτό το "122". Η δεύτερη εντολή λέει: τύπωσε το αποτέλεσμα της πράξης "`x << 2`", που, όπως είδαμε, είναι 48.◀

Η C++ μας επιτρέπει ακόμη τις πράξεις "`~`", "`&`", "`|`" και "`^`" μεταξύ τιμών ακέραιων τύπων (αντίστοιχες των "`!`", "`&&`", "`||`" και "`!=`" μεταξύ λογικών τιμών). Αν οι x, y είναι τύπου `unsigned char`:

- Η "`~`x" (Σχ. 17-1) προκύπτει από τη x με αλλαγή κάθε ψηφίου "`0`" σε "`1`" και κάθε ψηφίου "`1`" σε "`0`" (συμπλήρωμα ως προς 1).
- Η "x `&` y" προκύπτει από τις x και y με **and** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-2 βλέπεις ότι το μοναδικό "`1`" που προκύπτει είναι στο ψηφίο 2 διότι στη θέση αυτήν έχουν "`1`" και η x και η y.
- Η "x `|` y" προκύπτει από τις x και y με **or** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-3 βλέπεις ότι η x `|` y έχει "`1`" στις θέσεις
  - 2 που έχουν "`1`" και η x και η y,
  - 3 όπου έχει "`1`" η x και
  - 5 όπου έχει "`1`" η y.

  Αν δεν έχουμε πρόσημα και "`1`" στην ίδια θέση τότε το x `|` y είναι το ίδιο με το x `+` y. Π.χ. αν στη y είχαμε "`00100010`" (παράσταση του 34_{10}) τότε το x `|` y είναι "`00101110`" που είναι παράσταση του $46_{10} = 12 + 34$.
- Η "x `^` y" προκύπτει από τις x και y με **xor** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-4 βλέπεις ότι η x `^` y έχει "`1`" στις θέσεις
  - 3 όπου έχει "`1`" μόνον η x και
  - 5 όπου έχει "`1`" μόνον η y.

  Για τους τρεις διμελείς ορίζονται συντομογραφίες εκχώρησης:
- Αντί για "`x = x & y`" μπορείς να γράφεις "`x &= y`",

- αντί για "`x = x | y`" μπορείς να γράφεις "`x |= y`" και
- αντί για "`x = x ^ y`" μπορείς να γράφεις "`x ^= y`".

Εδώ όμως χρειάζεται και πάλι προσοχή: Ας πούμε ότι έχεις:

```
short int m( 3 );
char c( -5 );
```

και θέλεις να υπολογίσεις το: `m |= c`. Η *c* έχει εσωτερική παράσταση: "`111110 11`". Για να γίνει η ψηφιοπράξη "`|`" θα πρέπει η *m* και η *c* να έχουν εσωτερική παράσταση με το ίδιο πλήθος ψηφίων. Η *m* «προωθείται» σε `short int` με εσωτερική παράσταση

"`1111111111111011`" (= -5 σε `short int`)!!!

Μάλλον δεν είναι αυτό που θέλεις. Αν είχες δηλώσει:

```
unsigned char c( 251 );
```

η *c* θα είχε την ίδια εσωτερική παράσταση (11111011) και με την προώθηση θα γίνονταν "`0000000011111011`".

Μετά από όσα είπαμε μπορείς να καταλάβεις τη σύσταση του (CERT 2009):[4]

♦ ***Χρησιμοποίησε τους τελεστές ψηφιοπράξεων μόνο με ορίσματα τύπων*** `unsigned`.[5]

Ας δούμε τώρα ένα παράδειγμα χρήσης των "`<<`" και "`&`".

Παράδειγμα

Θέλουμε να γράψουμε μια:

```
int bitValue( unsigned char b, int pos )
```

που θα μας επιστρέφει την τιμή του ψηφίου στη θέση *pos* της ψηφιολέξης *b*.

Πώς θα σκεφτούμε; Ας πούμε ότι έχουμε δηλώσει: `unsigned char t` και η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`1`". Πώς θα είναι τα ψηφία της `b & t`; Αφού η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* και αφού ξέρουμε ότι *P* `&& false` ≡ `false`, το ίδιο θα ισχύει και για την `b & t`: θα έχει "`0`" σε όλες τις θέσεις εκτός από την *pos*. Στη θέση *pos*, όπου η *t* έχει "`1`":

- Αν η *b* έχει "`0`" τότε και η `b & t` θα πάρει "`0`" και θα έχει τιμή 0 (μηδέν) αφού θα έχει παντού μηδενικά.
- Αν η *b* έχει "`1`" τότε και η `b & t` θα πάρει "`1`" και θα έχει τιμή ≠ 0 αφού θα έχει ένα μη μηδενικό ψηφίο.

Και πώς θα δώσουμε στην *t* την τιμή που θέλουμε; Έτσι:

```
unsigned char t( 1 );
t = t << pos;
```

δηλαδή:

- Δίνοντας στην *t* τιμή 1 βάζουμε 1 στο ψηφίο 0 της *t* και 0 σε όλα τα άλλα και
- με την "`t = t << pos`" μεταφέρουμε το 1, κατά *pos* θέσεις προς τα αριστερά, ενώ σε όλες τις άλλες θέσεις υπάρχουν 0.

Πριν γράψουμε τη συνάρτησή να παρατηρήσουμε ότι ενώ ορίζεται για όλες τις τιμές της *b*, δεν ορίζεται για τιμές της *pos* < 0 ή > 7· δηλαδή δεν είναι ολική.

Να λοιπόν πώς θα είναι η

```
int bitValue( unsigned char b, int pos )
{
   if ( pos < 0 || 7 < pos )   throw pos;

   unsigned char t( 1 );
   t <<= pos;
```

---

[4] Σύσταση *INT*13: "Use bitwise operators only on unsigned operands."

[5] Αν μελετάς σωστά αυτό το κεφάλαιο, θα έχεις ήδη παραβιάσει τη συσταση και θα την παραβιάσεις αρκετές φορές ακόμη για να δεις εσωτερικές παραστάσεις αρνητικών αριθμών. Δεν πειράζει, αφού είναι για εκπαιδευτικούς λόγους!

```
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Αν θέλεις να δεις την εσωτερική παράσταση του (`unsigned char`) "`12`" δηλώνεις

```
   unsigned char x( 12 );
```

ζητάς:

```
   for ( int pos(7); pos >= 0; --pos )
      cout << bitValue( x, pos );
   cout << endl;
```

και παίρνεις:

```
00001100
```

Με χρήση της *bitValue()* μπορούμε να γράψουμε τη:

```
void display( ostream& tout, unsigned char b )
{
   for ( int pos(7); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

που σου είναι χρήσιμη αν μελετάς σοβαρά αυτό το κεφάλαιο.

Το παρακάτω πρόγραμμα μας δίνει τις εσωτερικές παραστάσεις των ακεραίων 12 και 20 χρησιμοποιώντας την *display()*:

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

int bitValue( unsigned char b, int pos );
void display( ostream& tout, unsigned char b );

int main()
{
   unsigned char  x;
   int   k;

   try
   {
      x = 12;
      display( cout, x );  cout << endl;
      x = 20;
      display( cout, x );  cout << endl;
   }
   catch ( int& p )
   {
      cout << " η bitValue κλήθηκε με δεύτερο όρισμα "
           << p << endl;
   }
} // main
```

17.6 Ψηφιοχάρτες και Συνηθισμένες Πράξεις

Παρ' όλο που στα παραδείγματά μας, μέχρι τώρα, το τελικό μας ενδιαφέρον φαίνεται να βρίσκεται στην τιμή της ψηφιολέξης ή, γενικώς, της συνολικής παράστασης, η διαχείριση δυαδικών ψηφίων είναι ενδιαφέρουσα καθ' εαυτή. Με τη διαχείριση δυαδικών ψηφίων, εκτός άλλων εφαρμογών, μπορούμε να υλοποιήσουμε σύνολα, στα οποία μας ενδιαφέρει μόνον η πληροφορία *ανήκει* ("1") ή *δεν ανήκει* ("0"). Σε τέτοιες περιπτώσεις χρησιμοποιούμε πίνακες ακέραιων τιμών, π.χ. τύπου `unsigned long`, που όμως τις βλέπουμε ως **ψηφιοσύνολα** (bitsets) ή **ψηφιοχάρτες** (bitmaps)[6].

[6] Ο όρος *bitmap* χρησιμοποιείται και σε έναν τρόπο παράστασης γραφικών (bitmap graphics).

Στη συνέχεια δίνουμε μερικές συναρτήσεις –για την ακρίβεια: περιγράμματα συναρτησεων– για μερικές πολύ συνηθισμένες περιπτώσεις διαχείρισης ψηφιοπινάκων. Μπορεί να κληθούν με τύπο πρώτης παραμέτρου (*T*) κάποιον από τους `int`, `unsigned int`, `long`, `unsigned long`, `short`, `unsigned short`, `char`, `unsigned char`, `wchar_t`, `long long`, `unsigned long long`.

Μερικές από αυτές ρίχνουν εξαίρεση τύπου:

```
struct BitmapXptn
{
   enum { outOfRange, paramErr };
   char funcName[100];
   int  errorCode;
   int  errVal1, errVal2;
   BitmapXptn( const char* mn, int ec, int v1, int v2 = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = v1;  errVal2 = v2;  }
}; // BitmapXptn
```

Προσοχή! Στα σχόλια τεκμηρίωσης (και μόνο) των περιγραμμάτων που δίνουμε στη συνέχεια θα χρησιμοποιούμε τον συμβολισμό `b[pos]` για το δυαδικό ψηφίο της *b* στη θέση *pos*. *Ο συμβολισμός αυτός δεν είναι δεκτός από τη C++ για τέτοια χρήση.*

17.6.1 Τιμή Δυαδικού Ψηφίου

Ξεκινούμε μετατρέποντας σε περίγραμμα τη *bitValue* που γράψαμε πιο πριν:

```
template < typename T >  int bitValue( T b, int pos )
```

Η βασική διαφορά είναι ότι τώρα το τελευταίο δυαδικό ψηφίο δεν βρίσκεται στη θέση 7, αλλά στη θέση

$$8\texttt{(sizeof}\ b\texttt{)} - 1$$

Έτσι έχουμε:

```
// bitValue -- επιστρέφει τη b[pos]
//             Προϋπόθεση: 0 <= pos < 8*(sizeof b)
template < typename T >
int bitValue( T b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) <= pos )
      throw BitmapXptn( "bitValue", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Μετατρέπουμε σε περίγραμμα και τη *display()* για να μπορείς να κάνεις τις δοκιμές σου:[7]

```
template < typename T >
void display( ostream& tout, T b )
{
   int lastb( 8*(sizeof b) - 1 );

   for ( int pos(lastb); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

[7] Κανονικώς θα πρέπει να ελέγχουμε αν είναι ανοικτό το ρεύμα, αν το γράψιμο έγινε επιτυχώς και να ρίχνουμε τις αντίστοιχες εξαιρέσεις.

17.6.2 Βάλε Τιμή 1 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void setBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "`1`" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Φυσικά θα πρέπει να υπάρχει «*δυαδικό ψηφίο της θέσης pos*», δηλαδή:

$$0 \texttt{ <= } pos \texttt{ < } 8(\texttt{sizeof } b)$$

Πώς θα πετύχουμε το στόχο μας; Όπως μάθαμε, μετά τις

```
   T t( 1 );
   t <<= pos;
```

η *t* έχει "`0`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`1`". Η τιμή της παράστασης `b | t` θα έχει:

- "`1`" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "`1`" και ξέρουμε ότι *P* `|| true` ≡ `true`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "`0`" και ξέρουμε ότι *P* `|| false` ≡ *P*.

Να λοιπόν το περίγραμμα της συνάρτησης:

```
// setBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//           "1" στη θέση pos
//           Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//           Απαίτηση:
//           βτελ[pos] == 1 && για κάθε k!=pos: βτελ[k]==βαρχ[k]
template < typename T >
void setBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "setBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   b |= t;
} // setBit
```

Ας δούμε δύο παραδείγματα χρήσης:

Παράδειγμα 1

Σε μια μεταβλητή *si* τύπου `unsigned short int` (16 δυαδικά ψηφία) θέλουμε να έχουμε όλα τα ψηφία "`0`" εκτός από αυτά που βρίσκονται στις θέσεις 10, 3 και 9:

```
      si = 0;
      setBit( si, 10);  setBit( si, 3);  setBit( si, 9);
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
0000011000001000
```

Παράδειγμα 2

Σε μια μεταβλητή *si* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή μιας άλλης μεταβλητής *uc* τύπου `unsigned char`. Θέλουμε ακόμη, το ψηφίο 13 της *si* να έχει τιμή "`1`":

```
      display( cout, uc );  cout << endl;
      si = uc;
      setBit( si, 13 );
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
11111011
0010000011111011
```

🖈🖈🖈

Παρατήρηση: ▶

Ένα εύλογο ερώτημα που μπορεί να έχουν πολλοί είναι το εξής: Γιατί να μην ελέγξουμε αν το συγκεκριμένο ψηφίο είναι ήδη "`1`"; Στην περίπτωση αυτή δεν χρειάζεται να κάνουμε οτιδήποτε. Ας το δούμε· ο έλεγχος θα γίνει όπως είδαμε στη *bitValue()*:

```
   T t( 1 );
   t <<= pos;
   if ( (b & t) == 0 )  b |= t;
```

Δηλαδή:

- Σε κάθε περίπτωση έχουμε υπολογισμό της "`b & t`" και της **if** και
- Όταν το ψηφίο δεν έχει τιμή "`1`" εκτέλεση και της "`b |= t`".

Προφανώς ο τρόπος που επιλέξαμε –υπολογισμός μόνον της "`b |= t`"– είναι σαφώς πιο συμφέρων.◀

17.6.3 Βάλε Τιμή 0 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void clearBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Η λύση στο πρόβλημά μας μπορεί να προκύψει από τη λύση στο προηγούμενο πρόβλημα αν εναλλάξουμε τα "`0`" και "`1`" καθώς και τα **and** και **or**: Πράγματι, ας πούμε ότι η *t* έχει "`1`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`0`". Η τιμή της παράστασης `b & t` θα έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "`0`" και ξέρουμε ότι *P* `&& false` ≡ `false`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "`1`" και ξέρουμε ότι *P* `&& true` ≡ *P*.

Και πώς κάνουμε την t να «*έχει "`1`" σε όλες τις θέσεις εκτός από την pos όπου έχει "`0`"*»; Αυτό είναι απλό:

```
   T t( 1 );
   t <<= pos;  t = ~t;
```

Να λοιπόν η

```
// clearBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//             "0" στη θέση pos
//             Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//             Απαίτηση:
//             βτελ[pos] == 0 && για κάθε k!=pos: βτελ[k]==βαρχ[k]
template < typename T >
void clearBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "clearBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;  t = ~t;
   b &= t;
} // clearBit
```

Παράδειγμα

Σε μια μεταβλητή *m* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή της *si* του Παραδ. 2, της προηγούμενης παραγράφου, με τη διαφορά ότι η *m* θα έχει "`0`" σε όλες τις άρτιες θέσεις:

```
      m = si;
      for ( int k(0); k < 8*sizeof(m); k += 2 )
         clearBit( m, k );
      display( cout, m );  cout << endl;
```

Αποτέλεσμα:

```
0010000010101010
```

17.6.4 Πλήθος "1"

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  size_t count1( T b )
```

που θα επιστρέφει ως τιμή το πλήθος των δυαδικών ψηφίων της που έχουν τιμή "`1`" σε τιμή *b* τύπου *T*.

Θα μπορούσαμε να καλέσουμε 8(`sizeof` *b*) φορές. Αντί για αυτό εξετάζουμε τόσες φορές την τιμή της `b & t` όπου η *t* –τύπου *T*– έχει μόνο ένα "`1`". Κάθε φορά το "`1`" μετατοπίζεται ώστε να περάσει από όλες τις θέσεις της *t*.

```
// count1 -- επιστρέφει το πλήθος των ψηφίων της b με τιμή 1
//           Προϋπόθεση: true
//           Απαίτηση: fv == πληθος ψηφίων της b με τιμή 1
template < typename T >
size_t count1( T b )
{
   const size_t lastb( 8*(sizeof b) - 1 );
   T t( 1 );
   size_t fv( 0 );

   for ( int k(0); k <= lastb; ++k )
   {
      if ( (b & t) != 0 )  ++fv;
      t <<= 1;
   }
   return fv;
} // count1
```

Αν η *b* υλοποιεί κάποιο σύνολο ("`1`": «ανήκει»), η *count*1 μας δίνει τον *πληθάριθμο* του συνόλου.

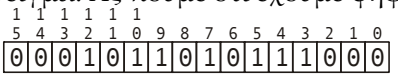
17.6.5 Μέρος Ψηφιοχάρτη

Θέλουμε ένα περίγραμμα συνάρτησης:

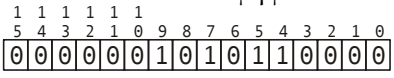
```
template < typename T >  T part( T b, int pos1, int pos2 )
```

που θα επιστρέφει ως τιμή τύπου *T* τα ψηφία της b από ***pos*1** μέχρι και ***pos*2**.

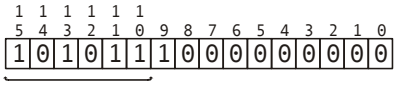
Να το δούμε με ένα παράδειγμα: Ας πούμε ότι έχουμε ψηφιοχάρτη 16 ψηφίων:



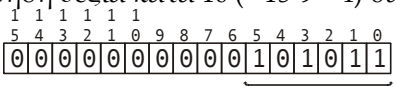
Θέλουμε έναν ψηφιοχάρτη με το ίδιο μέγεθος που θα έχει το υπογραμμισμένο κομμάτι δηλαδή τα ψηφία από 4 μέχρι 9 και όλα τα άλλα ψηφία 0:



Πρέπει δηλαδή να μηδενίσουμε τα ψηφία από 0 μέχρι 3 και από 10 μέχρι 15. Αυτό μπορεί να γίνει με την *clearBit*· μπορεί όμως να γίνει και ταχύτερα με τις πράξεις ολίσθησης. Αν στον αρχικό ψηφιοχάρτη κάνουμε μια ολίσθηση αριστερά και 6 (=15-9) θέσεις θα πάρουμε:



Αν σε αυτό κάνουμε ολίσθηση δεξιά κατά 10 (= 15-9 + 4) θα πάρουμε:



Τέλος, με ολίσθηση αριστερά κατά 4 θέσεις παίρνουμε αυτό που θέλουμε.

Αν πάρουμε υπόψη μας ότι: αντί για 15 έχουμε (γενικώς) το 8(`sizeof` *b*)–1, 4 είναι το *pos*1 και 9 είναι το *pos*2 έχουμε:

```
// part -- επιστρέφει το μέρος της b
//         με τα ψηφία της από pos1 μέχρι και pos2
//         Προϋπόθεση: 0 <= pos1 <= pos2 < 8*(sizeof v)
//         Απαίτηση: για κάθε k: 0..pos1-1 βτελ[k]==0 &&
//                   για κάθε k: pos1..pos2 βτελ[k]==βαρχ[k] &&
//                   για κάθε k: pos2+1..8*(sizeof v)-1 βτελ[k]==0
template < typename T >
T part( T b, int pos1, int pos2 )
{
   const size_t lastb( 8*(sizeof b) - 1 );

   if ( pos2 < pos1 )
      throw BitmapXptn( "part", BitmapXptn::paramErr, pos1, pos2 );
   if ( pos1 < 0 || pos2 < 0 || lastb < pos1 || lastb < pos2 )
      throw BitmapXptn( "part", BitmapXptn::outOfRange, pos1, pos2 );
// 0 <= pos1 <= pos2 <= lastb
   b <<= (lastb - pos2);
   b >>= (lastb - pos2 + pos1);
   b <<= pos1;
   return b;
} // part
```

Παράδειγμα

Αν η *si* είναι τύπου `unsigned short int` οι

```
      display( cout, si );  cout << endl;
      setBit( si, 12);
      unsigned short int sip( part(si, 4, 9) );
      display( cout, sip );  cout << endl;
```

δίνουν:

```
0000011010111000
0000001010110000
```

Και δύο λόγια για την πρώτη εξαίρεση: Για ορισμένες εφαρμογές το *pos*2 < *pos*1 δεν είναι και τόσο παράνομο. Απλώς στην περίπτωση αυτήν η συνάρτηση θα πρέπει να επιστρέφει *T*(0). Διαλέγεις και παίρνεις.

17.7 Τύποι *bitmask*

Πρωτοείδαμε τον τελεστή "`|`" στο Κεφ. 8, όταν συζητούσαμε για άνοιγμα ρεύματος. Ας πούμε ότι είχαμε να γράψουμε εμείς μια συνάρτηση που να ανοίγει ένα ρεύμα προς/από αρχείο. Γυρίζουμε λοιπόν στην §8.12 για να θυμηθούμε τα συστατικά του τρόπου ανοίγματος και καταλαβαίνουμε ότι (αν δεν θέλουμε να βάλουμε 6 ξεχωριστές παραμέτρους) θα πρέπει να βάλουμε μια παράμετρο τύπου:

```
struct OpenFlags
```

```
{
   bool app;
   bool ate;
   bool binary;
   bool in;
   bool out;
   bool trunc;
}; // OpenFlags
```

Επειδή μια τέτοια παράμετρος πιάνει 6 ψηφιολέξεις μπορούμε να κάνουμε οικονομία χρησιμοποιώντας ψηφιοπεδία:

```
struct OpenFlagsBF
{
   bool app:    1;
   bool ate:    1;
   bool binary: 1;
   bool in:     1;
   bool out:    1;
   bool trunc:  1;
}; // OpenFlagsBF
```

Μια τιμή τύπου *OpenFlagsBF* δεν χρειάζεται πάνω από μια ψηφιολέξη.

Σε αυτό το κεφάλαιο μάθαμε ότι μπορούμε να χειριστούμε το κάθε δυαδικό ψηφίο μιας τιμής ακέραιου τύπου. Και αφού οι τιμές που μπορεί να παίρνει είναι "`0`" ή "`1`" μπορούμε να το χειριστούμε ως τιμή τύπου `bool`. Έτσι, αντί για μεταβλητή τύπου *OpenFlagsBF* μας αρκεί μια μεταβλητή τύπου `unsigned char` (τη «μικρότερη» με 6 δυαδικά ψηφία). Πράγματι, ας πούμε ότι σε μια τέτοια τιμή ορίζουμε ότι το δυαδικό ψηφίο 0 ως σημαία για το "`app`", το ψηφίο 1 για το "`ate`", ..., το ψηφίο 5 για το "`trunc`". Για να αποφύγουμε λάθη δηλώνουμε τις σταθερές:

```
const unsigned char appPos = 0, atePos = 1, binaryPos = 2,
                    inPos = 3, outPos = 4, truncPos = 5;
```

Έτσι, για να πούμε ότι θέλουμε να ανοίξουμε ένα ρεύμα *in*, *out* και *binary* σε μια μεταβλητή:

```
   unsigned char om;
```

βάζουμε:

```
   om = 0;
   setBit( om, inPos );  setBit( om, outPos );
   setBit( om, binaryPos );
```

Ελέγχουμε αν, ας πούμε, το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `bitValue(om, binaryPos) == 1`.

Μπορούμε να τα καταφέρουμε χωρίς τη *setBit*; Ναι! Δες μια άλλη, διαφορετική, ομάδα σταθερών:

```
const unsigned char app = 1, ate = (app << 1),
                    binary = (app << 2), in = (app << 3),
                    out = (app << 4), trunc = (app << 5);
```

Αυτές δεν κρατούν τη θέση του ψηφίου που αντιστοιχεί στην κάθε σημαία *αλλά σε εκείνη ακριβώς τη θέση έχουν το μοναδικό* "`1`".[8] Με αυτές δίνουμε στη *om* την τιμή που θέλουμε έτσι:

```
   om = in | out | binary;
```

Αφού οι σταθερές έχουν τα "`1`" σε διαφορετικές θέσεις θα μπορούσαμε να γράψουμε και:

```
   om = in + out + binary;
```

[8] Θα μπορούσαμε να είχαμε γράψει:

```
const unsigned char app = 1, ate = (1 << 1), binary = (1 << 2),
                    in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

αλλά αυτό δεν είναι και πολύ καλή ιδέα!

Πάντως μπορούμε να κάνουμε και αυτό που κάνουμε στη *setBit*:

```
om = 0;
om |= in;  om |= out;  om |= binary;
```

Αν δεν έχουμε τη θέση πώς θα ελέγχουμε αν κάποιο ψηφίο έχει τιμή 1. Για να δούμε, ας πούμε, αν το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `(om & binary) != 0`.

Τα παραπάνω είναι ένας τρόπος υλοποίησης ενός τύπου bitmask.

Ένας **τύπος bitmask** έχει τα εξής χαρακτηριστικά:

- N+1 σταθερές c_0 = `1`, c_1 = `(1 << 1)`, …, c_N`(1 << N)`.
- Αν v_1, v_2 τιμές του τύπου τότε και οι v_1 `&` v_2, v_1 `|` v_2, v_1 `^` v_2, `~`v_1 είναι τιμές του τύπου.
- Εκτός από τις ψηφιοπράξεις "`&`", "`|`", "`^`", "`~`" ορίζονται και οι «συντομογραφίες» εκχώρησης "`&=`", "`|=`", "`^=`".

Μπορούμε να υλοποιήσουμε έναν τέτοιον τύπο με τρεις τρόπους.

1. Ο πρώτος είναι *με κατάλληλο ακέραιο τύπο*, δηλαδή τύπο που οι τιμές του να παριστάνονται σε N+1 δυαδικά ψηφία τουλάχιστον. Παραπάνω, είδαμε παράδειγμα τέτοιας υλοποίησης. Αφού στην περίπτωσή μας N = 5 οποιοσδήποτε ακέραιος τύπος είναι κατάλληλος. Επιλέγουμε τον «μικρότερο»:

```
typedef unsigned char OpenMode;
const OpenMode app = 1, ate = (1 << 1), binary = (1 << 2),
               in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

   Όλες οι ψηφιοπράξεις που μας ενδιαφέρουν είναι ήδη ορισμένες.

2. Ο δεύτερος τρόπος υλοποίησης είναι με κάποιον *απαριθμητό τύπο*. Για το παράδειγμά μας ορίζουμε:

```
enum OpenMode
{  app = 1, ate = (1 << 1), binary = (1 << 2), in = (1 << 3),
   out = (1 << 4), trunc = (1 << 5)  };
```

   Αν δεν διάβασες την §17.4 να πούμε ότι θα πρέπει να ξεχάσεις αυτά που μάθαμε για τους απαριθμητούς τύπους: σε μια μεταβλητή τύπου *OpenMode* μπορούμε (με ή χωρίς διαμαρτυρίες από τον μεταγλωττιστή) να βάλουμε τιμές που δεν υπάρχουν στην παραπάνω απαρίθμηση.

   Αν και ορισμένοι μεταγλωττιστές θα δεχτούν –με διαμαρτυρίες (warnings)– να κάνουν ψηφιοπράξεις με αυτές τις σταθερές, το σωστό είναι να επιφορτώσεις τους τελεστές που είδαμε παραπάνω. Δίνουμε για το παράδειγμά μας την επιφόρτωση των "`&`" και "`&=`":

```
OpenMode operator&( OpenMode x, OpenMode y )
{
   return static_cast<OpenMode>( static_cast<unsigned char>(x) &
                                 static_cast<unsigned char>(y) );
} // operator&( OpenMode

OpenMode& operator&=( OpenMode& x , OpenMode y )
{
   x = x & y;
   return x;
} // operator&=( OpenMode
```

   Όπως βλέπεις, ενώ με τη χρήση της απαρίθμησης δεν χρειάστηκε να επιλέξουμε ακέραιο τύπο, χρειάζεται τώρα για την επιφόρτωση των τελεστών.

3. Ο τρίτος τρόπος υλοποίησης είναι *με χρήση του περιγράμματος bitset* της STL. Θα τον δούμε αργότερα.

Οι τύποι bitmask χρησιμοποιούνται πολύ συχνά σε προγράμματα και βιβλιοθήκες C++ (και C).

17.8 Αριθμητικές Πράξεις και Ψηφιοπράξεις

Σε υποσημείωση της §8.6 λέγαμε για την "`ios_base::in|ios_base::out`" «*αυτό μπορεί να το δεις και ως:* "`ios_base::in+ios_base::out`".» Στην προηγούμενη παράγραφο είδαμε γιατί μπορεί να γραφεί κάτι τέτοιο, τουλάχιστον για τον πρώτο τρόπο υλοποίησης ενός τύπου *bitmask*. Αλλά,

- για τον τρίτο τρόπο υλοποίησης, η πρόσθεση δεν είναι δεκτή ενώ
- για τον δεύτερο τρόπο θα πρέπει να βάλεις τυποθεωρήσεις ώστε να περνάει από όλους τους μεταγλωττιστές.

Φυσικά, μπορεί να το δεις γραμμένο κάπου αλλού αλλά εσύ *δεν θα πρέπει να το γράφεις* με βάση το εξής σκεπτικό: Αφού αυτό που κάνουμε είναι διαχείριση δυαδικών ψηφίων και όχι αριθμητική πράξη γιατί να βάλουμε το "+"; Μόνο και μόνο επειδή δουλεύει;

Στις προηγούμενες παραγράφους επισημάναμε ότι είναι επικίνδυνο και το αντίστροφο: να προσπαθήσεις να πάρεις αριθμητικά αποτελέσματα με ψηφιοπράξεις. Τα αριθμητικά αποτελέσματα θα τα παίρνεις με αριθμητικές πράξεις.

Καταλήγουμε λοιπόν στη σύσταση του (CERT 2009):[9]

♦ ***Απόφυγε να κάνεις ψηφιοπράξεις και αριθμητικές πράξεις στα ίδια δεδομένα.***

17.9 Παράσταση και Πράξεις στον Τύπο "`float`"

Όπως ο `int` σε σχέση με το σύνολο $\mathbb{Z}$ των ακεραίων, έτσι και ο τύπος `float`, σε σύγκριση με το σύνολο $\mathbb{R}$ των πραγματικών, έχει δυο σοβαρούς περιορισμούς:

- Υπάρχει μέγιστος και ελάχιστος αριθμός τύπου `float`.
- Κάθε αριθμός τύπου `float` παριστάνεται με πεπερασμένο πλήθος ψηφίων.

Συνήθως, έναν πραγματικό αριθμό x τον γράφουμε ως:

$\sigma\ \psi_n \psi_{n-1} \ldots \psi_0 . \psi_{-1} \psi_{-2} \ldots$

και με αυτό εννοούμε ότι:

$$x = \sigma(\psi_n 10^n + \psi_{n-1} 10^{n-1} \ldots + \psi_0 10^0 + \psi_{-1} 10^{-1} + \psi_{-2} 10^{-2} \ldots)$$

όπου σ: πρόσημο (+ ή −) και ψ_k: δεκαδικό ψηφίο.

Φυσικά, ο ΗΥ δεν μπορεί να αποθηκεύσει στην μνήμη του τα άπειρα ψηφία ενός άρρητου αριθμού. Κάθε αριθμός παριστάνεται με πεπερασμένο αριθμό ψηφίων. Έχουμε λοιπόν **απώλεια σημαντικών ψηφίων** (loss of significant digits). Αυτό το σφάλμα, που εισάγεται με την παράσταση των πραγματικών αριθμών, μεγαλώνει με την εκτέλεση των πράξεων μεταξύ τους.

Ας υποθέσουμε ότι δουλεύουμε σε έναν **δεκαδικό** υπολογιστή. Κάθε τιμή τύπου `float`, για να αποθηκευτεί, μετατρέπεται στη μορφή:

$$M \times 10^e \qquad (1)$$

και αποθηκεύονται η **μαντίσα** (mantissa) M και ο **εκθέτης** (exponent) e. Η μαντίσα έχει τη μορφή:

$$\sigma\ 0 . \psi_1 \psi_2 \psi_3 \psi_4 \psi_5$$

όπου σ το πρόσημο και ψ_k, $k = 1..5$ τα πέντε πιο σημαντικά ψηφία του αριθμού (με στρογγύλευση). Ο εκθέτης είναι διψήφιος ακέραιος και έχει τη μορφή:

$$\sigma\ \varepsilon_1 \varepsilon_2$$

Η παράσταση στη μορφή (1) γίνεται μονοσήμαντη αν κάνουμε τη σύμβαση ότι *το ψ_1 δεν μπορεί να είναι μηδέν*. Στην περίπτωση αυτήν λέμε ότι η παράσταση είναι **κανονικοποιημένη** (normalized). Θα παριστάνουμε την αποθηκευμένη πληροφορία, στη μορφή:

$$\sigma\psi_1 \psi_2 \psi_3 \psi_4 \psi_5 : \sigma\varepsilon_1 \varepsilon_2$$

---

[9] Σύσταση *INT*14: "*Avoid performing bitwise and arithmetic operations on the same data.*"

Αυτό είναι ένα παράδειγμα αποθήκευσης σε μορφή **κινητής υποδιαστολής** (floating point).

Το γνωστό μας $\pi = 3.1415926535...$, θα αποθηκευτεί ως:

+31416 : +01

(Θα γράφουμε π_f = +31416 : +01 ή π_f = 3.1416).

Η μέγιστη θετική τιμή που μπορεί να αποθηκευτεί είναι η:

+99999 : +99 (= 0.99999×10^{99})

και η ελάχιστη θετική τιμή:

+10000 : -99 (= 0.1×10^{-99})

Ένα χαρακτηριστικό της υλοποίησης του τύπου `float` είναι ο ελάχιστος θετικός ε που αν προστεθεί στο 1 μας δίνει τιμή μεγαλύτερη από 1:

$$\varepsilon = \min_{u>0}\{u \mid (1+u)_f \neq 1_f\}$$

Στον υπολογιστή μας, το 1 παριστάνεται ως:

+10000 : +01

και προφανώς η ελάχιστη αλλαγή που μπορούμε να του κάνουμε, είναι στο τελευταίο σημαντικό ψηφίο, κατά 1:

+10001 : +01 (= $0.10001 \times 10^{1} = 1.0001$)

Έχουμε λοιπόν ότι: $\varepsilon = 0.0001$.

Προσοχή όμως! Αυτό δεν σημαίνει ότι για κάθε $x \in$ `float` θα έχουμε και $(x + \varepsilon)_f \neq X_f$. Για παράδειγμα: $(10 + \varepsilon)_f = 10_f$ (Άσκ. 14-1). Πάντως το ε μας δείχνει πόσα σημαντικά ψηφία μπορούμε να παραστήσουμε: εφ' όσον μπορούμε να παραστήσουμε το $1+\varepsilon = 1.0001$ μπορούμε να παραστήσουμε 5 σημαντικά ψηφία. Αν αυτό φαίνεται τετριμμένο τώρα που δουλεύουμε στο δεκαδικό σύστημα, δεν είναι τετριμμένο όταν δουλεύουμε στο δυαδικό ή στα παράγωγά του, που δεν μας είναι και τόσο οικεία.

Μπορούμε λοιπόν να πούμε ότι ο τύπος `float` είναι υποσύνολο του συνόλου των πραγματικών:

- `float` $\subset \mathbb{R}$ ***(ακριβέστερα:*** `float` $\subset \mathbb{Q}$ ***(ρητοί))***

Η αποθήκευση πραγματικών τιμών στον υπολογιστή, είναι μια απεικόνιση από το σύνολο $\mathbb{R}$ στο υποσύνολό του `float`. Μερικές ιδιότητες αυτής της απεικόνισης θα δούμε στη συνέχεια.

Όπως φαίνεται από το παράδειγμά μας με το π, η αποθήκευση δεν γίνεται με ακρίβεια. Πάντως, για κάθε υπολογιστή, υπάρχει η εγγύηση ότι δυο τουλάχιστον πραγματικοί αριθμοί παριστάνονται με ακρίβεια: το 0 (μηδέν) και το 1 (ένα)!

Το 0 παριστάνεται ως:

+00000 : +00

και το 1 ως:

+10000 : +01 (= $0.1 \times 10^{1} = 1$)

- $\mathbf{0_f == 0}$ ***και*** $\mathbf{1_f == 1}$

Ο δείκτης $_f$ υποδηλώνει την παράσταση στο σύνολο `float`.

Αν μια πραγματική τιμή α παριστάνεται στο σύνολο `float` με την α_f, τότε και η $-\alpha$ παριστάνεται στο `float` και μάλιστα με την $-\alpha_f$:

$(-\alpha)_f == -\alpha_f$

Όπως είδαμε παραπάνω, η τιμή 3.1415926535..., θα αποθηκευτεί ως:

+31416 : +01

Αλλά με τον ίδιο τρόπο θα αποθηκευτεί και η τιμή 3.14156 και η 3.1416 κλπ. Γενικά:

- ***Αν οι τιμές*** α_1, α_2 ***παριστάνονται στον υπολογιστή με την ίδια τιμή*** $\tau \in$ `float`, ***τότε με την ίδια τιμή παριστάνονται όλες οι τιμές του διαστήματος*** $[\alpha_1, \alpha_2]$.

Από την ιδιότητα αυτή βγαίνουν τα εξής:

αν $\alpha > \beta$ τότε $\alpha_f \geq \beta_f$

αν $\alpha == \beta$ τότε $\alpha_f == \beta_f$

αν $\alpha < \beta$ τότε $\alpha_f \leq \beta_f$

Ας δούμε τώρα τι γίνεται με τις πράξεις.

Το πρώτο που θα δούμε είναι η **υπερχείλιση** και η **υποχείλιση**: είναι δυνατόν το αποτέλεσμα μιας πράξης να μην παριστάνεται γιατί είναι πολύ μεγάλο ή πολύ μικρό. Οι τιμές:

+60000:+99 (= 0.60000×10^{99})

και

+70000:+99 (= 0.70000×10^{99})

παριστάνονται στον υπολογιστή μας χωρίς πρόβλημα. Το άθροισμά τους όμως, $1.30000\times10^{99} = 0.13000\times10^{100}$ *δεν μπορεί να παρασταθεί* διότι είναι πολύ μεγάλο για τον υπολογιστή μας! Έχουμε δηλαδή **υπερχείλιση** (overflow). Τι θα κάνει ο υπολογιστής σε μια τέτοια περίπτωση; Σίγουρα θα σου δώσει μήνυμα για την κατάσταση που δημιουργήθηκε· στις περισσότερες περιπτώσεις θα σταματήσει και την εκτέλεση του προγράμματος.

Όπως καταλαβαίνεις, τέτοια αποτελέσματα μπορεί να βγουν και από τις τέσσερις πράξεις της αριθμητικής, όταν τις εκτελεί ο υπολογιστής. Δηλαδή:

- ***Το σύνολο `float` δεν είναι κλειστό ως προς τις πράξεις +, –, * και / .***

Καταλαβαίνεις ακόμη, ότι η υπερχείλιση στον τύπο `float` είναι λιγότερο επικίνδυνη από αυτήν του `int`, αφού αποφεύγεις την περίπτωση να συνεχιστεί η εκτέλεση του προγράμματος με τιμές που δεν έχουν νόημα. Έχει νόημα να γράψουμε κάτι σαν *addFloat*, για ασφαλή πρόσθεση τιμών `float`; Ναι, διότι συχνά ξέρεις τί πρέπει να κάνεις σε περίπτωση υπερχείλισης και οπωσδήποτε είναι ενοχλητικό να βλέπεις το πρόγραμμα να σταματάει, έστω και αν σου γνωστοποιεί τί έγινε.

Αν από τον:

+11000:-99 (= 0.11×10^{-99})

αφαιρέσουμε τον:

+10000:-99 (= 0.1×10^{-99})

το αποτέλεσμα $0.01\times10^{-99} = 0.1\times10^{-100}$ δεν μπορεί να παρασταθεί στον υπολογιστή μας, γιατί είναι πολύ μικρό! Στην περίπτωση αυτή λέμε ότι έχουμε **υποχείλιση** (underflow). Συνήθως, ο υπολογιστής σε μια τέτοια περίπτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!

Ας δούμε τώρα ένα άλλο επακόλουθο της πεπερασμένης παράστασης. Έστω ότι θέλουμε να προσθέσουμε με τον υπολογιστή μας τους αριθμούς 14.563 και 0.16773. Η αποθήκευσή τους θα γίνει με ακρίβεια:

+14563:+02 (= $0.14563\times10^{2} = 14.563$)

και

+16773:+00 (= $0.16773\times10^{0} = 0.16773$)

Για να τους προσθέσει ο υπολογιστής θα πρέπει πρώτα να τους μετασχηματίσει ώστε να έχουν τον ίδιο εκθέτη. Για την ακρίβεια μετασχηματίζει την τιμή με το μικρότερο εκθέτη (στρογγυλεύοντας σε πέντε θέσεις)[10]:

---

[10] Συνήθως, η Αριθμητική Μονάδα του υπολογιστή θα κάνει τις πράξεις με μεγαλύτερη ακρίβεια, αλλά η τιμή που θα μας επιστρέψει τελικά είναι σύμφωνη με τη μορφή που έχουμε στην αποθήκευση. Στην περίπτωσή μας, η δεύτερη τιμή θα γίνει:

+0016773:+02

Στη συνέχεια θα γίνει η πρόσθεση:

+1473073:+02

Αλλά, το αποτέλεσμα που θα είναι διαθέσιμο στο πρόγραμμά μας θα είναι:

+14731:+02

+00168 : +02

Στη συνέχεια κάνει την πρόσθεση:

+14731 : +02

Βλέπουμε λοιπόν, ότι και οι πράξεις εισάγουν **σφάλματα στρογγύλευσης** (roundoff errors). Πρόσεξε ότι, αν προσπαθούσαμε να προσθέσουμε στο 14.563 τον 0.0001, το αποτέλεσμα θα ήταν 14.563!

Θα πρέπει να έχει γίνει πια φανερό, ότι όπως ξεχωρίσαμε τις ακέραιες τιμές από τις παραστάσεις τους στον τύπο `int`, *θα πρέπει να ξεχωρίσουμε και τις πράξεις των πραγματικών από αυτές του υπολογιστή για τον τύπο* `float`. Θα παριστάνουμε λοιπόν με:

$+_f \quad -_f \quad *_f \quad /_f$

τις πράξεις:

$+ \quad - \quad \times \quad /$

όπως εκτελούνται στον υπολογιστή.

Στη συνέχεια θα δούμε μερικές «αναμενόμενες» ιδιότητες των πράξεων αλλά και μερικές «περίεργες». Το σύμβολο της ισότητας θα έχει πιο γενικό νόημα από ότι συνήθως: π_1 `==` π_2, όπου π_1, π_2 αριθμητικές παραστάσεις, θα σημαίνει ότι: αν μεν οι πράξεις γίνουν χωρίς πρόβλημα (υπερχείλιση ή υποχείλιση) οι τιμές των δύο παραστάσεων θα είναι ίσες· θα έχουμε υπερχείλιση ή υποχείλιση αριστερά αν και μόνον αν έχουμε το ίδιο πράγμα και δεξιά.

Η αντιμεταθετικότητα της πρόσθεσης και του πολλαπλασιασμού ισχύει στον τύπο `float`:

- ***Αν*** $a, b \in$ `float` ***τότε*** $a +_f b = b +_f a$ ***και*** $a *_f b = b *_f a$

Αυτές οι ισότητες ισχύουν με το γενικευμένο νόημα που δώσαμε πιο πάνω.

Η προσεταιριστικότητα της πρόσθεσης δεν ισχύει! Ας δούμε ένα

Παράδειγμα

Οι αριθμοί $a = 0.10000\times10^{94}$, $b = 0.55000\times10^{99}$, $c = -0.50000\times10^{99}$ παριστάνονται στον υπολογιστή μας με ακρίβεια. Το άθροισμα:

$$\begin{aligned}(a +_f b) +_f c &= (0.000001 +_f 0.55000)\times10^{99} +_f (-0.50000\times10^{99})\\ &= 0.55000\times10^{99} +_f (-0.50000\times10^{99})\\ &= 0.50000\times10^{98}\end{aligned}$$

δεν είναι ίσο με το:

$$\begin{aligned}a +_f (b +_f c) &= 0.10000\times10^{94} +_f (0.55000\times10^{99} +_f 0.50000\times10^{99})\\ &= 0.10000\times10^{94} +_f 0.05000\times10^{99}\\ &= 0.50001\times10^{98}\end{aligned}$$

Παρ' όλα αυτά, αν: $a, b \in$ `float` και $a \geq b \geq 0$, ισχύει η γνωστή μας ιδιότητα:

$(a -_f b) +_f b$ `==` a

δηλαδή: η πρόσθεση ακυρώνει την αφαίρεση.

Προβλήματα έχουμε και με την προσεταιριστικότητα του πολλαπλασιασμού: Αν πάρουμε: $a = 0.10000\times10^{60}$, $b = 0.10000\times10^{60}$, $c = 0.10000\times10^{-60}$ τότε το γινόμενο:

$$\begin{aligned}(a *_f b) *_f c &= (0.10000\times10^{60} *_f 0.10000\times10^{60}) *_f 0.10000\times10^{-60}\\ &= 0.10000\times10^{119} *_f 0.10000\times10^{-60} \quad \text{(υπερχείλιση!!)}\end{aligned}$$

δεν είναι ίσο με το:

$$\begin{aligned}a *_f (b *_f c) &= 0.10000\times10^{60} *_f (0.10000\times10^{60} *_f 0.10000\times10^{-60})\\ &= 0.10000\times10^{60} *_f 0.10000\times10^{-1}\\ &= 0.10000\times10^{58}\end{aligned}$$

Οι παραπάνω ενδεικτικές επισημάνσεις δεν εξαντλούν πλήρως τα προβλήματα του τύπου `float`, που ξεκινούν από την πεπερασμένη παράσταση. Αλλά δείχνουν μερικά χαρακτηριστικά σημεία που πρέπει να προσέχεις όταν γράφεις αριθμητικά προγράμματα.

17.9.1 Υπολογισμός Περιοδικής Συνάρτησης

Ας ξαναδούμε τώρα κάτι που μάθαμε παλιά υπό το φως αυτών που είδαμε παραπάνω.

Στην §7.8 μάθαμε να κάνουμε αναγωγή της τιμής του ορίσματος μιας περιοδικής συνάρτησης στο διάστημα ορισμού με τις εντολές:

```
    x0 = x;
    while ( x0 >= b )  x0 = x0 - T;
// (x0 < b) && (f(x0) == f(x))
    while ( x0 < a )   x0 = x0 + T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Τι θα γίνει αν η απόλυτη τιμή του x είναι πολύ μεγαλύτερη από την τιμή της περιόδου T; Στην περίπτωση αυτήν η εντολή `x0 = x0 - T` (ή `x0 = x0 + T`) *δεν αλλάζει την τιμή της* $x0$ και *η εκτέλεση της αντίστοιχης* ***while*** *δεν τελειώνει*. Θα γράφαμε πιο σωστά:

```
    x0 = x;
    if (x0 >= b)
    {
       if (x0 - T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 >= b)  x0 = x0 - T;
       // (x0 < b) && (f(x0) == f(x))
       }
    }
    else if (x0 < a)
    {
       if (x0 + T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 < a)   x0 = x0 + T;
       // (a ≤ x0 < b) && (f(x0) == f(x))
       }
    }
```

Θα πεις: καλύτερα τότε να δουλεύουμε με τον γρήγορο τρόπο αναγωγής. Ναι, αλλά πρόσεξε: είναι καλύτερο να γράψουμε:

```
    x0 = x - T*floor((x-a)/T);
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Όμως και στην περίπτωση αυτή, αν το $x - a$ είναι πολύ μεγαλύτερο από την περίοδο T τότε η $x0$ θα πάρει τιμή που δεν έχει και πολύ νόημα, πιθανότατα 0. Ακόμη, αν η περίοδος είναι μικρότερη από 1 είναι δυνατόν να έχουμε υπερχείλιση στη διαίρεση $(x - a)/T$.

17.10 Άλλοι Τύποι Κινητής Υποδιαστολής

Για να αντιμετωπισθούν τα προβλήματα του τύπου `float` η C++ –και άλλες γλώσσες προγραμματισμού– δίνουν στον προγραμματιστή άλλους αριθμητικούς τύπους με μερικές βελτιώσεις.

Η πιο συνηθισμένη περίπτωση είναι ο «διπλός `float`», που η C++ ονομάζει `double`. Οι τιμές αυτού του τύπου αποθηκεύονται σε χώρο (μνήμης) *διπλό* απ' όσον πιάνουν οι τιμές του τύπου `float`. Η αποθήκευση γίνεται συνήθως με κάποιον από τους παρακάτω τρόπους:

- Ο αριθμός θεωρείται σαν άθροισμα δύο τιμών τύπου `float`.
  - Η πρώτη έχει τα περισσότερο σημαντικά ψηφία του αριθμού και
  - η δεύτερη τα λιγότερο σημαντικά ψηφία.

  Για παράδειγμα, η τιμή 12.34567891234 θα γραφεί ως:

  $$12.345678912 = 12.345 + 0.00067891$$
  $$= 0.12345\times10^{2} + 0.67891\times10^{-3}$$

  και στις δυο θέσεις θα αποθηκευτούν τα:

  +12345:+02 και +67891:-03
- Ο δεύτερος τρόπος διαφέρει ως προς το περιεχόμενο της δεύτερης θέσης: εκεί απόθηκεύονται μόνο (τα λιγότερο) σημαντικά ψηφία. Δηλαδή, η δεύτερη θέση δεν είναι οργανωμένη όπως οι θέσεις τύπου `float` (δεν αποθηκεύεται εκθέτης). Για το παραπάνω παράδειγμα θα αποθηκευτούν τα εξής:

  +12345:+02 και 6789123
- Μια παραλλαγή του δεύτερου τρόπου, που επικρατεί στους νέους υπολογιστές, επιτρέπει και την αύξηση των θέσεων του εκθέτη. Στην περίπτωση αυτήν η περιοχή τιμών που μπορεί να παρασταθεί είναι ευρύτερη από αυτήν που παριστάνεται στον τύπο `float`. Σε μια τέτοια περίπτωση ο αριθμός μας θα αποθηκεύονταν ως:

  +1234:+002 και 5678912

  Ανακεφαλαιώνοντας, μπορούμε να πούμε για τον "διπλό `float`":
- Επιτρέπει την αποθήκευση περίπου διπλού πλήθους σημαντικών ψηφίων απ' ότι ο τύπος `float`.
- Μπορεί να επιτρέπει την παράσταση ευρύτερης περιοχής τιμών απ' ότι ο τύπος `float`, αλλά αυτό δεν είναι αναγκαίο.
- Κάθε τιμή του καταλαμβάνει διπλό χώρο μνήμης απ' ότι ο τύπος `float`.

  Φυσικά, τα προβλήματα του τύπου `float` δεν λύνονται· απλώς μετατοπίζονται.

17.11 Ο Τύπος "`float`" στο Δυαδικό Σύστημα

Μέχρι τώρα χρησιμοποιήσαμε το δεκαδικό σύστημα για να δούμε μερικά από αυτά που συμβαίνουν στους τύπους `float`. Αλλά οι ψηφιακοί υπολογιστές δουλεύουν συνήθως με το **δυαδικό** (binary) σύστημα και τα παράγωγά του **οκταδικό** (octal) και **δεκαεξαδικό** (hexadecimal). Τώρα θα δούμε μερικά από τα παραπάνω στο δυαδικό σύστημα.

Στο δυαδικό σύστημα η παράσταση μιας τιμής τύπου `float` γίνεται ως εξής:

$$M \times 2^{e}$$

Η **μαντίσα** M και ο **εκθέτης** e που αποθηκεύονται, είναι δυαδικοί αριθμοί. Ο εκθέτης είναι ακέραιος και έχει τη μορφή:

$$s\ \varepsilon_1\ \varepsilon_2 \ldots\ \varepsilon_L$$

όπου s πρόσημο και ε_k, $k = 1..L$ τα L δυαδικά ψηφία.

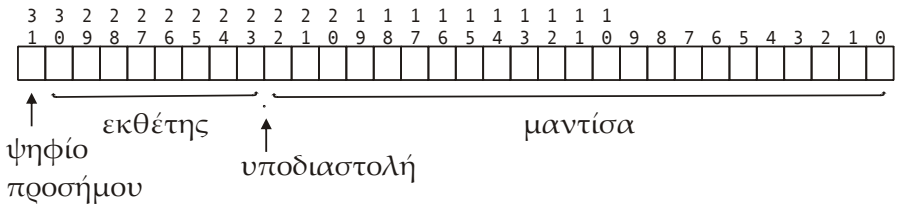
Η μαντίσα μπορεί να έχει τη μορφή:

$$s\ 0.\psi_1\psi_2\ldots\psi_N$$

ή

$$s\ \psi_1\psi_2\ldots\psi_N.0$$

Συνήθως η παράσταση κανονικοποιείται, δηλαδή *το ψ_1 δεν μπορεί να είναι μηδέν*. Στο δυαδικό σύστημα αυτό σημαίνει ότι $\psi_1 = 1$.



Σχ. 17-5 Παράσταση κινητής υποδιαστολής.

Μια τέτοια παράσταση βλέπουμε στο Σχ. 17-5.

Στο ψηφίο 31 αποθηκεύεται το πρόσημο του αριθμού –"**0**" για το "**+**", "**1**" για το "**-**". Στα ψηφία 30 - 23 αποθηκεύεται ο εκθέτης: Στο ψηφίο 30 το πρόσημό του (όπως παραπάνω) και στα ψηφία 29 - 23 η απόλυτη τιμή του. Στα ψηφία 22 - 0 αποθηκεύεται η μαντίσα. Το πρώτο της ψηφίο (22) είναι πάντοτε 1, εκτός αν ο αριθμός είναι 0 (μηδέν). Η υποδιαστολή νοείται ακριβώς πριν από το ψηφίο αυτό (22). Δηλαδή:

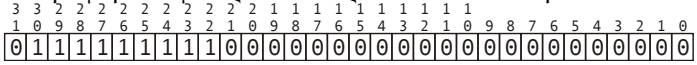
$$e=\begin{cases}\varepsilon_{29}\varepsilon_{28}\ldots\varepsilon_{23}, & \text{αν } \varepsilon_{30}=0\\ -\varepsilon_{29}\varepsilon_{28}\ldots\varepsilon_{23}, & \text{αν } \varepsilon_{30}=1\end{cases}$$

$$M=\begin{cases}0.\psi_{22}\psi_{21}\ldots\psi_{0}, & \text{αν } s_{31}=0\\ -0.\psi_{22}\psi_{21}\ldots\psi_{0}, & \text{αν } s_{31}=1\end{cases}$$

Ο εκθέτης μπορεί να παίρνει (ακέραιες) τιμές, από –127 μέχρι +127. Η ελάχιστη μη μηδενική τιμή της μαντίσας μπορεί να είναι $0.1_2 = 0.5_{10}$. Η μέγιστη σχηματίζεται όταν όλα τα ψηφία (22 - 0) είναι 1:

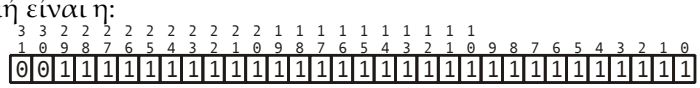
$2^{-1} + 2^{-2} + \ldots + 2^{-23} = 1 - 2^{-23} \approx 0.99999988 \approx 1$

Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί είναι η:



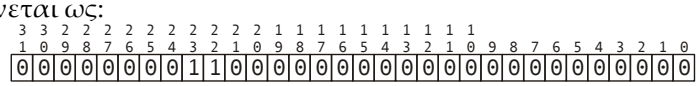
Δηλαδή, ελάχιστη μη-μηδενική μαντίσα ($0.1_2 = 0.5_{10}$) και ελάχιστος εκθέτης (-127). Προφανώς πρόκειται για το $2^{-128} \approx 0.29387e\text{-}38$.

Η μέγιστη τιμή είναι η:



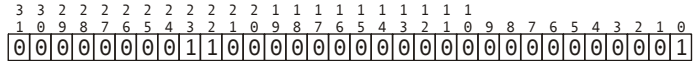
που σημαίνει: μέγιστη μαντίσα (≈ 1) και μέγιστος εκθέτης (+127). Η τιμή είναι: $2^{127} \approx 0.17014e{+}39$.

Το 1 παριστάνεται ως:



δηλαδή: $0.1_2 \times 2^1$.

Ο πλησιέστερος μεγαλύτερος αριθμός είναι αυτός που προκύπτει αν αλλάξουμε σε 1 το ψηφίο της θέσης 0.



Η διαφορά του από το 1 είναι $2^{-23} \times 2^1 \approx 0.23842e{-}06$. Αυτό είναι το ε για την παράσταση αυτή.

17.11.1 Πόλωση

Η μορφή πρόσημο - απόλυτη τιμή δεν είναι η καλύτερη δυνατή για τον εκθέτη:

- οι πράξεις είναι πολύπλοκες και
- περιορίζεται λιγάκι η περιοχή τιμών (έχουμε δυο παραστάσεις για το 0: +0, –0).

Ένα απλό τέχνασμα μας απαλλάσσει από τα παραπάνω: όταν αποθηκεύουμε μια τιμή, ο εκθέτης δεν αποθηκεύεται όπως είναι, αλλά αφού πρώτα του προσθέσουμε 127. Μέ τον τρόπο αυτόν ο εκθέτης είναι πάντοτε μη αρνητικός. Λέμε ότι ο εκθέτης είναι **πολωμένος** (biased) κατά 127.

Πρόσεξε ότι όταν κάνουμε πράξεις μεταξύ τιμών τύπου `float` δεν χρειάζεται να κάνουμε κάτι ιδιαίτερο λόγω της πόλωσης του εκθέτη. Τώρα οι πράξεις είναι πιο απλές (δεν υπάρχει πρόσημο) και οι τιμές του εκθέτη μπορούν να είναι από 0 μέχρι 255 ή πραγματικές τιμές –127 μέχρι 128. Έχουμε δηλαδή 256 διαφορετικές τιμές αντί για 255 που είχαμε χωρίς πόλωση.

Όταν όμως θέλουμε να τυπώσουμε τον αριθμό, θα πρέπει να αφαιρούμε από τον εκθέτη το 127. Στην παράσταση:

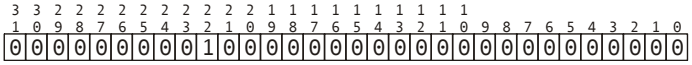
$$M\times 2^{e}$$

το νόημα του e αλλάζει:

$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

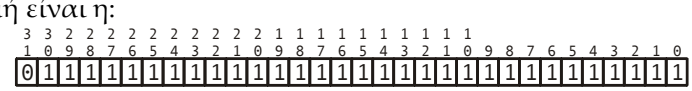
Όταν έχουμε πόλωση,

- Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί έχει τη μορφή:



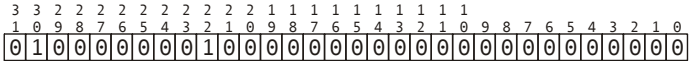
  ελάχιστος εκθέτης: 0 –στην πραγματικότητα 0 – 127 = –127– και φυσικά ελάχιστη μημηδενική μαντίσα ($0.1_2 = 0.5_{10}$). Προφανώς πρόκειται για την τιμή που υπολογίσαμε και πριν: $2^{-128} \approx 0.29387e{-}38$.

- Η μέγιστη τιμή είναι η:



  μέγιστος εκθέτης: 255 –με πραγματική τιμή 255 – 127 = 128– και μέγιστη μαντίσα (≈ 1). Η τιμή είναι: $2^{128} \approx 0.34028e{+}39$, δηλαδή διπλή από αυτήν που είχαμε χωρίς πόλωση.

- Το 1 παριστάνεται ως:



  δηλαδή: $0.1_2\times 2^{128-127}$.

  Είναι φανερό ότι το e δεν αλλάζει.

17.11.2 Άλλες Περιπλοκές – Πρότυπο IEEE

Μια άλλη παρατήρηση που μπορούμε να κάνουμε είναι η εξής: Αφού η τιμή του πρώτου ψηφίου της μαντίσας είναι πάντοτε 1, δεν υπάρχει λόγος να αποθηκεύεται· αυτό και γίνεται σε ορισμένες παραστάσεις. Βεβαίως, όπως είπαμε, είναι πάντοτε 1, εκτός από μια περίπτωση: όταν αποθηκεύουμε το 0. Θα πρέπει λοιπόν να κάνουμε ορισμένες συμβάσεις. Θα δούμε μερικές τέτοιες συμβάσεις που γίνονται στο πρότυπο της IEEE για αριθμητική κινητής υποδιαστολής, μια και το πρότυπο αυτό το συναντούμε σε πολλούς σύγχρονους υπολογιστές.

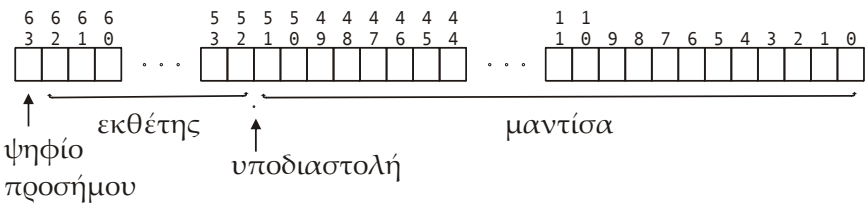
Χρησιμοποιούμε και πάλι 32 δυαδικά ψηφία (31 - 0), στο ψηφίο 31 έχουμε το πρόσημο και στα 30 - 23 έχουμε τον εκθέτη, πολωμένο κατά 127.

Για τη μαντίσα όμως τα πράγματα αλλάζουν: ακριβώς πριν από το ψηφίο 22 νοείται ότι υπάρχει το "`.1`". Έτσι, το νόημα της μαντίσας είναι:

$$M = \begin{cases} 0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu s_{31}=0 \\ -0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu s_{31}=1 \end{cases}$$

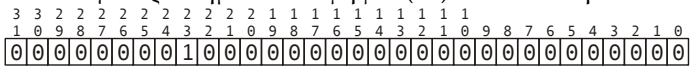
$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

Το μηδέν παριστάνεται με 32 δυαδικά ψηφία ίσα με 0. Αυτή είναι μια *σύμβαση*, αφού σύμφωνα με όσα είπαμε ο αριθμός αυτός θα έπρεπε να είναι ίσος με: $1\times 2^{-127} = 5.877\times 10^{-39}$. Γενικώς, υπάρχει μια σύμβαση (IEEE) για τις παραστάσεις που έχουν όλα τα ψηφία του εκθέτη 0. Πρόκειται για **μη-κανονικοποιημένες** (denormalized ή subnormal) παραστάσεις (πολύ μικρών) αριθμών.



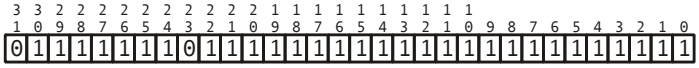
Σχ. 17-6 Τύπος **double**.

- Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει 1 το λιγότερο σημαντικό ψηφίο (23) του εκθέτη και όλα τα άλλα 0:



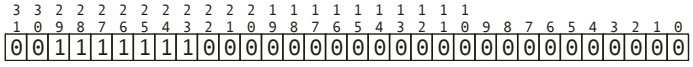
  δηλαδή: $1\times2^{1-127} = 2^{-126} \approx 0.11755e\text{-}37$.

- Η παράσταση του μέγιστου θετικού είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (23), ίσα με 1 και όλα τα ψηφία της μαντίσας 1:

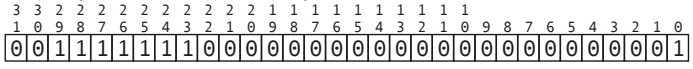


  Η τιμή αυτή είναι περίπου: $2\times2^{254-127} = 2^{128} \approx 0.34028e\text{+}39$.

- Το 1 παριστάνεται ως:



  ενώ ο πλησιέστερος μεγαλύτερος αριθμός είναι ο:



  Το ε είναι $2^{-23} \approx 1.1921e\text{-}07$. Μπορούμε δηλαδή να παραστήσουμε 24 σημαντικά δυαδικά ψηφία, που ισοδυναμούν με 7 δεκαδικά ψηφία περίπου.

- Το **άπειρο** (**INF**) παριστάνεται ως εξής: όλα τα ψηφία του εκθέτη 1 και όλα τα ψηφία της μαντίσας 0.

- Αν όλα τα ψηφία του εκθέτη είναι 1 και ένα τουλάχιστον ψηφίο της μαντίσας δεν είναι 0, *η παράσταση θεωρείται ότι δεν παριστάνει αριθμό* (**N**ot **a** **N**umber, **NaN**).

17.11.3 Οι Τύποι "double" και "long double"

Ο τύπος **double** της C++ είναι «διπλός **float**» (§17.10). Θα δώσουμε τώρα μια παράσταση σύμφωνη με το πρότυπο της IEEE.

Όπως προαναφέραμε, ο χώρος που καταλαμβάνεται από μια θέση τύπου **double** καταλαμβάνει στη μνήμη διπλό χώρο απ' ότι μια θέση τύπου **float**. Αν λοιπόν αποθηκεύουμε μια τιμή τύπου **float** σε 32 δυαδικά ψηφία, μια τιμή τύπου **double** θα αποθηκεύεται σε 64 δυαδικά ψηφία, που τα αριθμούμε από 0 μέχρι 63 (Σχ. 17-6).

Ο εκθέτης αποθηκεύεται σε 11 δυαδικά ψηφία (62 - 52) πολωμένος κατά 1023, ενώ η μαντίσα αποθηκεύεται σε 52 ψηφία (51 - 0).

Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει το λιγότερο σημαντικό ψηφίο (52) του εκθέτη ίσο με 1 και όλα τα άλλα 0: δηλαδή: $1\times2^{1-1023} = 2^{-1022} \approx 0.22251e\text{-}307$. Με μη κανονικοποιημένες μορφές μπορεί να παρασταθούν και μικρότεροι αριθμοί, μέχρι $5.0e\text{-}324$.

Η παράσταση του μέγιστου ακέραιου είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (52), ίσα με 1 και όλα τα ψηφία της μαντίσας 1: Η τιμή αυτή είναι περίπου: $2\times2^{2046-1023} = 2^{1024} \approx 0.17977e\text{+}309$

Το ε είναι $2^{-52} \approx 2.220446049250e\text{-}16$. Μπορούμε δηλαδή να παραστήσουμε 53 σημαντικά ψηφία στο δυαδικό σύστημα ή περίπου 16 σημαντικά ψηφία στο δεκαδικό.

Η C++ δίνει και τον τύπο **long double**, οι τιμές του οποίου αποθηκεύονται σε 10 ψηφιολέξεις, δηλ. 80 δυαδικά ψηφία, με 15ψήφιο εκθέτη και 64ψήφια μαντίσα. Δίνει ακρίβεια 20 (δεκαδικών) ψηφίων με τιμές από 1.9*e*-4951 μέχρι 1.1*e*4932.

Οι παραπάνω τύποι, **float**, **double**, **long double** δεν είναι αποκλειστικότητα της C++· θα τους βρεις και σε άλλες γλώσσες προγραμματισμού.

17.11.4 Μερικά Χρήσιμα Εργαλεία από τη C

Η C μας δίνει μερικές (μακρο)συναρτήσεις για να καταλαβαίνουμε τι συμβαίνει με παραστασεις τιμών κινητής υποδιαστολής. Οι ορισμοί τους υπάρχουν στο **cmath**.

int fpclassify(*τύπος κινητής υποδιαστολής* **x);**

όπου:

τύπος κινητής υποδιαστολής **= "float" | "double" | "long double" ;**

Η *fpclassify()* επιστρέφει ως τιμή μια από τις *αμοιβαίως αποκλειόμενες* τιμές **FP_INFINITE**, **FP_NAN**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO** που ορίζονται (με "**#define**") επίσης στο **cmath**.

Μας δίνει ακόμη τα εξής κατηγορήματα:

int isinf(*τύπος κινητής υποδιαστολής* **x);**
int isnan(*τύπος κινητής υποδιαστολής* **x);**
int isnormal(*τύπος κινητής υποδιαστολής* **x);**
int isfinite(*τύπος κινητής υποδιαστολής* **x);**

Σημείωση: ▶

Για να τις χρησιμοποιήσεις θα πρέπει να τις σκέφτεσαι ως

bool isinf(*τύπος κινητής υποδιαστολής* **x);**
bool isnan(*τύπος κινητής υποδιαστολής* **x);**
bool isnormal(*τύπος κινητής υποδιαστολής* **x);**
bool isfinite(*τύπος κινητής υποδιαστολής* **x);**

Έτσι, μπορείς να γράφεις "**if (isnan(sqrt(x)) …**"◀

Το *isnormal()* επιστρέφει "**1**" ("**true**") αν η *x* έχει κανονικοποιημένη παράσταση (ούτε "**0**", ούτε μη-κανονικοποιημένη, ούτε άπειρο, ούτε NaN).

Το *isfinite()* επιστρέφει "**1**" ("**true**") αν η *x* δεν είναι άπειρο ούτε NaN.

Για την (πεπερασμένη!) παράσταση του απείρου ορίζονται οι σταθερές **HUGE_VALF**, **HUGE_VAL**, **HUGE_VALL** για τους **float**, **double**, **long double** αντιστοίχως.[11]

Το παρακάτω προγραμματάκι χρησιμοποιεί τα παραπάνω εργαλεία:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   float x( 0 );
   double dn( 1e-100 ), dx( 1e100 );

   if ( fpclassify(x) == FP_ZERO )
      cout << "x is ZERO" << endl;
   cout << dn << "  " << isinf(dn) << "  " << isnan(dn)
        << "  " << isnormal(dn) << "  " << isfinite(dn) << endl;
   cout << dx << "  " << isinf(dx) << "  " << isnan(dx)
        << "  " << isnormal(dx) << "  " << isfinite(dx) << endl;
   x = dn;
   cout << x << "  " << isinf(x) << "  " << isnan(x)
        << "  " << isnormal(x) << "  " << isfinite(x) << endl;
```

[11] Στο πρότυπο C99 υπάρχει πρόβλεψη και για άλλο (πιο «πραγματικό») άπειρο, το **INFINITY**, για υλοποιήσεις που μπορούν να το υποστηρίξουν.

```
   x = dx;
   cout << x << "  " << isinf(x) << "  " << isnan(x)
        << "  " << isnormal(x) << "  " << isfinite(x) << endl;
   if ( x == HUGE_VALF )
      cout << "It IS HUGE!!!" << endl;
   cout << 1/INFINITY << endl;
}
```

και μας δίνει:

```
x is ZERO
1e-100  0  0  1  1
1e+100  0  0  1  1
0  0  0  0  1
1.#INF  1  0  0  0
It IS HUGE!!!
0
```

17.12 Σφάλμα από Μετατροπή Τύπου

Ξεκινάμε με ένα μικρό προγραμματάκι:

```
int main()
{
   long   i1, i2;
   float  r1, r2;

   i1 = 123456787;
   i2 = 123456786;
   r1 = i1;  r2 = i2;
   if ( r1 == r2)  cout << "E, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!" << endl;
}
```

Μεταγλωττίζουμε το πρόγραμμα αυτό με τη Borland C++ όπου οι τιμές τύπων `long` και `float` παριστάνονται με 32 δυαδικά ψηφία, όπως είδαμε παραπάνω. Ακόμα, χειρίζεται την παράσταση κινητής υποδιαστολής σύμφωνα με το πρότυπο της IEEE. Το αποτέλεσμα; Νάτο:

```
E, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!
```

Ας προσπαθήσουμε να βγάλουμε άκρη: Οι μεταβλητές *i*1 και *i*2 παίρνουν τις τιμές τους χωρίς κανένα πρόβλημα, μια και βρίσκονται μέσα στα όρια των τιμών που μπορούν να πάρουν (§4.3). Στην εντολή "`r1 = i1`" γίνονται τα εξής: πριν αποθηκευτεί η τιμή του *i*1 στην θέση της μνήμης *r*1 μετατρέπεται σε παράσταση κινητής υποδιαστολής. Αλλά, όπως είδαμε στην §4.3, μπορούμε να κρατήσουμε το πολύ 7 σημαντικά ψηφία. Έτσι, η «λεπτομέρεια» των δυο τελευταίων ψηφίων χάνεται. Το ίδιο συμβαίνει και με την εκτέλεση της "`r2 = i2`". Στην επόμενη εντολή –"`if (r1 == r2) cout <<...`"– όταν γίνεται η σύγκριση των *r*1 και *r*2, αυτές βρίσκονται ίσες αφού η σύγκριση γίνεται με τα πρώτα έξη ψηφία!

Φτάνουμε λοιπόν στο συμπέρασμα: *Κατά τη μετατροπή τύπου από ακέραιο τύπο σε τύπο κινητής υποδιαστολής μπορεί να έχουμε απώλεια σημαντικών ψηφίων.*

17.13 Τα Σφάλματα και πώς Μεταδίδονται

Αφού είδαμε τι συμβαίνει με την παράσταση στον τύπο `float` στο δεκαδικό σύστημα και στο δυαδικό, θα δούμε τώρα την παράσταση πιο γενικά· θα βγάλουμε ένα άνω φράγμα για το σχετικό σφάλμα από την παράσταση.

Στη συνέχεια θα δούμε πως μεταδίδονται τα σφάλματα από τους αριθμούς στις πράξεις μεταξύ τους.

17.13.1 Το Σφάλμα Παράστασης

Ας υποθέσουμε ότι έχουμε έναν υπολογιστή που δουλεύει σε σύστημα με βάση b, μπορεί να παραστήσει N ψηφία στο σύστημα αυτό. Το b είναι συνήθως 2, 8, 10, 16. Ας υποθέσουμε ότι η παράσταση ενός αριθμού x γίνεται στη μορφή:

$$x_f = M \times b^e$$

όπου:

$$M = \pm(\psi_{-1}b^{-1} + \psi_{-2}b^{-2} \ldots \psi_{-N}b^{-N})$$
$$0 \le \psi_k \le b - 1, \quad k = 1 \,..\, N$$
$$E_E \le e \le E_M$$

Όταν παριστάνουμε έναν αριθμό κρατάμε τα N πιο σημαντικά ψηφία του. Μπορούμε να πάρουμε μια τέτοια προσέγγιση είτε με **στρογγύλευση** (rounding) είτε με **αποκοπή** (truncation, choping). Να δούμε τι σφάλμα κάνουμε στις δυο περιπτώσεις.

Στρογγύλευση: Αν το ψηφίο (N-1) τάξης είναι $\ge b/2$ τότε το ψ_N αυξάνεται κατά μια μονάδα. Επομένως το (απόλυτο) σφάλμα που κάνουμε, $|x_f - x|$, είναι το πολύ:

$$|x_f - x| \le (b/2) \times b^{-(N+1)} \times b^e = 0.5 \times b^{e-N}$$

Από τη σχέση αυτή μπορούμε να υπολογίσουμε το μέγιστο **σχετικό σφάλμα** (relative error) λόγω της στρογγύλευσης:

$$\frac{|x_f - x|}{|x|} \le 0.5\frac{b^{e-N}}{|x|} = 0.5\frac{b^{e-N}}{(0.\psi_1\psi_2\ldots)b^e} = 0.5\frac{b^{-N}}{0.\psi_1\psi_2\ldots}$$

Αλλά, $0.\psi_1\psi_2\ldots_b \ge 0.100\ldots_b$ ($= b^{-1}$) και έτσι:

$$\Sigma\chi\varepsilon\tau(x_f) = \frac{|x_f - x|}{|x|} \le 0.5b^{-N+1}$$

Ας εφαρμόσουμε αυτή τη σχέση στις παραστάσεις που είδαμε στις προηγούμενες παραγράφους. Πρώτα στον δεκαδικό υπολογιστή: εδώ έχουμε $b = 10$, $N = 5$, άρα, για κάθε x:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.5\times10^{-4} \quad \text{ή} \quad \Sigma\chi\varepsilon\tau(x_f) \le 0.00005$$

Στη δυαδική παράσταση της §14.7: $b = 2$, $N = 23$, άρα:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.5\times2^{-23}$$

Αλλά, $0.5\times2^{-23} = 2^{-24} \approx 10^{-7.22} \le 0.6\times10^{-7}$ και μπορούμε να γράψουμε:

$$\Sigma\chi\varepsilon\tau(x_f) \le 0.00000006$$

Αποκοπή: Στην περίπτωση αυτή αποκόπτονται όλα τα ψηφία μετά το N-οστό. Όλο το τμήμα που αποκόπτεται είναι μικρότερο από μια μονάδα της τάξης του N-οστού ψηφίου. Δηλαδή:

$$|x_f - x| < 1\times b^{e-N}$$

Έτσι, το φράγμα του σχετικού σφάλματος είναι διπλάσιο αυτού που είχαμε στην περίπτωση της στρογγύλευσης:

$$\Sigma\chi\varepsilon\tau(x_f) = \frac{|x_f - x|}{|x|} < b^{-N+1}$$

17.13.2 Μετάδοση Σφαλμάτων

Έχουμε δυο τιμές x, y και τις παριστάνουμε στον υπολογιστή μας ως x_f, y_f σε παράσταση κινητής υποδιαστολής. Θα έχουμε:

$$x = x_f + \delta x \qquad y = y_f + \delta y$$

Ας συμβολίσουμε με π μια από τις πράξεις +, −, ×, / και με π_f την αντίστοιχη πράξη του υπολογιστή μας, μια από τις: $+_f$, $-_f$, $*_f$, $/_f$. Για το σφάλμα του αποτελέσματος θα έχουμε:

$$(x \pi y) - (x_f \pi_f y_f) = (x \pi y) - (x_f \pi_f y_f) + (x_f \pi y_f) - (x_f \pi y_f)$$
$$= (x_f \pi y_f - x_f \pi_f y_f) + (x \pi y - x_f \pi y_f)$$

Ο πρώτος όρος αναφέρεται μόνο σε τιμές που παριστάνονται ήδη στον υπολογιστή. Μας δίνει το σφάλμα που γίνεται από την *πράξη* π_f. Αν γυρίσεις στην §17.8 μπορείς να δεις πώς κάναμε στον δεκαδικό υπολογιστή την πρόσθεση των 14.563 (= x_f) και 0.16773 (= y_f). Ο πρώτος όρος για την περίπτωση αυτή είναι:

$$(x_f + y_f) - (x_f +_f y_f) = (14.563 + 0.16773) - (14.563 +_f 0.16773)$$
$$= 14.73073 - 14.731$$
$$= -0.0008$$

Αν υποθέσουμε ότι, όπως λέγαμε στην §14.8, η Αριθμητική Μονάδα κάνει τις πράξεις με μεγαλύτερη ακρίβεια και το αποτέλεσμα στρογγυλεύεται (ή αποκόπτεται) στη συνέχεια, μπορούμε να πούμε ότι:

$$(x_f \pi_f y_f) = (x_f \pi y_f)_f$$

Οπότε, σύμφωνα με αυτά που είπαμε για το σφάλμα στρογγύλευσης, στην προηγούμενη παράγραφο, έχουμε:

$$|x_f \pi y_f - x_f \pi_f y_f| \le 0.5 |x_f \pi y_f| b^{1-N}$$

Ας έρθουμε τώρα στο δεύτερο όρο:

$$(x \pi y - x_f \pi y_f)$$

Εδώ *οι πράξεις είναι ακριβείς*, αλλά συγκρίνουμε το αποτέλεσμα που παίρνουμε από τις αληθείς τιμές με αυτό που παίρνουμε από τις (προσεγγιστικές) τιμές σε μορφή κινητής υποδιαστολής. Αυτό λέγεται **μεταδιδόμενο σφάλμα** (propagated error). Θα δούμε στη συνέχεια το μεταδιδόμενο σφάλμα για τις τέσσερις πράξεις.

Πολλαπλασιασμός:

$$xy - x_f y_f = xy - (x - \delta x)(y - \delta y)$$
$$= x\delta y + y\delta x - \delta x \delta y$$

και

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f) + \Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f)$$

Αν $\Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)$$

Διαίρεση:

$$\Sigma\chi\varepsilon\tau\left(\frac{x_f}{y_f}\right) = \frac{|x/y - x_R/y_R|}{|x/y|} = \frac{|y_R/y - x_R/x|}{|y_R/y|}$$
$$= \frac{|\delta x/x - \delta y/y|}{|1 - \delta y/y|} \le \frac{\Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)}{|1 - \Sigma\chi\varepsilon\tau(y_f)|}$$

Και εδώ, αν $\Sigma\chi\varepsilon\tau(x_f)\Sigma\chi\varepsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\varepsilon\tau(x_f y_f) \le \Sigma\chi\varepsilon\tau(x_f) + \Sigma\chi\varepsilon\tau(y_f)$$

Όπως βλέπεις, στον πολλαπλασιασμό και τη διαίρεση τα σφάλματα μεταδίδονται αργά. Είναι, θα μπορούσαμε να πούμε, αρκετά «σίγουρες» πράξεις. Δεν συμβαίνει όμως το ίδιο με τις άλλες δυο πράξεις που έχουμε αφήσει τελευταίες.

Πρόσθεση - Αφαίρεση: Αν χρησιμοποιήσουμε το π για το + ή το -, το απόλυτο σφάλμα (που αντιστοιχεί στο δεύτερο όρο) είναι:

$$x \pi y - x_f \pi y_f = x \pi y - (x - \delta x) \pi (y - \delta y)$$
$$= \delta x \pi \delta y$$
$$\le |\delta x| + |\delta y|$$

Ας δούμε τι σημαίνει αυτό, με ένα παράδειγμα: έχουμε τους αριθμούς 0.123445 και 0.123454 και χρησιμοποιούμε το δεκαδικό μας υπολογιστή για να υπολογίσουμε τη διαφορά

τους. Ο πρώτος παριστάνεται ως: 0.12345 (σφάλμα -0.000005) και ο δεύτερος: 0.12345 (σφάλμα 0.000004) και η διαφορά τους υπολογίζεται: 0. Το ακριβές αποτέλεσμα είναι 0.000009 και το σχετικό σφάλμα είναι 100%! Για να δούμε τι έγινε εδώ: Έχουμε δυο αριθμούς με έξη ψηφία και παραπλήσιες απόλυτες τιμές –η διαφορά τους βρίσκεται στα δυο τελευταία ψηφία. Για να παρασταθούν οι αριθμοί στον υπολογιστή, που δέχεται μόνο 5 ψηφία, γίνεται το καλύτερο δυνατό: στρογγύλευση σε 5 ψηφία. Τώρα, τα τέσσερα πρώτα –και σωστά– ψηφία είναι ίδια ενώ τα 5α ψηφία είναι λάθος, λόγω της στρογγύλευσης. Το αποτέλεσμα υπολογίστηκε ακριβώς με βάση τα 5α ψηφία.

Μήπως αυτό ήταν ένα «παρατραβηγμένο» παράδειγμα; Πόσο συχνά συμβαίνει κάτι τέτοιο; Σχεδόν κάθε φορά που έχουμε να αφαιρέσουμε δυο αριθμούς με παραπλήσιες απόλυτες τιμές. Οι αριθμοί αυτοί συνήθως προκύπτουν από (μη ακριβείς) πράξεις και έχουν κάποιο σφάλμα. Το σφάλμα αυτό βρίσκεται στα τελευταία (λιγότερο σημαντικά) ψηφία. Έτσι, με μια αφαίρεση, παίρνουμε αποτέλεσμα που μπορεί να είναι τελείως λάθος, όπως στο παράδειγμά μας.

17.14 Ισότητα στους Τύπους Κινητής Υποδιαστολής

Τα παραπάνω γεννούν και ένα άλλο ερώτημα: αν έχουμε δηλώσει:

```
float A, B;
```

πόσο νόημα έχει η σύγκρισή τους για ισότητα:

```
if ( A == B ) ...
```

Ας πούμε α, β τις ακριβείς τιμές των ποσοτήτων που φιλοδοξούμε να υπολογίσουμε στο πρόγραμμά μας με τα A, B αντίστοιχα. Οι πράξεις που θα κάνουμε στον υπολογιστή μας θα εισαγάγουν σφάλματα και τελικώς:

$$|A - \alpha| \le \delta A \quad \text{και} \quad |B - \beta| \le \delta B$$

ή

$$\alpha - \delta A \le A \le \alpha + \delta A \quad \text{και} \quad \beta - \delta B \le B \le \beta + \delta B$$

Από αυτές παίρνουμε:

$$(\alpha - \beta) - (\delta A + \delta B) \le A - B \le (\alpha - \beta) + (\delta A + \delta B)$$

Αυτό που μας ενδιαφέρει βεβαίως, είναι να συγκρίνουμε για ισότητα τα α και β. Αν $\alpha = \beta$ τότε για τα A και B έχουμε:

$$-(\delta A + \delta B) \le A - B \le (\delta A + \delta B)$$

Όπως φαίνεται λοιπόν, το σωστό είναι να ρωτήσουμε:

```
if ( fabs(A - B) <= eps ) ...
```

Το *eps*, κατ' αρχήν, εξαρτάται από

- το συγκεκριμένο πρόβλημα, αφού εξαρτάται από τα σφάλματα παράστασης και πράξεων που συσσωρεύτηκαν κατά τους υπολογισμούς των A και B.
- την ακρίβεια που απαιτούμε στον υπολογισμό του A ή του B.

Αλλά, το *eps* έχει περιορισμούς από τον τύπο `float` που δουλεύουμε: *δεν μπορεί να είναι μικρότερο* από το

$$\delta A = min\{\, u: \mathbb{R} \mid u > 0 \wedge (|A|+u)_R \neq |A|_R \bullet u \,\}$$

Αποδεικνύεται ότι για το δA έχουμε:

$$|A|\varepsilon/b < \delta A \le |A|\varepsilon$$

όπου b η βάση του αριθμητικού συστήματος του υπολογιστή σου και ε το *έψιλον* του τύπου `float`. Αν έχεις $b = 2$, τότε $|A|\varepsilon/2 < \delta A \le |A|\varepsilon$.

Από τον ορισμό έχεις ότι: αν $0 \le x < \delta A$ τότε αν $B = A + x$ θα πρέπει να δεχτείς ότι A `==` B αφού ο υπολογιστής σου δεν μπορεί να σου δώσει κάτι ακριβέστερο· όλοι οι B, με τα παραπάνω χαρακτηριστικά, παριστάνονται με τον ίδιο τρόπο με τον A. Δηλαδή, αν ήδη δεν μπορείς να ξεχωρίσεις δυο αριθμούς που διαφέρουν λιγότερο από δA, λόγω του τρόπου

που παριστάνονται στον τύπο **float**, δεν έχει νόημα να προσπαθείς να ανιχνεύσεις διαφορά παράστασης μικρότερη από δ*A*. Το *eps* λοιπόν δεν έχει νόημα αν είναι μικρότερο από δ*A* και, επειδή αυτό που έχουμε εύκολα είναι το πάνω φράγμα του, θα πρέπει $eps \geq |A|\varepsilon$.

«Δεν έχει νόημα» ή θα έχουμε πρόβλημα; Μπορεί να έχεις και πρόβλημα! Για παράδειγμα, στις επαναπροσεγγιστικές (iterative) μεθόδους συχνά δουλεύουμε ως εξής: βρίσκουμε ένα διάστημα [*a*, *b*] μέσα στο οποίο βρίσκεται η τιμή ξ που θέλουμε να προσεγγίσουμε και μικραίνουμε το διάστημα έτσι ώστε να γίνει τελικά μηδενικού μήκους, οπότε θα έχουμε *a*=ξ=*b*. Σύμφωνα μεαυτά που είπαμε, θα κάνουμε έλεγχο ως εξής:

```
while ( fabs(b-a) > eps )...
```

αλλά, θα έλεγε κανείς, εδώ πέρα το *eps* ας είναι ό,τι θέλει, ακόμη και μηδέν. Το πολύ-πολύ να γίνουν τα *a* και *b* ίσα, πράγμα όχι άσχημο. Λοιπόν: *δεν είναι έτσι*. Μπορεί (αυτό εξαρτάται από τη μέθοδο) το *b* να γίνει *a*+δ*A*. Από κει και πέρα οι τιμές των *a* και *b* (επιλεγόμενες από το [*a*,*b*]) μπορεί να μην αλλάζουν, ενώ *fabs*(*b*-*a*) = δ*A* > *eps*. Μια τέτοια μέθοδος είναι αυτή της διχοτόμησης και το πρόβλημα φαίνεται στο παρακάτω

Παράδειγμα

Από την §14.3 αντιγράφουμε, με μικρές αλλαγές, το παρακάτω πρόγραμμα, που δοκιμάζει μια συνάρτηση για επίλυση αλγεβρικών εξισώσεων με τη μέθοδο της διχοτόμησης.

```
#include <iostream>
#include <cmath>

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main
```

Δεν έχουμε αλλάξει τίποτε στη *bisection*· αλλά, ενώ στήν §14.3 την είχαμε καλέσει με *epsilon* = 1*e*-5 –και μας έδωσε προσέγγιση της ρίζας 0.1585983– τώρα την καλούμε με *epsilon* = 1e-20. Ο τύπος **double**, στην C++ που δουλεύουμε, είναι αυτός που είδαμε στην §14.7.3, με $\varepsilon = 2^{-23} \approx 2.220446049250e\text{-}16$.

Το πρόγραμμα, κατά την εκτέλεση, πέφτει σε αέναη ανακύκλωση. Ζητώντας εκτύπωση ενδιαμέσων στοιχείων βλέπουμε τα εξής:

```
#Επαν       a           b              |b-a|/2              m
   :        :           :                 :                :
   54  0.15859434  0.15859434  5.5511151231257827e-17  0.15859434
   55  0.15859434  0.15859434  2.7755575615628914e-17  0.15859434
   56  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   57  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   58  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
   59  0.15859434  0.15859434  1.3877787807814457e-17  0.15859434
    :        :           :                 :                :
```

Δηλαδή, το $m = (a+b)/2$ δεν μπορεί να γίνει καλύτερο και το *fabs*(*b*-*a*)/2 παραμένει μεγαλύτερο από το *epsilon*. Το πρόβλημα ξεκινάει από την τιμή του *epsilon*, που είναι πολύ μικρή για τον τύπο κινητής υποδιαστολής (**double**) που δουλεύουμε. Πράγματι,

$$|a|\varepsilon = 0.15859434 \times 2.220446049250e\text{-}16 \approx 3.521501756864e\text{-}17$$

και $\delta A \in [1.76e\text{-}17, 3.52e\text{-}17]$. Το *epsilon* δεν θα έπρεπε να είναι μικρότερο από δA.[12]

Και τί κάνουμε τώρα; Πώς διορθώνουμε το πρόγραμμα; Μια διόρθωση θα ήταν η εξής: Να βάλουμε ένα

$$trEps = max(|m|\varepsilon/b, epsilon)$$

και να αλλάξουμε τη συνθήκη της while σε `fabs(b-a)/2 >= trEps`. Στην περίπτωση αυτήν όμως η διαδικασία μας εξαρτάται από χαρακτηριστικά του τύπου `float` και χάνει τη δυνατότητα μεταφοράς. Καταφεύγοντας στην καλύτερη μελέτη της μεθόδου, βλέπουμε ότι $m_{νέο} = m_{παλιό} \pm |b - a|/2$. Αν λοιπόν το $|b - a|/2$ γίνει πολύ μικρό –*πρακτικώς μηδέν*– ως προς το m, δεν έχει νόημα να συνεχίζουμε. Πώς ελέγχεται αυτό σε C++; Με τον εξής «περίεργο» τρόπο:

```
m == m + fabs(b-a)/2
```

Εκτός από αυτό, η διαδικασία χρειάζεται και μια άλλη βελτίωση: το πρόγραμμα που την καλεί θα πρέπει να μπορεί να καθορίσει μέγιστο πλήθος επαναλήψεων που θα εκτελεσθούν, άσχετα με το αν πετύχαμε την ακρίβεια που θέλουμε:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   double m, d;

   if ( f(a)*f(b) > 0 )
      errCode = 3;
   else
   {
      int n( 0 );
      do {
         ++n;
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                             else  a = m;
         d = fabs(b - a)/2;
      } while ( (d >= epsilon) && (m != m + d) && (n != nMax) );
      root = m;
      if (d <= epsilon)  errCode = 0;
      else if (m == (m + d))  errCode = 1;
      else if (n == nMax)  errCode = 2;
   } // if
} // bisection
```

Τώρα, που η επαναλήψεις σταματούν για τρεις διαφορετικούς λόγους, θα πρέπει να ελέγξουμε το λόγο τερματισμού και να τον γνωστοποιήσουμε στο πρόγραμμα που κάλεσε τη διαδικασία. Αυτό γίνεται με τη *errCode*:

errCode == 1 σημαίνει: Δεν μπορεί να γίνει άλλη βελτίωση.

errCode == 2 σημαίνει: Ξεπεράσαμε τις *nMax* επαναλήψεις χωρίς να επιτευχθεί η ακρίβεια που ζητήθηκε.

errCode == 3 σημαίνει: Λάθος αρχικό διάστημα.

Και να μια δοκιμή:

```
   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
```

αποτέλεσμα:

[12] Όπως καταλαβαίνεις, τα τελευταία ψηφία των *a*, *b* δεν έχουν νόημα.

```
Λάθος 1
Ρίζα =  0.158594
Λάθος 2
Ρίζα =  0.184375
Λάθος 0
Ρίζα =  0.158594
```

Με την ευκαιρία, πρόσεξε και κάτι άλλο: στον έλεγχο της *n* δεν βάλαμε *n* < *nMax* αλλά *n* != *nMax*· το αποτέλεσμα; Αν η συνάρτηση κληθεί με *nMax* ≤ 0 ο αριθμός των επαναλήψεων δεν ελέγχεται.

Με τον ίδιο τρόπο που γράψαμε το «αν η βελτίωση είναι 0 ως προς το *m*» θα πρέπει να χειρίζεσαι πολλές φορές την περίπτωση: «αν το *A* είναι 0 τότε...». Αντί να γράφεις:

```
if ( fabs(A) <= eps ) ...
```

συχνά είναι προτιμότερο να γράφεις

```
if ( x + A = x ) ...
```

δηλαδή, αν το *A* είναι 0 ως προς το *x*. Φυσικά, το *x* εξαρτάται από το πρόβλημα που έχεις να λύσεις και πρέπει να επιλεγεί προσεκτικά.

17.15 Πρακτικές Συμβουλές

Οι πρώτες δυο συμβουλές για κάποιον που γράφει προγράμματα για επεξεργασία αριθμητικών στοιχείων είναι μάλλον τετριμμένες, αλλά πρέπει να τις πούμε:

- ***Μη χρησιμοποιείς χωρίς λόγο τους τύπους κινητής υποδιαστολής. Προτίμησε ακέραιους τύπους όπου μπορείς. Πρόσεχε, όμως τους πολύ μεγάλους ακέραιους!***
- ***Βελτίωσε τον αλγόριθμό σου ώστε να ελαττώσεις τις πράξεις που κάνεις στο ελάχιστο δυνατό!***

Καλά, θα πεί ο πεπειραμένος προγραμματιστής, για νέο μας το λες; Αυτό ισχύει σε κάθε περίπτωση. Σωστά! Αλλά, αν είναι στόχος να ελαττώνουμε τον χρόνο επεξεργασίας στο ελάχιστο για κάθε πρόγραμμα, για τα αριθμητικά προγράμματα έχουμε ένα λόγο ακόμη: *λιγότερες πράξεις σημαίνει και πιο ακριβές αποτέλεσμα*.

- ***Πρόσεξε τις προσθέσεις και τις αφαιρέσεις! Αν νομίζεις ότι μπορεί να σου δημιουργήσουν προβλήματα προσπάθησε να τις αποφύγεις!***

Να τις αποφύγω; Αν πρέπει να τις κάνω, πως να τις αποφύγω; Δυο παραδείγματα σου δίνουν μια ιδέα. Και τα δυο προγράμματα εκτελέστηκαν σε τύπο `float` 32 ψηφίων.

Παράδειγμα 1

Θέλουμε να λύσουμε την εξίσωση: $ax^2 + bx + c = 0$ όπου $a = 1.0$, $b = 1000000000.0 = 10^9$, $c = 4.0$ και φυσικά χρησιμοποιούμε τους γνωστούς μας τύπους:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ο τύπος `float` δουλεύει όπως περιγράψαμε στην §14.6.2. Όταν υπολογίζουμε το υπόριζο $b^2 = 10^{18}$, ενώ $4ac = 16$. Η αφαίρεση στον υπολογιστή μας θα δώσει αποτέλεσμα 10^{18} και τελικά θα έχουμε $x_+ =$ -1000000000.0 και $x_- = 0$. Η πρώτη ρίζα προσεγγίστηκε θαυμάσια. Στη δεύτερη, η προσέγγιση $\sqrt{b^2 - 4ac} \approx b$ και η πράξη $-b + b$ (υποχείλιση) που ακολούθησε, μας έδωσε ένα τελείως λάθεμένο αποτέλεσμα. Τον υπολογισμό του υπορίζου δεν μπορούμε να τον βελτιώσουμε· ας τον χρησιμοποιήσουμε λοιπόν μόνο για την πρώτη ρίζα και ας αποφύγουμε την αφαίρεση. Θα στηριχτούμε στην ιδιότητα: $x_+x_- = c/a$, από όπου παίρνουμε: $x_- = c/(ax_+)$.

Το παρακάτω πρόγραμμα υπολογίζει το x_+(x1) και στη συνέχεια το x_- (x2) με τον γνωστό τύπο. Στη συνέχεια υπολογίζει τη δεύτερη ρίζα ως x3 χωρίς να κάνει την αφαίρεση.

```
int main()
{
   float a, b, c, x1, x2, x3;

   a = 1.0f;  b = 1000000000.0f;  c = 4.0f;
   x1 = (-b - sqrt(b*b - 4*a*c))/(2*a);
   x2 = (-b + sqrt(b*b - 4*a*c))/(2*a);
   x3 = c/(a*x1);
   cout << x1 << "  " << x2 << "  " << x3 << endl;
}
```

Να το αποτέλεσμα:

```
-1e+009  0  -4e-009
```

Παρατηρήσεις ▶

Για πρώτη φορά αναφέραμε ότι «*η αφαίρεση είναι μια «κακή» πράξη για τους τύπους κινητής υποδιαστολής διότι μπορεί να προκαλέσει απώλεια σημαντικών ψηφίων*» στην παρατήρηση 3 του Παραδ. 2 της §5.5. Μπορείς να κάνεις αυτά που λέμε παραπάνω με τα προγράμματα που είδαμε εκεί αφού αλλάξεις τον `double` σε `float`.◀

Παράδειγμα 2

Θέλουμε να υπολογίσουμε την τιμή της παράστασης 1 - *συνt* για πολύ μικρά *t*. Αν δεν προσέξουμε θα πάρουμε 0 (υποχείλιση). Ας αποφύγουμε την αφαίρεση:

$$1-\sigma\upsilon\nu t = (1-\sigma\upsilon\nu t)\frac{1+\sigma\upsilon\nu t}{1+\sigma\upsilon\nu t} = \frac{\eta\mu^2 t}{1+\sigma\upsilon\nu t}$$

Και δοκιμάζουμε τους δυο τρόπους:

```
   t = 0.0000000001;
   x1 = 1 - cos(t);
   cout << x1 << endl;
   x2 = sin(t)*sin(t)/(1+cos(t));
   cout << x2 << endl;
```

αποτέλεσμα:

```
0
5e-021
```

Δηλαδή, με την ασφαλέστερη "1+*συνt*" αποφύγαμε την «καταστροφική» "1-*συνt*".

- ***Απόφυγε την υπερχείλιση χωρίς λόγο.***

Όπως είδαμε πιο πριν, για τους τύπους κινητής υποδιαστολής, δεν ισχύουν η προσεταιριστικότητα της πρόσθεσης και του πολλαπλασιασμού, όπως και η επιμεριστικότητα των δυο πράξεων. Είναι πολύ πιθανό, με μια αναδιάταξη πράξεων σε μια αριθμητική παράσταση, να αποφύγεις μια υπερχείλιση από ενδιάμεσες πράξεις.

Ας δούμε δυο παραδείγματα:

Παράδειγμα 3

Από τα πρώτα πράγματα που μαθαίνει κανείς στον προγραμματισμό είναι να μην διαιρεί δια 0. Έτσι, η εντολή:

```
z = x/y;
```

αντικαθίσταται πολλές φορές από κάτι σαν:

```
if ( y != 0 )  z = x/y;
         else  ...
```

Όπως είπαμε πιο πάνω, κάτι τέτοιο δεν έχει και πολύ νόημα. Πιο σωστό θα ήταν κάτι σαν:

```
if ( fabs(y) > eps ) ...
```

Πράγματι, αν η τιμή του *y* έχει προκύψει μετά από πολλές πράξεις στον τύπο `float`, είναι απίθανο να είναι 0 όταν την περιμένουμε. Στην περίπτωση αυτήν ένα «απρόσεκτο» πρόγραμμα μπορεί να υποστεί τις συνέπειες μιας «κρυφής υπερχείλισης». Δηλαδή, το *y* δεν είναι ακριβώς μηδέν, αλλά έχει μια πολύ μικρή τιμή χωρίς κανένα νόημα. Το αποτέλεσμα θα είναι, πολλές φορές, να μην έχουμε υπερχείλιση αλλά το *z* να πάρει μια απίθανη τιμή που αλλοιώνει όλους τους υπολογισμούς στη συνέχεια.

Παράδειγμα 4

Παρόμοιες προφυλάξεις μπορούμε να πάρουμε και στην πρόσθεση. Ας υποθέσουμε ότι οι *x* και *y* μπορούν να πάρουν μεγάλες θετικές τιμές και η εντολή:

```
z = x + y;
```

θα μπορούσε να οδηγήσει σε υπερχείλιση. Θα μπορούσαμε και εδώ να προλάβουμε μια τέτοια πρόσθεση. Όπως ξέρουμε η C++ μας δίνει για τον τύπο `float` τη σταθερά *FLT_MAX* που είναι η μέγιστη τιμή που μπορεί να παρασταθεί στον τύπο `float`[13]. Μπορούμε λοιπόν, όπου χρειάζεται, να αντικαταστήσουμε την εντολή εκχώρησης με κλήση κάποιας διαδικασίας *addFloat*, που γράφεται όπως η *addInt* που γράψαμε για τον τύπο `int`.

Φυσικά, παρόμοιες προφυλάξεις μπορούμε να πάρουμε για όλες τις πράξεις και να έχουμε το κεφάλι μας ήσυχο. Ε, όχι κι έτσι! Ένα τέτοιο πρόγραμμα θα ήταν *απαράδεκτο*. Οι περισσότεροι από τους ελέγχους, που είναι *χρονοβόροι*, θα ήταν άχρηστοι. Ο *αμυντικός προγραμματισμός* σε σχέση με τις αριθμητικές πράξεις μπορεί να χρησιμοποιείται, αλλά μόνον όπου χρειάζεται.

Η τελευταία μας συμβουλή δεν μπορεί να είναι άλλη:

- ***Διάβασε Αριθμητική Ανάλυση.***

Με όσα είπαμε στις προηγούμενες παραγράφους, δεν φιλοδοξούμε να καλύψουμε τα θέματα που παρουσιάσαμε, αλλά περισσότερο να σου δείξουμε τα προβλήματα: *Τα αριθμητικά προγράμματα είναι δύσκολα. Η σύνθεσή τους απαιτεί* ***γνώσεις*** *και πολλή προσοχή.* Τις περισσότερες από τις γνώσεις που θα σου χρειαστούν θα σου τις δώσει η Αριθμητική Ανάλυση.

Ασκήσεις

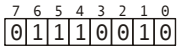
Α Ομάδα

17-1 (*γενίκευση της count*1) Γράψε περίγραμμα συνάρτησης *xCount*1 που θα τροφοδοτείται με τιμή *x*, ακέραιου τύπου και δύο φυσικούς $k_1 \le k_2$ και θα υπολογίζει και θα επιστρέφει το πλήθος των δυαδικών ψηφίων που έχουν τιμή 1 στις θέσεις από k_1 μέχρι k_2 της *x*.

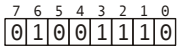
17-2 (*κυκλική ολίσθηση*) Γράψε συνάρτηση, με το όνομα *rShiftRight*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με μια τιμή *x*, τύπου `unsigned char`, και μια μη αρνητική ακέραιη τιμή *p* και θα υπολογίζει και θα επιστρέφει την *κυκλική* ολίσθηση δεξιά της *x* κατά *p* θέσεις (τα *p* δυαδικά ψηφία που χάνονται προς τα δεξιά εμφανίζονται από τα αριστερά, αντί μηδενικών, με την ίδια σειρά), όπως φαίνεται στο παρακάτω παράδειγμα. Αν η συνάρτησή μας κληθεί με το *x* που φαίνεται και *p* = 3 παίρνουμε το αποτέλεσμα που βλέπεις:

---

[13] Παρομοίως υπάρχουν και οι *DBL_MAX*, *LDBL_MAX* για τους τύπους `double` και `long double` αντίστοιχως.



x



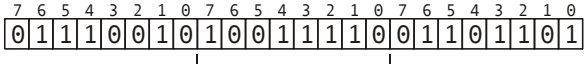
αποτέλεσμα

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *rShiftLeft*, για κυκλική ολίσθηση αριστερά.

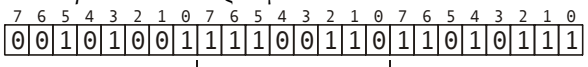
Β Ομάδα

17-3 (*ολίσθηση σε πίνακα*) Γράψε συνάρτηση, με το όνομα *shiftLeft*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου `unsigned char`, το πλήθος των στοιχείων του *n* και μια μη αρνητική ακέραιη τιμή *p* και θα κάνει ολίσθηση αριστερά κατά *p* θέσεις των δυαδικών ψηφίων *ολόκληρου του πίνακα*, όπως φαίνεται στο παρακάτω παράδειγμα. Έχουμε αρχικώς (*n* = 3):



a[2] a[1] a[0]

και ζητείται ολίσθηση κατά *p* = 4. Θα πάρουμε:



a[2] a[1] a[0]

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *shiftRight*, για ολίσθηση πίνακα δεξιά.

17-4 Γενίκευσε τις *bitValue()*, *setBit()*, *clearBit()*, *count1()* και *part()* για πίνακα ακέραιου τύπου *T*. Αν κάθε τιμή τύπου *T* αποθηκεύεται σε s_T ψηφιολέξεις τότε η παράμετρος *p* των τριών πρώτων θα μπορεί να παίρνει τιμές από 0 μέχρι $n \times s_T - 1$, όπου *n* το πλήθος των στοιχείων του πίνακα.

17-5 Ακολουθώντας το παράδειγμα που δώσαμε στην §17.3 για την πρόσθεση, γράψε διαδικασίες για ασφαλείς πράξεις: αφαίρεση (*subtrInt*), πολλαπλασιασμό (*multInt*), διαίρεση (*divInt*) στον τύπο `int`.

17-6 Γράψε διαδικασίες για ασφαλείς πράξεις για τον τύπο `float`.

Γ Ομάδα

17-7 Αν *x* τιμή τύπου `float`, ορίζουμε:

$$\varepsilon_x = min\{\, u: \mathbb{R} \mid u > 0 \wedge (|x|+u)_f \neq |x|_f \bullet u \,\}$$

Μπορείς να βρεις σχέση του ε_x με το ε; Πόσο σωστό είναι το $\delta x = x \cdot \varepsilon$;

[Απάντηση: $x \cdot \varepsilon$ /βάση $< \varepsilon_x \leq x \cdot \varepsilon$]

κεφάλαιο

18

Προετοιμάζοντας Βιβλιοθήκες

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να κάνεις (στατικές) βιβλιοθήκες συναρτήσεων

- είτε διότι το ζήτησε κάποιος πελάτης (ή εργοδότης)
- είτε για να κεφαλαιοποιήσεις την προγραμματιστική δουλειά που κάνεις, ώστε να μη χρειάζεται να ανακαλύπτεις τον τροχό κάθε τόσο.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράψεις και να χρησιμοποιήσεις στατικές βιβλιοθήκες συναρτήσεων και περιγραμμάτων.

Έννοιες κλειδιά:

- *οδηγία* `define`
- *οδηγία* `ifdef`
- *βιβλιοθήκη περιγραμμάτων συναρτήσεων*
- *στατική βιβλιοθήκη*
- *χωριστή μεταγλώττιση*
- `namespace`

Περιεχόμενα:

18.1 Οι Οδηγίες "`define`", "`ifdef`" κλπ 582
18.2 Μια Βιβλιοθήκη Περιγραμμάτων Συναρτήσεων 586
18.3 Χωριστή Μεταγλώττιση 588
18.4 Μια Στατική Βιβλιοθήκη 592
18.5 "`namespace`": Το Πρόβλημα και η Λύση 593
18.6 Ανακεφαλαίωση 596
Ασκήσεις 597
Α Ομάδα 597
Β Ομάδα 597

Εισαγωγικές Παρατηρήσεις:

Αρχίζοντας, να τονίσουμε ότι οι προγραμματιστικές βιβλιοθήκες με τις οποίες θα ασχοληθούμε είναι *βιβλιοθήκες συναρτήσεων* και όχι εκτελέσιμων προγραμμάτων (υπάρχουν και τέτοιες). Οι πρώτες εμφανίστηκαν σχετικώς νωρίς με πιο γνωστές –τα πρώτα χρόνια– τις βιβλιοθήκες της FORTRAN.

Οι συναρτήσεις των βιβλιοθηκών *συνδέονται* στα προγράμματα που αναπτύσσουν οι προγραμματιστές απαλλάσσοντάς τους έτσι από το να «ξαναεφευρίσκουν τον τροχό»!

Οι προγραμματιστές αναπτύσσουν βιβλιοθήκες συναρτήσεων «κατά παραγγελία». Αλλά πολλές βιβλιοθήκες προκύπτουν και ως παραπροϊόντα της ανάπτυξης προγραμμάτων: ο προγραμματιστής, κρίνοντας από την εμπειρία του, ότι κάποιες συναρτήσεις έχουν γενι-

κότερη χρησιμότητα και όχι μόνο για το πρόγραμμα που γράφηκαν τις εντάσσει σε βιβλιοθήκες ώστε να μπορεί να τις ξαναχρησιμοποιήσει.

Μια βιβλιοθήκη, σε σχέση με τον τρόπο που χειρίζεται τις συνιστώσες της ο **συνδέτης** (linker), μπορεί να είναι **στατική** (static) ή **δυναμικής σύνδεσης** (dynamic linking):

- *Στατικές Βιβλιοθήκες*: Μετά τη μεταγλώττιση του (αρχικού) προγράμματος ο συνδέτης εντάσσει σε αυτό τις συναρτήσεις που καλούνται από τη βιβλιοθήκη δημιουργώντας το τελικό εκτελέσιμο πρόγραμμα που είναι αυθύπαρκτο, μπορεί δηλαδή να εκτελεσθεί χωρίς την παρουσία της βιβλιοθήκης.
- *Βιβλιοθήκες Δυναμικής Σύνδεσης*: Τέτοιες είναι οι *DLL* (Dynamic Link Library) των Windows και οι *DSO* (Dynamic Shared Object) των Unix, Linux. Στη φάση της δημιουργίας του εκτελέσιμου προγράμματος ο συνδέτης δεν αντιγράφει σε αυτό τις συναρτήσεις που καλούνται· απλώς καταγράφει τι καλείται και σε ποιο σημείο. *Η τελική φάση της σύνδεσης γίνεται κάθε φορά που αρχίζει η εκτέλεση του προγράμματος*: ο συνδέτης αναζητεί τις δυναμικές βιβλιοθήκες, τις φορτώνει στη μνήμη και κάνει την τελική σύνδεση. *Αν κάποια βιβλιοθήκη είναι ήδη φορτωμένη, επειδή τη ζήτησε άλλο πρόγραμμα, δεν ξαναφορτώνεται*. Η δυναμική βιβλιοθήκη σβήνεται από τη μνήμη όταν δεν εκτελείται πρόγραμμα που να έχει συνδεθεί με αυτήν. Ένα (εκτελέσιμο) πρόγραμμα που χρησιμοποιεί βιβλιοθήκες δυναμικής σύνδεσης *δεν είναι αυθύπαρκτο*. Όταν το μεταφέρεις από τη μια εγκατάσταση στην άλλη θα πρέπει να φροντίσεις να υπάρχουν –στη νέα εγκατάσταση– και οι απαιτούμενες βιβλιοθήκες.

Πέρα από τις βιβλιοθήκες συναρτήσεων υπάρχουν βιβλιοθήκες περιγραμμάτων (συναρτήσεων ή κλάσεων), κλάσεων κλπ.

Εδώ θα ασχοληθούμε με

- στατικές βιβλιοθήκες συναρτήσεων και
- βιβλιοθήκες περιγραμμάτων συναρτήσεων.

Πριν ξεκινήσουμε όμως θα πρέπει να δούμε μερικές «οδηγίες προς τον προεπεξεργαστή» που θα μας είναι απαραίτητες από εδώ και πέρα: τις **define** και **undef** καθώς και τις **ifdef** και **ifndef**.

> *ΠΡΟΣΟΧΗ! Σε αυτό το κεφάλαιο, για να δείξουμε ορισμένα πράγματα, θα πρέπει να χρησιμοποιήσουμε διαδικασίες που εξαρτώνται από το περιβάλλον ανάπτυξης και το ΛΣ. Εδώ θα στηριχτούμε στον Dev-C++, σε περιβάλλον Windows. Εσύ θα πρέπει, πιθανότατα, να ψάξεις τα εγχειρίδια (ή το Help) της C++ που χρησιμοποιείς.*

18.1 Οι Οδηγίες "`define`", "`ifdef`" κλπ

Στην §1.5 είδαμε την οδηγία προς τον προεπεξεργαστή "`include`" και από τότε τη χρησιμοποιούμε σε όλα τα προγράμματά μας.

Στην επόμενη παράγαρφο –αλλά και στα επόμενα κεφάλαια– θα χρειαστούμε μερικές οδηγίες ακόμη· τις βλέπουμε στη παρούσα παράγραφο.

Η οδηγία "`define`" (όρισε) έχει συντακτικό:

"`#`", "`define`", *όνομα*, *λίστα λεξικών οντοτήτων*

και ορίζει μια **μακροεντολή** ή **μακροσυνάρτηση** (macro). Το *όνομα* είναι το **όνομα της μακροεντολής** (macro identifier), ενώ η *λίστα λεξικών οντοτήτων* είναι το **σώμα της μακροεντολής** (macro body).

Αποτέλεσμα της οδηγίας είναι να αντικατασταθεί, στις γραμμές που ακολουθούν, το *όνομα* της μακροεντολής από αυτά που ορίζει η *λίστα λεξικών οντοτήτων*. Αυτή η διαδικασία λέγεται **μακροανάπτυξη** (macro expansion).

Η οδηγία "`define`" αναιρείται από μια οδηγία "`undef`"(ine):

"`#`", "`undef`", *όνομα*
με το ίδιο όνομα μακροεντολής.

Μετά από οδηγία "`undef`" το όνομα της μακροεντολής είναι **αόριστο** (undefined), όπως και πριν από την οδηγία "`define`". Μετά την define είναι **ορισμένο** (defined).

Παράδειγμα

Η οδηγία:

```
#define NMAX 50
```

ζητάει να αντικατασταθεί η λέξη *NMAX* με το "`50`" οπουδήποτε βρεθεί στη συνέχεια μέχρι να βρεθεί οδηγία "`#undef NMAX`".

Στο παρακάτω κομμάτι προγράμματος:

```
#define NMAX 50

int main()
{
   int a[NMAX];
      :
   for (int k(0); k <= NMAX-1; ++k)
   {
#undef NMAX
       swap(a[k], a[NMAX-k]);
   } // for
      :
```

η αντικατάσταση της *NMAX* από την τιμή 50 γίνεται μέχρι και τη γραμμή με την εντολή **for** ενώ δεν γίνεται στη γραμμή με την εντολή `swap(a[k], a[NMAX-k])` διότι προηγουμένως μεσολαβεί η οδηγία "`#undef NMAX`".Έτσι, ο μεταγλωττιστής θα μας βγάλει το μήνυμα "`Undefined symbol 'NMAX'`" για τη γραμμή αυτή.

Το *NMAX* είναι *ορισμένο* στις γραμμές του προγράμματος μεταξύ των οδηγιών "`#define NMAX 50`" και "`#undef NMAX`" ενώ είναι *αόριστο* πριν από την πρώτη και μετά τη δεύτερη.

Οι αντικαταστάσεις μπορεί να είναι και πιο πολύπλοκες:

Παράδειγμα

Αντιγράφουμε από το `stdlib.h`:

```
#define max(a,b)    (((a) > (b)) ? (a) : (b))
```

Αυτή η οδηγία θα έχει ως αποτέλεσμα μια εντολή σαν την:

```
      y = x + max(x+1, y-1);
```

να γίνει:

```
      y = x + (((x+1) > (y-1)) ? (x+1) : (y-1));
```

Η μακροανάπτυξη *δεν γίνεται* όταν το όνομα της μακροεντολής ευρεθεί:

- μέσα σε ορμαθό χαρακτήρων ή σε σχόλιο,
- μέσα στην ίδια τη μακροεντολή (π.χ. η "`#define A A`" δεν θα έχει ως αποτέλεσμα την επ' άπειρο εκτέλεσή της)

Φυσικά, μακροανάπτυξη *δεν γίνεται* και μέσα σε μια οδηγία "`undef`". Δηλαδή, για παράδειγμα, η

```
#define NMAX 50
```

δεν θα κάνει την

```
#undef NMAX
```

που ακολουθεί "`#undef 50`".

Και τι κερδίσαμε; Απλές συναρτήσεις, που καλούνται πολύ συχνά δεν «φορτώνουν» τον χρόνο εκτέλεσης του προγράμματος με το κόστος κλήσης συνάρτησης. Αυτό το πρόβλημα όμως το αντιμετωπίσαμε ήδη στην §14.1 με τις συναρτήσεις "`inline`". Έτσι, όσα είπαμε μέχρι εδώ για τη "`define`" σκοπό έχουν να την αναγνωρίζεις και να καταλαβαίνεις τη χρήση της σε άλλα προγράμματα. Στη C++

- η δήλωση σταθερών με "`const`" και
- οι συναρτήσεις "`inline`"

λύνουν τα αντίστοιχα προβλήματα με πολύ καλύτερο τρόπο και αυτά τα εργαλεία θα χρησιμοποιούμε.

Τώρα θα δούμε μια «περίεργη» μορφή της "`define`" την οποία και θα χρησιμοποιήσουμε στη συνέχεια.

Αν δεν υπάρχει σώμα της μακροεντολής τότε ο προεπεξεργαστής θα διαγράψει το όνομά της όπου το βρει. Π.χ. το:

```
#define NMAX
int main()
{
   int a[NMAX];

   for (int k(0); k <= NMAX-1; ++k)
   {
      swap(a[k], a[NMAX-k]);
   }
```

θα γίνει:

```
#define NMAX
int main()
{
   int a[];

   for (int k(0); k <= -1; ++k)
   {
      swap(a[k], a[-k]);
   }
```

Μια τέτοια χρήση της "`define`" κάνουμε όταν θέλουμε να έχουμε *ορισμένο* κάποιο όνομα σε μια περιοχή του προγράμματός μας, όπως θα δούμε στη συνέχεια.

Πολύ συχνά, θέλουμε να μεταγλωττίσουμε (ή να μη μεταγλωττίσουμε) ένα κομμάτι προγράμματος με κριτήριο το αν έχει ορισθεί ή δεν έχει ορισθεί κάποιο σύμβολο χωρίς να μας ενδιαφέρει η τιμή που μπορεί να του δόθηκε.

Ας πούμε ότι, σε κάποιο πρόγραμμα, αν έχει ορισθεί το σύμβολο *N* θέλουμε να έχουμε ως συνάρτηση *f* την

```
int f( int x )
{  ++x;  return x;  }
```

αλλιώς, αν δεν έχει ορισθεί το *N*, θέλουμε ως *f* την

```
int f( int x )
{  return x;  }
```

Αυτό γράφεται ως εξής:[1]

```
#ifdef N

int f( int x )
{  ++x;  return x;  }

#endif
```

[1] Υπάρχει και οδηγία "`#else`" και θα μπορούσαμε να το γράψουμε διαφορετικά αλλά ας την αφήσουμε...

```
#ifndef N

int f( int x )
{  return x;  }

#endif
```

Η "`#ifdef N`" είναι ο τρόπος που μας δίνει η C++ για να διατυπώσουμε την οδηγία: «`#if έχει ορισθεί το N`».

Για να διατυπώσουμε την "`#if δεν έχει ορισθεί το N`" θα πρέπει να γράψουμε: "`#ifndef N`".

Η "`#endif`" σημειώνει το τέλος αυτών που ισχύουν για την αντίστοιχη **if** ("`#ifdef N`" ή "`#ifndef N`").

Παράδειγμα

Αντιγράφουμε από το `iostream` (gcc – Dev-C++):

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

// . . . όλο το περιεχόμενο του αρχείου

#endif /* _GLIBCXX_IOSTREAM */
```

Ας δούμε πρώτα το πρόβλημα και μετά το νόημα των παραπάνω. Ας πούμε ότι έχεις ένα αρχείο, το `myFuncs.h`, με κάποιες συναρτήσεις. Μέσα στο αρχείο έχεις βάλει τη οδηγία:

```
#include <iostream>
```

Στο αρχείο `myprog.cpp`, που περιέχει ένα πρόγραμμα που γράφεις, βάζεις στην αρχή:

```
#include <iostream>
#include "myFuncs.h"
```

Τι θα μπορούσε να συμβεί στην περίπτωση αυτή; Οι δηλώσεις και οι ορισμοί του `iostream` θα υπάρχουν στο πρόγραμμά σου δύο φορές και αυτό δεν θα γίνει δεκτό από τον μεταγλωττιστή.

Το `iostream` αμύνεται, με τις οδηγίες που είδαμε πιο πάνω, ως εξής:

- Όταν εκτελεσθεί η "`#include <iostream>`" του `myprog.cpp` το σύμβολο "`_GLIBCXX_IOSTREAM`" δεν είναι ορισμένο. Έτσι, λόγω της
  `#ifndef _GLIBCXX_IOSTREAM`
  περικλείεται στο πρόγραμμά μας ολόκληρο το περιεχόμενο του `iostream`. Πρώτη και καλύτερη περικλείεται και εκτελείται η οδηγία
  `#define _GLIBCXX_IOSTREAM 1`
  Τώρα πια το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο.[2]
- Στη συνέχεια, όταν περιληφθεί το `myFuncs.h`, λόγω της οδηγίας
  `#include <iostream>`
  που υπάρχει εκεί, περιλαμβάνεται για δεύτερη φορά το `iostream`. Τώρα όμως, όταν εκτελείται η
  `#ifndef _GLIBCXX_IOSTREAM`
  το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο και δεν περιλαμβάνεται οτιδήποτε υπάρχει μέχρι και την
  `#endif /* _GLIBCXX_IOSTREAM */`

Αυτό θα πρέπει να κάνουμε και εμείς στα αρχεία με ορισμούς και δηλώσεις που θέλουμε να χρησιμοποιούμε συχνά. Για παράδειγμα, στο `myFuncs.h` θα πρέπει να περιλάβουμε όλο το περιεχόμενο μεταξύ των οδηγιών:

```
#ifndef _MYFUNCS_H
#define _MYFUNCS_H
```

[2] Εκείνο το "1" στη **define** δεν ειναι απαραίτητο, κατ' αρχήν.

```
// . . . όλο το περιεχόμενο του αρχείου

#endif  /* _MYFUNCS_H */
```

18.2 Μια Βιβλιοθήκη Περιγραμμάτων Συναρτήσεων

Το πιο απλό είδος βιβλιοθήκης είναι μια βιβλιοθήκη περιγραμμάτων συναρτήσεων. Θα κάνουμε λοιπόν και εμείς μια μικρή βιβλιοθήκη –τη *MyTmpltLib*– που θα περιέχει περιγράμματα που ήδη έχουμε χρησιμοποιήσει:

- Το περίγραμμα συνάρτησης *linSearch()*, που είδαμε στην §16.13.1, παρ' όλο που είπαμε ότι στις βιβλιοθήκες της C++ υπάρχει η *find()*.
- Το *renew()*, (§16.12), που μας είναι χρήσιμο μέχρι να μάθουμε τα περιγράμματα περιεχόντων (containers) της C++.
- Τα *new2d()* και *delete2d()* που είδαμε στην §16.9.

Σε ένα αρχείο, ας το πούμε `MyTmpltLib.h`, αντιγράφουμε τα τέσσερα περιγράμματα και πριν από όλα την κλάση εξαιρέσεων *MyTmpltLibXptn* (§16.9):

```
#ifndef _MYTMPLLIB_H
#define _MYTMPLLIB_H

#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// ΟΠΩΣ ΣΤΗΝ §16.13.1

template< typename T >
void renew( T*& p, int ni, int nf )
// ΟΠΩΣ ΣΤΗΝ §16.12

template< typename T >
T** new2d( int nR, int nC )
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
void delete2d( T**& a, int nR )
// ΟΠΩΣ ΣΤΗΝ §16.9

#endif // _MYTMPLLIB_H
```

Τα πάντα περιέχονται ανάμεσα στις `#ifndef _MYTMPLLIB_H` και `#endif`. Πριν από όλα η `#define _MYTMPLLIB_H`. Με αυτές αμυνόμαστε για περίπτωση πολλαπλής `#include "MyTmpltLib.h"`.

Ακόμη:

- Η `#include <string>` μας είναι απαραίτητη για τη *strncpy()*.
- Η `#include <new>` μας είναι απαράιτητη για τη *bad_alloc*.
- Η using `namespace std` μας είναι απαραίτητη για να μη γράφουμε `std::strncpy`, `std::bad_alloc`.

Να δούμε τώρα πώς τη χρησιμοποιούμε. Το πρόγραμμα της §16.12 γίνεται:

```
#include <iostream>
```

```
#include <new>
#include "MyTmpltLib.h"

using namespace std;

int main()
{
   int*    ip;
   double* dp;
// . . .
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
// . . .
} // main
```

Δηλαδή: υπάρχει η `#include "MyTmpltLib.h"` και δεν υπάρχει το περίγραμμα *renew()*. Από ολόκληρο το `MyTmpltLib.h` χρησιμοποιείται μόνο το *renew* από το οποίο δημιουργούνται δύο περιπτώσεις: μια για `int` (λογω της `renew(ip, 3, 7)`) και μια για `double` (λογω της `renew(dp, 3, 7)`).

Στο πρόγραμμα του Παράδ. 2 της §16.13 θα έχουμε:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>
#include "MyTmpltLib.h"

using namespace std;

struct ApplicXptn
// . . .
```

Και εδώ βάλαμε την `#include "MyTmpltLib.h"` και αφαιρέσαμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *renew()* και *linSearch()*. Από το *renew()* θα πάρουμε ένα στιγμιότυπο για τον τύπο *GrElmn* και από τη *linSearch()* ένα στιγμιότυπο για τον ίδιο τύπο.

Στο πρόγραμμα πολλαπλασιαμού πινάκων (§16.9) θα έχουμε:

```
#include <iostream>
#include <fstream>
#include "MyTmpltLib.h"

using namespace std;

void input2DAr( istream& tin, int** a, int nRow, int nCol );
void output2DAr( ostream& tout, int** a, int nRow, int nCol );

int main()
// . . .
```

Και πάλι βάζουμε την `#include "MyTmpltLib.h"` και αφαιρούμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *new2d* και *delete2d*. Παίρνουμε μια περίπτωση για το κάθε ένα για τον τύπο `int`.

Ως προς τη σύνδεση: τι είδους βιβλιοθήκη κάναμε; Στατική ή δυναμικής σύνδεσης; Παρ' όλο που οι όροι αυτοί αναφέρονται σε βιβλιοθήκες με μεταγλωττισμένο περιεχόμενο, μπορούμε να πούμε ότι έχουμε μια βιβλιοθήκη *στατική*· ο συνδέτης καλείται μόνο μια φορά να δώσει ένα αυθύπαρκτο εκτελέσιμο πρόγραμμα.

Όπως βλέπεις, τα πράγματα είναι πολύ απλά. Βέβαια, αν δεις τα πράγματα από εμπορική άποψη υπάρχει ένα προβληματάκι: Αν θέλεις να πουλήσεις προγράμματα σε κάποιον πελάτη του δίνεις υποχρεωτικώς το αρχικό πρόγραμμα, σε C++. Αλλά το αρχικό πρόγραμμα, συνήθως, τιμολογείται πολύ ακριβά. Είναι δυνατόν να του δίνεις μεταγλωττισμένες συναρτήσεις μόνο; Αυτό βλέπουμε στη συνέχεια.

18.3 Χωριστή Μεταγλώττιση

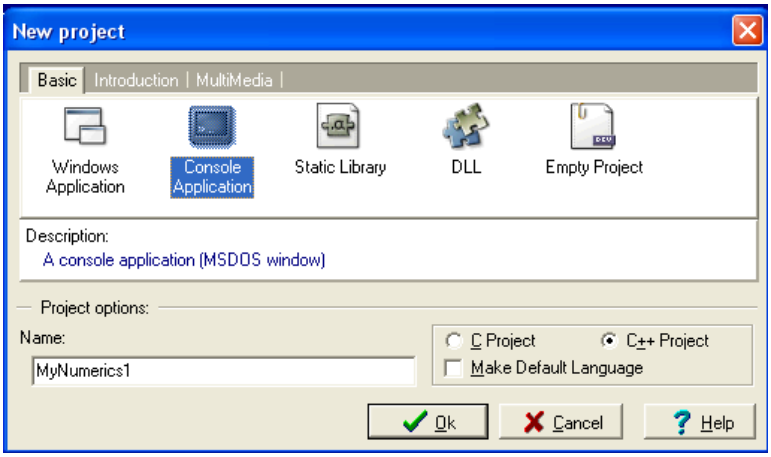
Ο επόμενος στόχος μας είναι να δημιουργήσουμε μια ακόμη πιο μικρή βιβλιοθήκη με τους δύο αριθμητικούς αλγόριθμους που έχουμε μεταφράσει σε πρόγραμμα: τον *αλγόριθμο του Horner* (§9.4, Παράδ. 1) και τον αλγόριθμο της διχοτόμησης (bisection, §17.14).

Ας ξεκινήσουμε ως εξής: Βάζουμε τις δύο συναρτήσεις σε ένα αρχείο με όνομα `MyNumerics.cpp`. Στο αρχείο `MyNumerics.h` βάζουμε τα εξής:

```
#ifndef _MYNUMERICS_H
#define _MYNUMERICS_H

#include <string>

using namespace std;

struct MyNumericsXptn
{
   enum {  domainError, noArray  };

   char   funcName[100];
   int    errCode;
   double errDblVal;
   MyNumericsXptn( const char* fn, int ec, double ev = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errDblVal = ev;  }
}; // MyNumericsXptn

// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x );

// bisection -- προσεγγίζει λύση (root) της εξίσωσης f(x) = 0
//              στο διάστημα [α,β] με τη μέθοδο της διχοτόμησης
//              με nMax επαναλήψεις το πολύ.
//              Στο [a,b] η f πρέπει να είναι συνεχής και
//              f(a)*f(b) <= 0
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );

#endif // _MYNUMERICS_H
```

Στο `MyNumerics.cpp` βάζουμε:

```
#include <string>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNumericsXptn( "ph", MyNumericsXptn::noArray );
   if ( m < 0 )
      throw MyNumericsXptn( "ph",
                            MyNumericsXptn::domainError, m );
// τα υπόλοιπα όπως ήταν

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNumericsXptn( "bisection",
```



Σχ. 18-1 Ο πίνακας επιλογής "New Project" του περιβάλλοντος ανάπτυξης Dev C++.

```
                                MyNumericsXptn::domainError, epsilon );
// τα υπόλοιπα όπως ήταν
```

Παρατηρήσεις: ▶

1. Όπως βλέπεις, τώρα οι συναρτήσεις μας ρίχνουν και εξαιρέσεις. Όταν έχεις να μετατρέψεις μια συνάρτηση που την έχεις αναπτύξει για μια συγκεκριμένη εφαρμογή, όπου η χρήση της είναι –πιθανότατα– ελεγχόμενη, σε συνάρτηση «γενικής χρήσης» θα πρέπει να κάνεις κάτι τέτοιες αλλαγές.
2. Η *bisection()* ρίχνει εξαίρεση αν κληθεί με *epsilon* < 0. Τα υπόλοιπα προβλήματα αντιμετωπίζονται με επιστρεφόμενο κωδικό λάθους.◀

Φυλάγουμε αυτά τα αρχεία στο ευρετήριο **MyNumerics1**.

Στο περιβάλλον ανάπτυξης της Dev-C++ επιλέγουμε:

File|New|Project

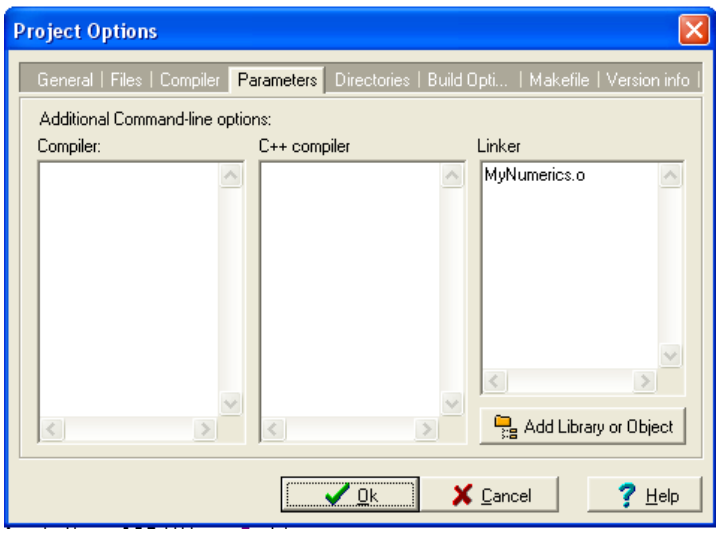
και βλέπουμε τον πίνακα επιλογής που βλέπεις στο Σχ. 18-1. Στο "**Name:**" γράφουμε **MyNumerics1** και επιλέγουμε ως είδος project "**Console Application**". Δίνοντας "**Ok**" μας ζητείται να επιλέξουμε πού θα αποθηκευτεί. Επιλέγουμε το ευρετήριο **MyNumerics1**. Στο **MyNumerics1**, δημιουργείται ένα αρχείο με όνομα **MyNumerics1.dev**. Αυτό είναι το αρχείο του project.

Στο περιβάλλον ανάπτυξης (επιλογή "**Project**" στο αριστερό παράθυρο) βλέπεις ότι έχεις ανοιγμένο το project **MyNumerics1**, που έχει ως περιεχόμενο το αρχείο **main.cpp**· το βλέπεις στο δεξιό παράθυρο:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Τώρα, επιλέγοντας **Project|Add to Project** ζητούμε να προστεθεί στο project το αρχείο **MyNumerics.cpp**. Βλέπουμε το όνομά του να εμφανίζεται στο αριστερό παράθυρο. Τέλος, επιλέγουμε **Execute|Rebuild all** και βλέπουμε να εμφανίζονται στο **MyNumerics1** τα εξής αρχεία:

```
main.o
Makefile.win
MyNumerics.o
MyNumerics1.exe
```



Σχ. 18-2 Ο πίνακας επιλογής "Project Options" του περιβάλλοντος ανάπτυξης Dev C++.

Μας ενδιαφέρει το **MyNumerics.o** που είναι το «μεταγλωττισμένο **MyNumerics.cpp**».[3] Δες τώρα πώς θα το χρησιμοποιήσουμε.

Απλουστεύουμε το πρόγραμμα της §17.14 τόσο:

```
#include <iostream>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double q( double x );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
}

double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και το φυλάγουμε στο **testBisection.cpp** στο ευρετήριο **testBisection**.

Όπως βλέπεις, το μόνο που υπάρχει από τη *bisection()* είναι η επικεφαλίδα της που υπάρχει στο **MyNumerics.h**. Παρ' όλα αυτά την καλούμε τρεις φορές.

Αντιγράφουμε στο **testBisection** τα **MyNumerics.h** και **MyNumerics.o**.

Και τώρα:

- Δημιουργούμε ένα project είδους **Console Application**, με όνομα **testBisection** και το φυλάγουμε στο ευρετήριο **testBisection**.

[3] Αλλού, μπορεί να το δεις ως **MyNumerics.obj**.

- Προσθέτουμε στο project το αρχείο **testBisection.cpp**.
- Κλείνουμε, *χωρίς να φυλάξουμε*, το **main.cpp**. Με τον τρόπο αυτόν το αρχείο διαγράφεται από το project όπου μένει μόνο το **testBisection.cpp**.
- Επιλέγουμε **Project|Project options** και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "**Parameters**".
- Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2) με το "**Add Library or Object**" επιλέγουμε το **MyNumerics.o**. Με αυτόν τον τρόπο ζητούμε από τον συνδέτη, να συνδεσει και το περιεχόμενο αυτού του αρχείου στο εκτελέσιμο που θα δημιουργήσει.
- Για να πάρουμε το εκτελέσιμο, επιλέγουμε **Execute|Rebuild all**.

Στο **testBisection** θα δημιουργηθούν τα εξής αρχεία:

```
testBisection.exe
testBisection.o
```

Το **testBisection.exe** είναι το εκτελέσιμο. Το «**testBisection.o** είναι το μεταγλωττισμένο **testBisection.cpp**».

Στο παράδειγμα αυτό βλέπουμε το εξής: *Το πρόγραμμα αποτελείται από δύο αρχεία που μεταγλωττίζονται ξεχωριστά*. Αν ξαναγυρίσουμε σε αυτά που λέγαμε –για το τι θα δώσουμε στον πελάτη χωρίς να του αποκαλύψουμε τα μυστικά μας– τώρα έχουμε μιαν απάντηση: θα του δώσουμε τα **MyNumerics.h** και **MyNumerics.o**.

Παρατηρήσεις: ▶

1. Χρησιμοποιήσαμε το όνομα "**testBisection**" α) για το project, β) για το ευρετήριο που βάλαμε τα αρχεία και γ) για το αρχείο **cpp** στο οποίο υπάρχει η **main**. Θα μπορούσαμε να χρησιμοποιήσουμε τρία διαφορετικά ονόματα. *Το όνομα του εκτελέσιμου θα είναι ίδιο με αυτό του project*.
2. Στο πρόγραμμά μας εντάσσεται και η *ph* αφού υπάρχει στο **MyNumerics.cpp** επομένως και στο **MyNumerics.o**, παρ' όλο που δεν καλείται από το πρόγραμμά μας. Θα το «ξεφορτωθούμε» στη συνέχεια.
3. Θα μπορούσαμε να είχαμε βάλει ("Add to project") το **MyNumerics.cpp** στο project **testBisection** και να είχαμε και πάλι ξεχωριστή μεταγλώττιση, με την έννοια ότι θα είχαμε δύο ξεχωριστά αρχεία **testBisection.o** και **MyNumerics.o**. Αλλά, σε μια τέτοια περίπτωση μπορεί να σου έμενε η υποψία ότι η **main** «έβλεπε» τη *bisection()*. Τις ξεχωρίσαμε λοιπόν τελείως για να σε πείσουμε.
4. Δες έναν άλλον τρόπο που μπορείς να γράψεις το πρόγραμμά σου:

```
#include <iostream>
#include <cmath>

using namespace std;

extern "C++"
{
   struct MyNumericsXptn;
   void bisection( double (*f)(double),
                   double a, double b, double epsilon,
                   int nMax,
                   double& root, int& errCode );
}

double q( double x );

int main()
// τα υπόλοιπα όπως ήταν
```

Όπως βλέπεις, δεν βάλαμε την **#include "MyNumerics.h"** αλλά βάλαμε την **extern** η οποία δηλώνει ότι σε κάποιο άλλο αρχείο, που θα συνδεθεί στο πρόγραμμα, υπάρχει ένας τύπος δομής με το όνομα *MyNumericsXptn* και μια συνάρτηση με το όνομα *bisection* και την επικεφαλίδα που φαίνεται.

Αν το αρχείο που συνδέουμε έχει βγεί από μεταγλωττιστή C θα έπρεπε να δηλώσουμε "`extern "C"`". Τα διάφορα περιβάλλοντα ανάπτυξης σου επιτρέπουν να συνδέσεις αρχεία που έχουν βγει από μεταγλωττιστές άλλων γλωσσών. Φυσικά, σου επιτρέπουν να κάνεις και την κατάλληλη δήλωση **extern**, π.χ. "`extern "Ada"`", "`extern "FORTRAN"`" κλπ.

5. Δεν είπαμε οτιδήποτε για το `Makefile.win`. Δεν θα πούμε τίποτε περισσότερο από το ότι σε αυτό υπάρχουν οι εντολές για το κτίσιμο του εκτελέσιμου.

6. Αυτά που είπαμε για το περιβάλλον ανάπτυξης Dev C++ ισχύουν περίπου και για το περιβάλλον ανάπτυξης Code::Blocks. Για την Borland C++ v.5.5 σε ένα («μαύρο») παράθυρο `cmd.exe` δώσε:

```
C:\Borland\bc55\bin>bcc32 main.cpp MyNumerics.cpp
```

όπου "`main.cpp`" είναι αυτό που είδαμε πιο πάνω. Έτσι, θα πάρεις το `MyNumerics.obj`. Στη συνέχεια δίνεις:

```
C:\Borland\bc55\bin>bcc32 testBisection.cpp MyNumerics.obj
```

και παίρνεις το "`testBisection.exe`".◀

18.4 Μια Στατική Βιβλιοθήκη

Και τώρα θα κάνουμε μια *στατική βιβλιοθήκη* που θα περιέχει τις δύο συναρτήσεις *ph()* και *bisection()*.

Στο ευρετήριο `MyNumerics` αντιγράφουμε και πάλι τα `MyNumerics.cpp` και `MyNumerics.h` και –όπως περιγράψαμε στην προηγούμενη παράγραφο– δημιουργούμε ένα project με το όνομα `MyNumerics` αλλά αυτή τη φορά διαφορετικού είδους: "`Static Library`". Προσθέτουμε (Add to project) το `MyNumerics.cpp` και δίνουμε `Execute|Rebuild all`. Στο `MyNumerics`, εκτός από τα `MyNumerics.dev` και `MyNumerics.o`, θα δεις και το `MyNumerics.a`. Αυτή είναι η στατική βιβλιοθήκη[4] που θα χρησιμοποιήσουμε για να ξαναχτισουμε το πρόγραμμά μας.

Στο ευρετήριο `testBisectionL` αντιγράφουμε τα

```
testBisection.cpp
MyNumerics.a
MyNumerics.h
```

Στη συνέχεια:

- Δημιουργούμε ένα project "`Console Application`"με το όνομα `testBisectionL` και
- προσθέτουμε το `testBisection.cpp` (κλείνουμε χωρίς να φυλάξουμε το `main.cpp`).
- Επιλέγουμε `Project|Project options` και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "`Parameters`". Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2), με το "`Add Library or Object`", επιλέγουμε τη βιβλιοθήκη `MyNumerics.a`.
- Τέλος, επιλέγουμε `Execute|Rebuild all` για να πάρουμε το εκτελέσιμο που θα έχει το όνομα `testBisectionL.exe`,

Αυτό το εκτελέσιμο διαφέρει από το προηγούμενο ως προς το ότι δεν έχει και τη συνάρτηση *ph()*, αλλά μόνον τη *bisection()*.

Όπως καταλαβαίνεις, στον πελάτη, για τον οποίον συζητούσαμε, θα δώσουμε το `MyNumerics.a` και το `MyNumerics.h`.

---

[4] Μια άλλη συνηθισμένη κατάληξη (π.χ. από τη Borland C++) για στατικές βιβλιοθήκες είναι η "`.lib`".

18.5 "namespace": Το Πρόβλημα και η Λύση

Ας πούμε τώρα ότι γράψαμε τη βιβλιοθήκη περιγραμμάτων, γράψαμε τη στατική βιβλιοθήκη μας και θέλουμε να τις χρησιμοποιήσουμε σε ένα πρόγραμμα που γράφουμε μαζί με δυο βιβλιοθήκες που «κατεβάσαμε» από το Internet. Και ξαφνικά, ο μεταγλωττιστής διαμαρτύρεται: βρίσκει δύο συναρτήσεις με το όνομα *renew()* και δύο συναρτήσεις με το όνομα *ph()*. Και τώρα τι γίνεται; Καλά, άντε και αλλάζουμε αυτά τα δύο ονόματα. Είναι σχεδόν σίγουρο ότι μετά από λίγο θα έχουμε άλλο παρόμοιο πρόβλημα. Μηπως να βάλουμε κάποια «εξωφρενικά» ονόματα στις συναρτήσεις μας. Ε, όχι και έτσι...

Η C++ προσφέρει μια λύση για το πρόβλημα: τον **ονοματοχώρο** (namespace). Δες ένα παράδειγμα. Σε κάποιο πρόγραμμα έχουμε:

```
namespace test
{
   const double x = 1.5;
}
char x = 'A';
int main()
{
   int x = 4;
      :
   cout << x << "  " << ::x << "  " << test::x << endl;
```

Εδώ βλέπουμε:

- Έναν ονοματοχώρο, με όνομα *test*, όπου δηλώνεται μια σταθερά τύπου `double` με όνομα *x* και τιμή 1.5.
- Μια δήλωση της καθολικής μεταβλητής *x* τύπου `char` με αρχική τιμή `'A'`.
- Μια δήλωση μεταβλητής τύπου `int`, τοπικής στην `main`, με *x* όνομα αρχική τιμή 4.

Από τη `main` μπορούμε να έχουμε πρόσβαση και στα τρία παραπάνω αντικείμενα, όπως φαίνεται και από την εντολή εξόδου που δίνει:

```
4  A  1.5
```

Όπως ήδη ξέρουμε, με τα `x` και `::x` παίρνουμε την τοπική και την καθολική μεταβλητή αντιστοίχως. Με το `test::x` εννοούμε: το αντικείμενο με το όνομα *x* που δηλώνεται στον ονοματοχώρο *test*. Όπως βλέπεις, ο τελεστής "`::`" είναι ο τελεστής **επίλυσης** των προβλημάτων **εμβέλειας** (scope resolution) στο πρόγραμμά μας

Έστω λοιπόν ότι μας δίνεται μια βιβλιοθήκη προγραμμάτων. Αν ξέρεις ότι μέσα στη βιβλιοθήκη υπάρχει δηλωμένος κάποιος ονοματοχώρος δεν έχεις πρόβλημα. Αν δεν υπάρχει ονοματοχώρος –αν π.χ. είναι προγράμματα C– μπορείς να κάνεις το εξής: αν, ας πούμε, οι δηλώσεις της βιβλιοθήκης υπάρχουν στο αρχείο `bib.h` δηλώνεις:

```
namespace xbib
{
#include "bib.h"
}
```

Ύστερα από αυτό μέσα στο πρόγραμμά σου χρησιμοποιείς οτιδήποτε δηλώνεται στο bib.h με το πρόθεμα "`xbib::` ".

Φυσικά, η λύση που δίνεται με το **namespace** μπορεί να δημιουργήσει το αντίθετο πρόβλημα: να θέλεις να χρησιμοποιήσεις μια βιβλιοθήκη, να μην υπάρχει πρόβλημα με τα ονόματα και παρ' όλα αυτά να έχεις την υποχρέωση να γράφεις το όνομα του ονοματοχώρου πριν από όλα τα αντικείμενα που χρησιμοποιείς. Υπάρχει λύση και γι' αυτό. Ας πούμε ότι χρησιμοποιείς μια βιβλιοθήκη, που οι δηλώσεις της υπάρχουν στο `prlib.h` στον ονοματοχώρο *prlib*. Αν βάλεις στο πρόγραμμά σου:

```
#include "prlib.h"
using namespace prlib;
```

μπορείς να χρησιμοποιείς οτιδήποτε υπάρχει στον ονοματοχώρο *prlib* χωρίς το πρόθεμα "`prlib::`".

Αντί για την καθολική οδηγία "`using namespace`" που κάνει ορατά τα πάντα παντού, υπάρχει και η δήλωση **`using`** με την οποία μπορείς να κάνεις πιο επιλεκτική δουλειά. Ας πούμε ότι εσύ θέλεις στη συνάρτηση *f1()* να χρησιμοποιήσεις τον τύπο *cl1* της *prlib* ενώ στη συνάρτηση *f2()* θέλεις να χρησιμοποιήσεις τη συνάρτηση *fl0()* της *prlib*. Μπορείς να γράψεις:

```
double f1(...)
{
   using prlib::cl1;
   cl1 a, b;
      :
} // f1

void f2(...)
{
   using prlib::fl0;
      :
   q = fl0(1, u) + w;
} // f2
```

Η δήλωση "`using prlib::cl1`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα **`cl1`** μέσα στη συνάρτηση *f1()* μόνον. Παρομοίως, η δήλωση "`using prlib::fl0`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα "`fl0`" μέσα στη συνάρτηση *f2()* μόνον. Για τη δήλωση **using** ισχύουν οι κανόνες εμβέλειας που ξέρουμε.

Τώρα μπορείς να καταλάβεις και τις δηλώσεις "`using namespace std`" που βάζουμε στα προγράμματά μας. Η C++ έχει τον δικό της πάγιο ονοματοχώρο με το όνομα "`std`" (**st**and**d**). Μέσα σε αυτόν είναι δηλωμένα τα πάντα. Αν δεν το βάζαμε θα έπρεπε να γράφουμε **`std::strncpy`**, **`std::cout`**, **`std::endl`**, **`std::cin`**, **`std::bad_alloc`**, κλπ.

Ας έλθουμε τώρα στα παραδείγματά μας. Στο αρχείο **`MyTmpltLib.h`** κάνουμε την εξής αλλαγή:

```
namespace MyTmplt
{
struct MyTmpltLibXptn
// . . .
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// . . .
template< typename T >
void renew( T*& p, int ni, int nf )
// . . .
template< typename T >
T** new2d( int nR, int nC )
// . . .
template< typename T >
void delete2d( T**& a, int nR )
// . . .
} // namespace MyTmplt
```

Έτσι, τα ονόματα: *MyTmpltLibXptn*, *linSearch*, *renew*, *new2d* και *delete2d* εισάγονται στον ονοματοχώρο *MyTmplt*.

Εδώ όμως πρόσεξε το εξής:

- ***Ο ορισμός δομής (κλάσης) ορίζει και έναν ονοματοχώρο με το όνομα της δομής.***

Έχουμε λοιπον τον ονοματοχώρο *MyTmpltLibXptn* φωλιασμένο μέσα στον ονοματοχώρο *MyTmplt*. Έτσι, στη διαχείριση εξαιρέσεων στο πρόγραμμα του Παράδ. 2 της §16.13 οι κωδικοί σφάλματος θα έχουν «διπλό πρόθεμα», για παράδειγμα,

"`MyTmplt::MyTmpltLibXptn::domainError`"

```
// . . .
   catch( MyTmplt::MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
```

```
         case MyTmplt::MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
// . . .
```

Ακόμη, στην *elmntInTable* θα έχουμε:

```
// . . .
   int lPos( MyTmplt::linSearch( grElmnTbl, nElmn,
                                 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         MyTmplt::renew( grElmnTbl, nElmn, nReserved+incr );
// . . .
```

Ας δούμε τώρα πώς θα δηλώσουμε ονοματοχώρο στη στατική βιβλιοθήκη μας. Στο αρχείο **MyNumerics.h** κάνουμε την εξής αλλαγή:

```
namespace MyNmr
{
struct MyNumericsXptn
// . . .
double ph( const double a[], int m, double x );
// . . .
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );
} // namespace MyNmr
```

ενώ στο **MyNumerics.cpp** θα πρέπει να βάλουμε:

```
// . . .
double MyNmr::ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNmr::MyNumericsXptn( "ph",
                                   MyNmr::MyNumericsXptn::noArray );
// . . .
void MyNmr::bisection( double (*f)(double),
                       double a, double b, double epsilon,
                       int nMax,
                       double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNmr::MyNumericsXptn( "linSearch",
                     MyNmr::MyNumericsXptn::domainError, epsilon );
// . . .
```

Δεν θα μπορούσαμε να βάλουμε στο **MyNumerics.cpp** μια

```
using namespace MyNmr;
```

και να γλυτώσουμε από όλα αυτά τα "`MyNmr`"; Όχι! Οι ορισμοί των δύο συναρτήσεων είναι πλήρεις και ο μεταγλωττιστής θα τις μεταγλωττίσει ως *ph()* και *bisection()* και όχι ως *MyNmr*::*ph()* και *MyNmr*::*bisection()*.

Στο `testBisection.cpp` θα πρέπει να βάλουμε:

```
MyNmr::bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
```

Αν είχαμε και διαχείριση εξαιρέσεων θα βάζαμε:

```
catch( MyNmr::MyNumericsXptn& x )
{
   if ( x.errCode == MyNmr::MyNumericsXptn::domainError )
      cout << x.funcName << " called with negative ( "
           << x.errDblVal << " ) epsilon" << endl;
   else
      cout << "unexpected MyNumericsXptn from "
           << x.funcName << endl;
}
```

Αν θέλεις μπορείς να χρησιμοποιήσεις τον ίδιο ονοματοχώρο για πολλές βιβλιοθήκες. Για παράδειγμα, αντί για τα δύο ονόματα, *MyTmplt* και *MyNmr*, θα μπορούσαμε να χρησιμοποιήσουμε ένα μόνο όνομα, ας πούμε *MyNmspc*.

Γενικώς, να θυμάσαι ότι:

♦ ***Μια καλή βιβλιοθήκη έχει και τον ονοματοχώρο της.***

Αυτό μπορεί να σε γλυτώσει από προβλήματα χωρίς να σου δημιουργεί νέα αφού αρκεί μια "`using namespace …`" για να σου επιτρέψει να τον αγνοήσεις όπου δεν τον χρειάζεσαι.

18.6 Ανακεφαλαίωση

Οι βιβλιοθήκες συναρτήσεων είναι πολύτιμα εργαλεία για τον προγραμματιστή. Στο διαδίκτυο υπάρχουν πολλές και αρκετές από αυτές είναι πολύ καλής ποιότητας. Μπορείς να κτίσεις και δικές σου.

Οι *βιβλιοθήκες περιγραμμάτων συναρτήσεων* είναι οι πιο απλές αφού το περιεχόμενό τους είναι γραμμένο σε C++. Περιέχουν περιγράμματα συναρτήσεων, εξειδικεύσεις τους και πιθανότατα κάποια κλάση εξαιρέσεων. Καλό είναι να δηλώνονται μέσα σε έναν *ονοματοχώρο*.

Πριν δούμε τις βιβλιοθήκες μεταγλωττισμένων συναρτήσεων είδαμε πώς μπορούμε να μεταγλωττίζουμε χωριστά διάφορα κομμάτια του προγράμματος και με τη σύνδεση να παίρνουμε το εκτελέσιμο.

Οι βιβλιοθήκες μεταγλωττισμένων συναρτήσεων χωρίζονται σε δύο μεγάλες κατηγορίες: τις *στατικές* και τις *δυναμικής σύνδεσης*.

- Στατικές: Οι συναρτήσεις τους, που καλούνται από κάποιο πρόγραμμα, ενσωματώνονται (αντιγράφονται) στο εκτελέσιμο που μπορεί να εκτελείται χωρίς να είναι παρούσα η βιβλιοθήκη.
- Δυναμικής Σύνδεσης: Οι συναρτήσεις τους δεν ενσωματώνονται στο εκτελέσιμο αλλά φορτώνονται στη μνήμη όταν αυτό εκτελείται.

Για τη χωριστή μεταγλώτιση και τη δημιουργία και τη χρήση βιβλιοθηκών τηρούνται διαδικασίες που εξαρτώνται σε μεγαλύτερο ή μικρότερο βαθμό από το περιβάλλον ανάπτυξης ή/και το ΛΣ.

Όταν έχεις χωριστή μεταγλώττιση ή χρησιμοποιείς βιβλιοθήκες συναρτήσεων είναι απαραίτητα τα αρχεία επικεφαλίδων (`.h`).

Ασκήσεις

Α Ομάδα

18-1 Στις συναρτήσεις *ph()* και *bisection()* αντιστοιχίσαμε στο $\mathbb{R}$ τον `double`. Θα μπορούσαμε να είχαμε αντιστοιχίσει τον `float` ή τον `long double`. Αν κάνουμε τις συναρτήσεις περιγράμματα δίνουμε στον προγραμματιστή τη δυνατότητα να επιλέξει τον τύπο για την περίπτωση που τον ενδιαφέρει.

Αφού μετατρέψεις τις δύο συναρτήσεις σε περιγράμματα, να τις ενταξεις σε μια βιβλιοθήκη περιγραμμάτων `NumTmplt`. Γράψε προγράμματα για να τη δοκιμάσεις.

Β Ομάδα

18-2 Στο Κεφ. 15 είδαμε τον τύπο

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

και τις συναρτήσεις

```
GrElmn GrElmn_copyFromElmn( const Elmn& a );
void GrElmn_save( const GrElmn& a, ostream& bout );
void GrElmn_load( GrElmn& a, istream& bin );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_display( const GrElmn& a, ostream& tout );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_writeToTable( const GrElmn& a, ostream& tout );
```

για τη διαχείριση των στοιχείων του.

Κάνε μια στατική βιβλιοθήκη `GrElmnLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις. Δοκίμασέ την στα προγράμματα των §15.14.1 και 15.14.2.

Προσοχή! Δυο συναρτήσεις ρίχνουν εξαιρέσεις *ApplicXptn*. Αυτό πρέπει να αλλάξει: θα ρίχνουν εξαιρέσεις `GrElmnXptn`. Φυσικά, αυτόν τον τύπο θα τον ορίσεις εσύ!

18-3 Στο Κεφ. 15 είδαμε και τον τύπο

```
struct Date
{
   enum { saveSize = sizeof(int) + 2*sizeof(char) };
   unsigned int  year;
   unsigned char month;
   unsigned char day;
   Date( int yp=1, int mp=1, int dp=1 )
   {  year = yp;  month = mp;  day = dp;  }
}; // Date
```

του οποίου τα στοιχεία διαχειριστήκαμε με τις συναρτήσεις

```
void Date_save( const Date& a, ostream& bout );
```

```
void Date_load( Date& a, istream& bin );
```

και τους τελεστές

```
bool operator==( const Date& lhs, const Date& rhs );
bool operator<( const Date& lhs, const Date& rhs );
ostream& operator<<( ostream& tout, const Date& rhs );
```

Κάνε μια στατική βιβλιοθήκη `DateLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις και τους τελεστές. Δοκίμασέ την σε αυτά που λέγαμε για τον τύπο *Date*.

Ξαναλύσε την Άσκ. 15-1 με χρήση της βιβλιοθήκης.

Προσοχή! Οι συναρτήσεις αυτές δεν ρίχνουν εξαιρέσεις αλλά θα έπρεπε. Δες την προηγούμενη άσκηση και τις (αντίστοιχες) συναρτήσεις της.