Μέρος Α
Εισαγωγή στον Προγραμματισμό

κεφάλαιο

0

Επεξεργασία Στοιχείων – Προγραμματισμός

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

Προσδοκώμενα αποτελέσματα:

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

Έννοιες κλειδιά:

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

Περιεχόμενα:

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

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

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

Παράδειγμα 1

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

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

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

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

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

Παράδειγμα 2

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

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

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

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

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

0.1 Αλγόριθμοι

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

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

17 13 67 104 2 69 45 375 35 84

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

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

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

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

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

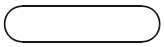
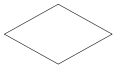
Ειδικώς για αυτήν την περίπτωση, ονομάζουμε a_k, $k = 0..9$ τους αριθμούς που έχουμε, m το μέγιστο που ψάχνουμε, k_m τη θέση του μέγιστου μέσα στη δεκάδα. Και να πώς γράφεται ο αλγόριθμός μας:

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

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

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

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

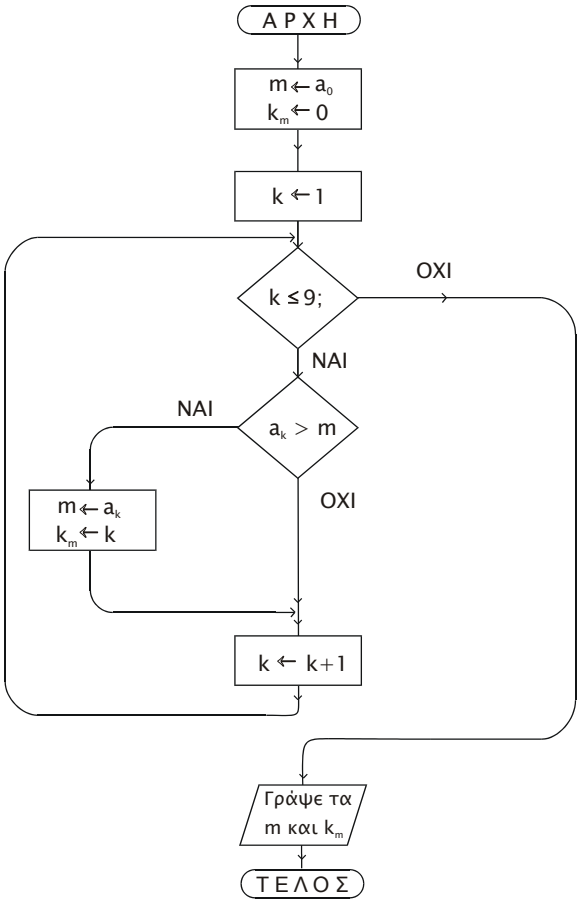
Σύμβολο	Χρήση
	για αρχή ή τέλος αλγορίθμου
	για διάβασμα (είσοδο) / γράψιμο (έξοδο) στοιχείων
	ρόμβος για αποφάσεις
	για περιγραφή επεξεργασίας
← ↑ → ↓	βέλη ροής

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

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

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

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



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

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

0.2 Προγραμματα και Γλώσσες Προγραμματισμού

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

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

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

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

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

Παράδειγμα

Στη γλώσσα μηχανής κάποιου υπολογιστή οι εντολές:

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

θα μπορούσαν να σημαίνουν τα εξής:

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

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

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

---

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

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

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

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

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

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

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

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

```
X ← A + B
```

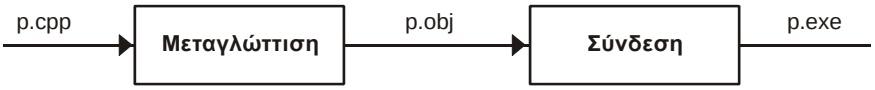
ή κάτι παρόμοιο.

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

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

---

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



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

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

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

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

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

0.3 Το Σωστό Πρόγραμμα

Ας πούμε ότι γράφεις το πρόγραμμα που είδαμε στην §0.2

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

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

Ας δούμε τώρα κάτι άλλο:

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

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

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

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

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

Δες και το παρακάτω:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ή αλλιώς:

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

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

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

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

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

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

Αν αποδείξουμε ότι

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

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

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

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

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

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

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

---

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

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

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

[5] ή λογικές ανώτερης τάξης.

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

0.3.1 Συμπερασματικοί Κανόνες

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

- Αν για κάθε x ισχύει η $P(x)$

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η $P(a)$ για τυχόν a.

Αυτό γράφεται συμβολικώς ως εξής:

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

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

Ας δούμε ακόμη έναν κανόνα από τον Προτασιακό Λογισμό:

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

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η Q.

Συμβολικώς (modus ponens):

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

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

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

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

Γράφεις το πρόγραμμα, αλλά στην απόδειξή του παίρνεις:

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

Είναι σωστό το πρόγραμμα E ή όχι;

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

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

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

τότε

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

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

Συμβολικώς:

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

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

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

τότε

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

Συμβολικώς:

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

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

0.4 Από τις Προδιαγραφές στο Πρόγραμμα

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

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

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

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

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

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

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

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

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

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

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

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

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

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

τότε

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

Συμβολικώς:

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

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

0.5 Αποδοτικότητα Προγράμματος

Δες πώς αλλιώς μπορούμε να γράψουμε τον αλγόριθμο του μεγίστου:

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

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

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

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

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

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

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

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

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

0.6 Η Γλώσσα C++

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

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

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

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

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

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

---

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

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

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

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

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

0.7 Ο Συμβολισμός BNF

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

Ας ξεκινήσουμε με ένα

Παράδειγμα 1

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

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

Όπως βλέπεις, ένας ορισμός οντότητας γράφεται ως εξής:

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

Ο κανόνας γράφεται ως εξής:

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

Παράδειγμα 2

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

διψήφιος αριθμός = *ψηφίο*, *ψηφίο*;

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

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

Έτσι, οι κανόνες του παραδ. 1 γράφονται:

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

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

Παράδειγμα 3

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

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

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

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

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

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

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

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

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

Παράδειγμα 4

Αντί να γράψουμε:

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

γράφουμε:

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

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

κεφάλαιο

1

Υπολογισμοί με Σταθερές

Ο στόχος μας σε αυτό το κεφάλαιο:

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

Προσδοκώμενα αποτελέσματα:

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

Έννοιες κλειδιά:

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

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Το Σύνολο Χαρακτήρων:

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

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

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

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

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

και πεζά:

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

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

Πριν αφήσουμε τα γράμματα θα τονίσουμε ότι

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

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

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

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

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

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

Παράδειγμα

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

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

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

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

---

[1] Οι περισσότερες γλώσσες προγραμματισμού δεν τα ξεχωρίζουν.

1.1 Το Αλφάβητο (Σύνολο Χαρακτήρων) της C++

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

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

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

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

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

Στο Παράρτημα D υπάρχει πίνακας αυτών των αντικαταστάσεων.

1.2 Το Πρώτο Πρόγραμμα

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

```
#include <iostream>

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

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

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

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

1.2.1 Να «Διώξουμε» το "`std::`"!

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

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

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

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

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

Πρόσεξε τώρα έναν άλλον τρόπο γραφής του προγράμματος:

```
#include <iostream>

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

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

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

Υπάρχει και άλλος τρόπος, πιο «τεμπέλικος»:

```
#include <iostream>

using namespace std;

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

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

Εδώ θα χρησιμοποιούμε τον τελευταίο τρόπο.

Για ονοματοχώρους θα μιλήσουμε εκτενώς αργότερα.

1.3 Πρόγραμμα

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

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

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

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

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

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

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

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

Αποτέλεσμα:

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

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

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

ή ως εξής:

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

   cout << " ΔΟΥΛΕΥΕΙ!" << endl;

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

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

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

Και εκείνο το

```
#include <iostream>
```

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

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

1.4 * Η Οδηγία "include"

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

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

ενώ στο **prwto.cpp** φύλαξε τα:

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

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

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

Βγάζεις κάποιο συμπέρασμα για την include; Αποτέλεσμα της

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

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

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

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

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

#include <*όνομα αρχείου***>** ή

#include "*όνομα αρχείου***"**

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

Και ποια η διαφορά της πρώτης μορφής από τη δεύτερη;

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

---

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

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

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

1.5 Ορμαθοί Χαρακτήρων

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

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

Αυτά είναι παραδείγματα ορμαθών χαρακτήρων.

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

Παραδείγματα

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

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

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

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

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

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

Παραδείγματα

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

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

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

Ας δούμε ένα άλλο παράδειγμα όπου χρησιμοποιούμε ορμαθούς χαρακτήρων.

Παράδειγμα

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

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

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

Αυτό το πρόγραμμα θα δώσει:

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

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

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

θα γράψει:

```
abc
def
```

1.5.1 Ορμαθοί Μεγάλου Μήκους

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

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

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

θα γράψει:

```
abcdef
```

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

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

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

θα δώσει:

```
abcdef
```

1.6 Έξοδος Αποτελεσμάτων

Η πρώτη εντολή που είδαμε ήταν η εντολή

`cout << "`*κείμενο*`" << endl`

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

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

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

```
cout << endl;
```

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

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

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

Παράδειγμα

Η εντολή:

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

γράφει:

```
 ΜΗΛΟΠΙΤΑ
```

ενώ η εντολή:

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

γράφει:

```
 ABCDEF
```

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

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

θα δώσουν:

```
abcdef
```

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

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

Είναι *σχεδόν* το ίδιο πράγμα.

1.7 Αριθμητικές Πραγματικές Σταθερές

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

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

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

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

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

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

αντί να γράψουν:

3000000000 γράφουν 3×10^{9}

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

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

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

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

Τα παραπάνω παραδείγματα γράφονται στη C++ ως εξής:

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

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

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

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

Δηλαδή:

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

```
1234.5678      .1357      2345.
```

Μια κλασματική σταθερά *είναι* σταθερά κινητής υποδιαστολής.

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

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

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

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

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

Εδώ θα πρέπει να προλάβουμε μια πολύ συνηθισμένη παρανόηση:

♦ ***Το "e" δεν σημειώνει πράξη!***

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

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

```
5.36e-8e2
```

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

Τέλος, για τις σταθερές κινητής υποδιαστολής ισχύει το εξής:

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

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

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

ενώ τα παρακάτω *δεν είναι:*

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

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

Παράδειγμα

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

Αποτέλεσμα:

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

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

Από το τελευταίο παράδειγμα καταλαβαίνουμε ότι:

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

1.7.1 Εσωτερική Παράσταση Πραγματικών Τιμών

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

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

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

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

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

Παράδειγμα

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

Αποτέλεσμα:

```
12.3456
12.3457
```

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

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

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

```
12.3456
12.345678901234567350
```

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

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

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

1.8 Ακέραιες Σταθερές

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

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

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

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

Παραδείγματα

Τα:

```
12  0  417  375  2048
```

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

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

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

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

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

δίνει:

```
12
```

1.8.1 Οκταδικοί Ακέραιοι

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

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

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

θα δώσει:

```
11  9
```

1.8.2 Δεκαεξαδικοί Ακέραιοι

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

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

θα δώσει:

```
10 255 41260
```

1.9 Πράξεις – Αριθμητική Παράσταση

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

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

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

---

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

Πλαίσιο 1.1

Οι Αριθμητικοί Τελεστές της C++

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

Υπολογισμός Αριθμητικής Παράστασης στη C++

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

2. Μετά γίνονται οι πράξεις *, /, %,

3. Στη συνέχεια γίνονται οι πράξεις +, -.

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

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

Παράδειγμα

Το πρόγραμμα:

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

δίνει:

```
0   0.5
```

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

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

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

όπου Δ ο διαιρετέος και δ ο διαιρέτης. Αλλιώς:

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

Έτσι,

η πράξη: `5 / 2` δίνει 2

η πράξη: `5 % 2` δίνει 1

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

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

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

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

θα δώσει:

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

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

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

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

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

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

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

Το ίδιο γίνεται και στη C++:

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

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

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

Η παράσταση:

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

θα γραφεί στη C++:

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

Η διαφορά είναι αμελητέα. Μόνο πρόσεχε!

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

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

Παράδειγμα

Έστω ότι θέλουμε να υπολογίσουμε το:

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

Αν γράψουμε:

```
3.45 / 5 * 3.14159
```

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

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

Χρησιμοποιώντας παρενθέσεις γράφουμε το σωστό:

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

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

1.10 Οι Συναρτήσεις της C++

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

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

Κανένα πρόβλημα! Νάτες:

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

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

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

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

  Ο υπολογισμός γίνεται ως εξής:

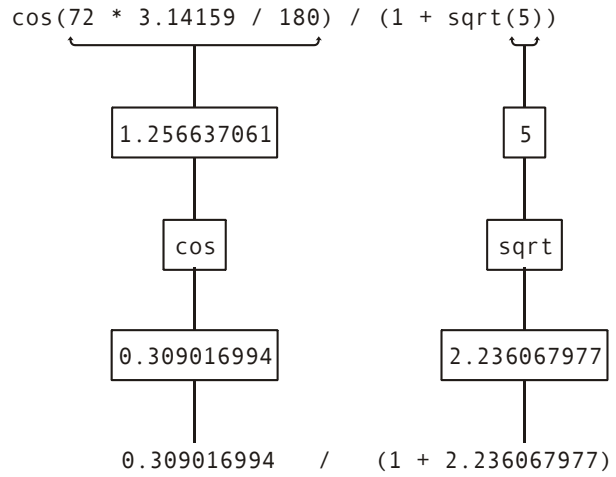
- Πρώτα υπολογίζεται η τιμή του ορίσματος,
- στη συνέχεια η τιμή της συνάρτησης και
- τέλος η τιμή αντικαθίσταται στη θέση της κλήσης συνάρτησης για να γίνουν οι άλλες πράξεις.

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

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

```
#include <cmath>
```

Ας δούμε ένα παράδειγμα με μερικές από αυτές:



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

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

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

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

θα δώσει:

```
1.41421    6
```

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

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

που θα σου δώσει:

```
2.71828
```

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

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

θα σου δώσει:

```
0    1
```

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

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

αποτέλεσμα:

```
0.866025
```

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

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

αποτέλεσμα:

```
0.866025
```

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

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

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

θα δώσει:

```
0.785398  π = 3.14159
```

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

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

αποτέλεσμα:

```
1000    1.41421
```

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

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

αποτέλεσμα:

```
7    7
```

1.10.1 "cmath" ή "math.h" ;

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

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

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

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

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

```
#include <iostream>
```

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

```
#include <cmath>
```

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

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

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

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

1.11 Έλεγχος Εκτύπωσης

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

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

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

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

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

θα δώσουν:

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

και ας δούμε γιατί. Ζητούμε πλάτος πεδίου

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

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

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

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

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

θα δώσουν:

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

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

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

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

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

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

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

θα δώσουν:

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

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

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

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

Αν τώρα δώσεις:

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

οι εντολές:

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

θα δώσουν:

```
1234.50000000  123456.78000000  123456789.01230000
```

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

Αν θέλεις να επιστρέψεις στον αυτόματο καθορισμό παράστασης δώσε:

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

1.12 Κληρονομιά από τη C: *printf()*

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

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

Ας ξεκινήσουμε με ένα παράδειγμα χρήσης της *printf*:

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

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

που δίνει:

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

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

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

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

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

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

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

θα δώσει:

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

ενώ η:

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

δίνει:

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

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

1.13 Σχόλια

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

Για την C++, σχόλιο είναι οτιδήποτε υπάρχει:

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

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

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

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

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

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

1.14 Προβλήματα;

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

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

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

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

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

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

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

Το διορθώνουμε και ξαναπροσπαθούμε. 2ο λάθος:

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

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

```
#include <cmath>
```

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

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

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

Το διορθώνουμε και όλα πάνε καλά. Να το πρόγραμμα:

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

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

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

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

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

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

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

Ασκήσεις

Α Ομάδα

1-1 Ποια από τα παρακάτω είναι πραγματικές σταθερές:

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

1-2 Ποια από τα παρακάτω είναι ακέραιες σταθερές:

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

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

Υπολόγισε τις τιμές τους.

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

Β Ομάδα

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

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

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

(e: η βάση των φυσικών λογαρίθμων).

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

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

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

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

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

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

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

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

και παίρνουμε αποτέλεσμα:

```
2  2
```

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

κεφάλαιο

2

Μεταβλητές και Εκχωρήσεις

Ο στόχος μας σε αυτό το κεφάλαιο:

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

Προσδοκώμενα αποτελέσματα:

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

Έννοιες κλειδιά:

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

Περιεχόμενα:

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

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

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

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

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

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

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

2.1 Μεταβλητές και Τύποι

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

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

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

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

Παραδείγματα

Τα παρακάτω είναι δηλώσεις μεταβλητών:

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

Τα ίδια θα μπορούσαν να δηλωθούν και ως εξής:

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

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

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

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

Σημείωση: ▶

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

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

Παραδείγματα

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

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

ή ως εξής:

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

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

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

Αν θέλουμε, έχουμε τη δυνατότητα να

- δίνουμε αρχικές τιμές στις μεταβλητές που δηλώνουμε.

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

Όταν αρχίζει η εκτέλεση του προγράμματος:

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

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

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

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

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

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

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

2.1.1 Ονόματα (Αναγνωριστικά)

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

Όπως ήδη είπαμε:

όνομα μεταβλητής = *όνομα*;

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

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

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

Ισχύουν ακόμα και οι εξής περιορισμοί:

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

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

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

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

```
double sqrt;
```

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

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

Παραδείγματα δεκτών αναγνωριστικών είναι τα εξής:

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

Δεν είναι δεκτά όμως τα:

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

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

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

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

Υπακούοντας στους παραπάνω κανόνες εφάρμοσε και τον παρακάτω:

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

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

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

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

---

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

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

2.2 Εκχώρηση – Μεταβλητές στις Παραστάσεις

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

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

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

Οι δουλειές που κάνουμε με μια μεταβλητή είναι:

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

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

```
double x;
```

μπορούμε να δώσουμε την εντολή:

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

Τι σημαίνει;

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

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

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

όνομα μεταβλητής "=" *παράσταση*

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

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

Παραδείγματα

Οι παρακάτω είναι εντολές εκχώρησης:

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

Οι παρακάτω *δεν είναι εντολές εκχώρησης*:

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

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

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

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

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

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

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

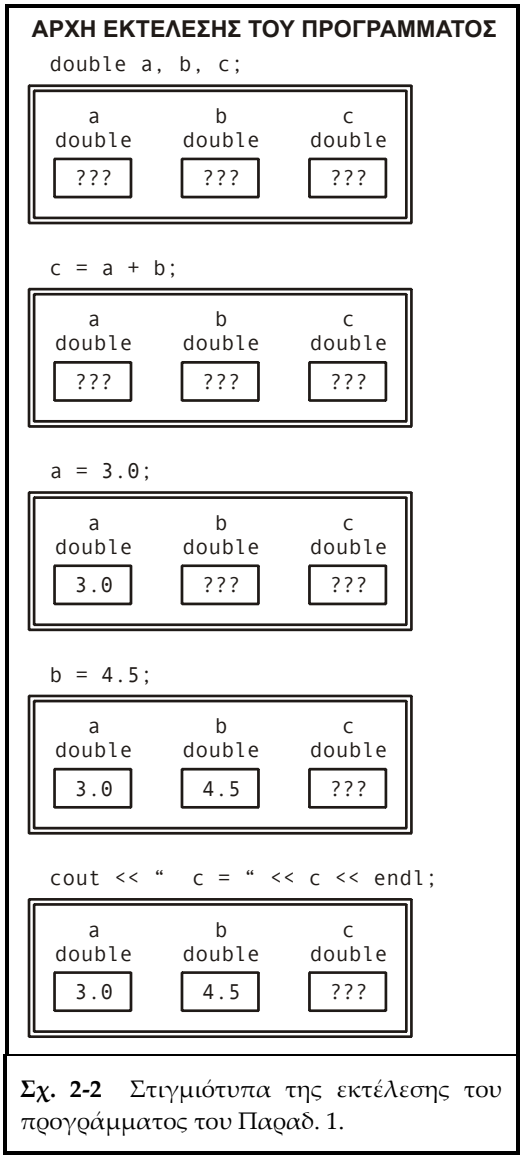
```
#include <iostream>
using namespace std;
int main()
{
   int k;

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

που δίνει:

```
 k = 3
```

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

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



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

Παράδειγμα 1

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

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

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

Αποτέλεσμα:

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

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

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

Και πώς εκτελείται;

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

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

Τα διδάγματα από τα παραπάνω είναι τα εξής:

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

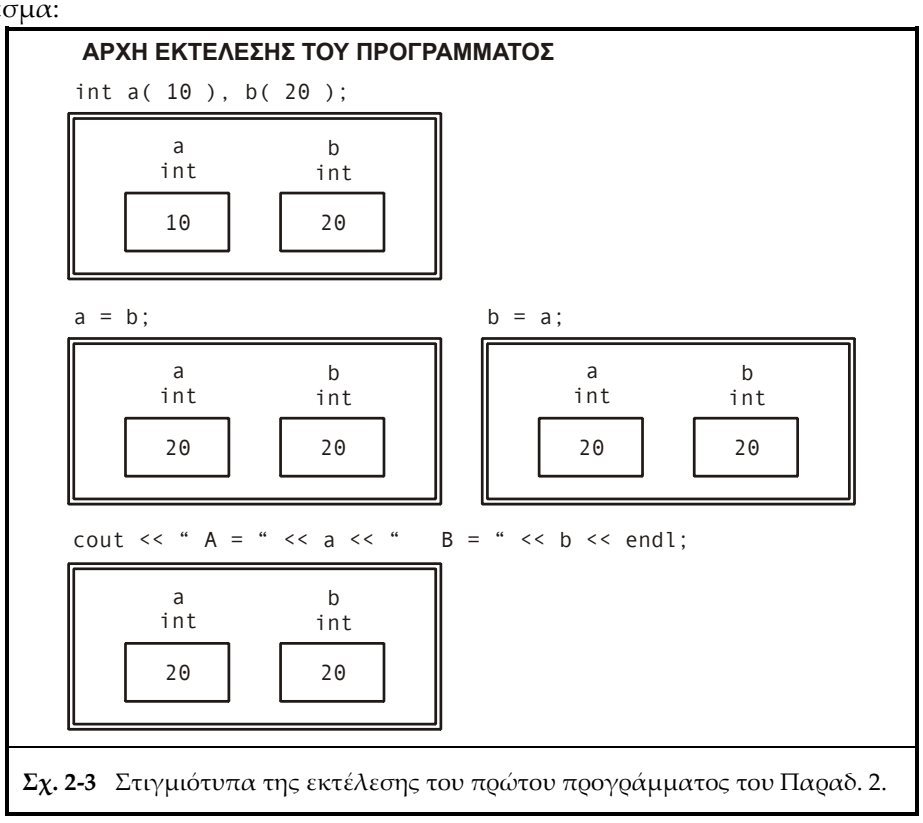
Παράδειγμα 2

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

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

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

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

Αποτέλεσμα:



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

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

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

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

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

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

που μας δίνει:

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

Το δίδαγμα και από εδώ είναι το εξής:

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

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

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

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

2.3 Εισαγωγή Στοιχείων

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

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

`cin >>` *λίστα ορισμάτων*`;`

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

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

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

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

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

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

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

  Το τι κάνει η εντολή εισόδου φαίνεται στο παρακάτω

Παράδειγμα 1

Γράφουμε το πρόγραμμα:

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

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

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

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

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

```
17 -375 145.78 310000
```

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

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

Το ίδιο αποτέλεσμα παίρνουμε και με τα:

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

Αν όμως πληκτρολογήσουμε:

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

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

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

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

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

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

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

`17  -375  145.78  31e4  123`

Αποτέλεσμα:

`17 -375 145.78 310000`

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

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

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

Παράδειγμα 2

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

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

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

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

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

Πληκτρολογούμε τα εξής:

`5  7`

και ο ΗΥ απαντάει:

`(5) + (7) = 12`

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

`57  -17`

να πάρουμε:

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

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

Αυτό ήταν εύκολο. Να δούμε κάτι πιο δύσκολο.

Παράδειγμα 3

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

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

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

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

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

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

```
Δώσε τρεις πραγματικούς αριθμούς
_
```

Πληκτρολογούμε:

```
               1  3  -10
```

και παίρνουμε:

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

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

```
 2   3   4
```

και παίρνουμε (gcc - Dev C++):

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

ή (Borland C++ 5.5):

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

sqrt: DOMAIN error

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

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

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

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

2.4 Σταθερές με Ονόματα

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

Μπορούμε βέβαια να γράφουμε

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

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

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

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

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

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

και να γράφουμε:

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

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

Παρομοίως για το π μπορούμε να ορίσουμε:

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

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

2.5 Οι Αριθμητικοί Τύποι της C++

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

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

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

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

Αν λοιπόν έχεις δηλώσει, όπως είδαμε παραπάνω:

```
   int number;
```

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

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

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

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

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

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

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

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

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

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

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

Αποτέλεσμα:

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

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

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

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

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

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

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

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

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

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

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

που μας δίνουν:

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

Η δήλωση:

```
double  length;
```

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

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

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

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

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

Παραδείγματα

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

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

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

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

Αλλά:

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

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

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

2.6 Έξω από τα Όρια

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

```
double d;
```

και στη συνέχεια έχουμε:

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

Όταν εκτελείται η `cin >> d` δίνουμε:

```
1e50000
```

Αποτέλεσμα:

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

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

Παρόμοια συμβαίνουν και με τους άλλους τύπους κινητής υποδιαστολής.

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

2.7 Πότε Λύνεται το Πρόβλημα – Δύο Παραδείγματα

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

Το πρόβλημα:

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

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

β) η ταχύτητά του τη στιγμή της πρόσκρουσης.

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

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

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

για τη θέση και:

[8] Αν αυτό σου λέει κάτι.

$$v = v_0 + at$$

για την ταχύτητα, όπου:

a : η επιτάχυνση,

v_0: η αρχική ταχύτητα,

z_0: η αρχική θέση.

Στην περίπτωσή μας:

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

και οι παραπάνω εξισώσεις γίνονται:

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

για τη θέση και:

$$v = -gt$$

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

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

παίρνουμε τον χρόνο πτώσης:

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

Η ταχύτητα την στιγμή της πρόσκρουσης είναι:

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

Αν λοιπόν δηλώσουμε:

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

έχουμε τις προδιαγραφές:

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

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

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

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

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

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

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

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

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

Η «*Υπολόγισε τα t_P, v_P*» θα γίνει:

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

σύμφωνα με αυτά που είπαμε παραπάνω.

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

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

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

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

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

Να λοιπόν ολόκληρο το πρόγραμμα:

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

και ένα παράδειγμα εκτέλεσης. Αρχικώς το πρόγραμμα ζητάει:

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

Απαντούμε:

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

και παίρνουμε:

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

Ας δούμε άλλο ένα

Παράδειγμα

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

α) το εμβαδό της βάσης σε m^2,

β) τον όγκο του κυλίνδρου σε *lt*.

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

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

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

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

Μετά από αυτές τις παρατηρήσεις ξαναγράφουμε το σχέδιό μας:

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

Στο πρόγραμμα θα χρειαστούμε μεταβλητές για:

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

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

Να πώς θα μπορούσε να είναι το πρόγραμμα:

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

Όταν αρχίσει η εκτέλεσή του, βλέπουμε στην οθόνη μας:

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

Σε απάντηση πληκτρολογούμε:

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

και στη συνέχεια βλέπουμε:

```
Δώσε το ύψος σε μέτρα: _
```

απαντούμε:

```
Δώσε το ύψος σε μέτρα: 1.8
```

και παίρνουμε το αποτέλεσμα:

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

Δες όμως και ένα άλλο παράδειγμα εκτέλεσης:

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

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

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

2.8 Τα Χαρακτηριστικά της Μεταβλητής στη C++

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

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

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

θα μας δώσει τον πίνακα του Σχ. 2.1.

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

2.8.1 Ο Τύπος – Ο Τελεστής "typeid"

Αν δώσεις:

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

θα πάρεις[9]:

```
double  int
```

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

Γενικώς, μπορείς να δώσεις:

typeid(*παράσταση* **).name()**

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

Να τώρα *Η λύση της άσκ. 1-8*: Η

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

μας δίνει την απάντηση:

```
double  int
```

2.8.2 Το Μέγεθος – Ο Τελεστής "sizeof"

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

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

και παίρνουμε:

```
4  8
```

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

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

Γενικώς, μπορείς να γράφεις:

"`sizeof`", *όνομα μεταβλητής* ή

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

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

Αν δώσεις:

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

θα πάρεις[10]:

```
8  4
8  4
```

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

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

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

θα πάρεις απάντηση:

```
0x343f2744  0
```

ή κάτι παρόμοιο. Η δεύτερη τιμή (0) είναι η τιμή της *number*, όπως την περιμένουμε. Η πρώτη είναι μια ακέραιη τιμή –γραμμένη στο *δεκαεξαδικό* σύστημα– και είναι η διεύθυνση, στη μνήμη, όπου αρχίζει η μεταβλητή *number*. Δοκίμασε και στον δικό σου υπολογιστή, αλλά μην περιμένεις να πάρεις την ίδια τιμή!

Έτσι:

- γράφοντας `number` παίρνουμε αυτό που μας ενδιαφέρει, δηλ. την τιμή της μεταβλητής `number`, ενώ
- γράφοντας `&number` παίρνουμε τη διεύθυνση μιας θέσης μνήμης όπου υπάρχει η πληροφορία που θέλουμε· παίρνουμε δηλαδή μια *παραπομπή* προς αυτό που μας ενδιαφέρει.

Λέμε λοιπόν ότι η `&number` είναι μια **παραπέμπουσα** ή **αναφερόμενη** (referencing) τιμή –αφού παραπέμπει ή αναφέρεται σε κάτι– ή τιμή-**βέλος** (pointer) –αφού κατευθύνεται προς (δείχνει) κάτι. Όπως θα δούμε αργότερα, στον προγραμματισμό με τη C++, τα βέλη χρησιμοποιούνται πάρα πολύ.

Ας πούμε ότι έχουμε μια τιμή-βέλος *p*, δηλαδή μια διεύθυνση· πώς μπορούμε να δούμε την τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η *p*; Ή, με άλλα λόγια, ποια είναι η *αντίστροφη πράξη* της "`&`"; Η C++ τη συμβολίζει με "`*`": η τιμή που είναι απόθηκευμενη στη θέση που μας δείχνει η *p* παριστάνεται με "`*p`". Αν, λοιπόν, πάρουμε τη "`*(&number)`" είναι σαν να παίρνουμε τη "`number`". Πράγματι, η:

```
cout << &number << "  " << number
     << "  " << *(&number) << endl;
```

θα δώσει:

```
0x343f2744  0  0
```

Δες και αυτό:

```
*(&number) = 4;
cout << &number << "  " << number
     << "  " << *(&number) << endl;
```

[10] Αυτό μπορείς να το θεωρήσεις και ως λύση στην άσκ. 1-8.

Αποτέλεσμα:

```
0x343f2744  4  4
```

«Μα, αριστερά του "=" δεν υπάρχει μεταβλητή!» Υπάρχει! Αφού είπαμε ότι «*Αν πάρουμε την* `*(&number)` *είναι σαν να παίρνουμε τη* `number`.»

Λέμε ότι ο τελεστής "`*`" **απο-παραπέμπει** (dereferences) την τιμή-βέλος στην οποία δρα.

Εδώ όμως μπορεί να υπάρχουν αντιρρήσεις: «ο τελεστής "`*`" ξέρουμε ότι κάνει πολλαπλασιασμό! Τι αποπαραπομπές και κουραφέξαλα μας λες;» Ναι, *ο "`*`" κάνει πολλαπλασιασμό όταν δρα σε δύο αριθμητικές τιμές· κάνει αποπαραπομπή όταν δρα σε μια τιμή-βέλος.*

2.8.4 Πώς Παίρνουμε τον Πίνακα

Και τώρα ας έρθουμε να γράψουμε το πρόγραμμα που θα βγάζει τον πίνακα του Σχ. 2-1 και μάλιστα με μια στήλη παραπάνω όπου θα γράφεται το μέγεθος σε ψηφιολέξεις. Εδώ θα γράψουμε τις εντολές που βγάζουν τις δύο πρώτες γραμμές και εσύ θα το συμπληρώσεις για να βγάλει τις υπόλοιπες.

Φυσικά, ξεκινούμε με τις δηλώσεις:

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

Στη αρχή θα πρέπει να τυπώσει το όνομα σε 8 θέσεις και στη συνέχεια τη διεύθυνση:

```
   cout.width( 8 );
   cout << "integer" << "  " << &integer << "  ";
```

Τώρα σειρά έχει ο τύπος, σε 6 θέσεις, και μετά το μέγεθος:

```
   cout.width( 6 );
   cout << typeid(integer).name() << "  "
        << (sizeof integer) << "  ";
```

Τέλος βγάζουμε και την τιμή:

```
   cout << integer << endl;
```

Να λοιπόν το πρόγραμμα:

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   double pi, e, distance( 0.0 ), length( 1.0 );
   int    i, counter( 0 ), j, number( 0 ), integer( 375 );

   cout << " όνομα\tδιεύθυνση\tτύπος\tμεγ\tτιμή" <<endl;
   cout.width( 8 );
   cout << "counter" << '\t' << &counter << '\t';
   cout.width( 6 );
   cout << typeid(counter).name() << '\t'
        << (sizeof counter) << '\t';
   cout << counter << endl;

   cout.width(8);
   cout << "distance" << '\t' << &distance << '\t';
   cout.width(6);
   cout << typeid(distance).name() << '\t'
        << (sizeof distance) << '\t';
   cout << distance << endl;
// εσύ γράφεις τα υπόλοιπα
}
```

που (συμπληρωμένο θα) δίνει:

```
όνομα  διεύθυνση    τύπος  μεγ    τιμή
 counter     0x12FF64        int 4      0
distance     0x12FF74     double 8      0
```

```
       e      0x12FF7C      double 8      2.12414e-314
       i      0x12FF68         int 4      1245072
 integer      0x12FF58         int 4      375
       j      0x12FF60         int 4      1
  length      0x12FF6C      double 8      1
  number      0x12FF5C         int 4      0
      pi      0x12FF84      double 8      2.122e-314
```

Παρατηρήσεις: ▶

- Δεν σου αρέσει ως πίνακας, ε; Πέρνα τον στο Excel ή σε πίνακα του Word και θα γίνει πανέμορφος.
- Οι τιμές των *e*, *i*, *j* και *pi* δεν πρέπει να σε εκπλήσσουν: αυτές οι μεταβλητές είναι αόριστες. Έτσι, ας πούμε για την *e*, ο ΗΥ «βλέπει» τα δυαδικά ψηφία στην αντίστοιχη διεύθυνση και τα «ερμηνεύει» ως τιμή τύπου `double`. ◀

2.9 Αλλαγή Τύπου

Για κάθε τύπο *T* της C++ υπάρχει μια συνάρτηση, με όνομα

`static_cast<`*T*`>`

που μετατρέπει τιμές άλλων τύπων σε τιμή τύπου *T* (αν αυτή η μετατροπή είναι δυνατή). Εδώ θα ασχοληθούμε με μετατροπές αριθμητικών τύπων. Ας δούμε ένα παράδειγμα:

Παράδειγμα

Η εντολή:

```
cout << static_cast<int>(9.916) << "  "
     << static_cast<int>(-9.916) << endl;
```

θα δώσει:

```
9  -9
```

Δηλαδή: οι τιμές "`9.916`" και "`-9.916`" μετατράπηκαν σε τιμές τύπου `int` από τη συνάρτηση `static_cast<int>`. Αυτό έγινε με *αποκοπή του κλασματικού μέρους*.

Η εντολή:

```
cout << static_cast<double>(901234567L) << "  "
     << static_cast<float>(901234567L) << endl;
```

θα δώσει:

```
9.01235e+08  9.01235e+08
```

Δηλαδή: η τιμή `901234567L` μετατράπηκε

- σε μια τιμή τύπου `double` από τη `static_cast<double>` και
- σε μια τιμή τύπου `float` από τη `static_cast<float>`.

Αλλά, η ακέραιη τιμή είχε 9 ψηφία ενώ από τη μετατροπή μας έμειναν 6. Μήπως υπάρχουν τα ψηφία αλλά δεν γράφονται; Ας δοκιμάσουμε να αλλάξουμε την ακρίβεια:

```
cout.precision(10);
cout << static_cast<double>(901234567L) << "  "
     << static_cast<float>(901234567L) << endl;
```

Αποτέλεσμα:

```
901234567  901234560
```

Βλέπουμε δηλαδή ότι *από τη μετατροπή* `long` *σε* `float` *μπορεί να έχουμε απώλεια σημαντικών ψηφίων*.

Να δούμε τώρα μια άλλη περίπτωση: Ας υποθέσουμε ότι έχουμε τύπο `int` με μέγιστη τιμή 2147483647· τι θα γίνει αν ζητήσω την

`static_cast<int>( 2345678901.23 )`

Η C++ μας λέει ότι το αποτέλεσμα είναι αόριστο. Γενικώς: *αν έχεις αριθμητικό τύπο T που περιλαμβάνει τιμές από* min_T *μέχρι* max_T *μπορείς να χρησιμοποιείς την* `static_cast<T>(x)` *μόνον αν*

$$min_T \le \texttt{static\_cast<}T\texttt{>}(x) \le max_T$$

Η αλλαγή παράστασης μιας τιμής από έναν τύπο σε έναν άλλο ονομάζεται (στατική) **τυποθεώρηση** (typecasting ή casting)[11].

Εδώ θα πρέπει να τονίσουμε ότι η τιμή που έχει μια μεταβλητή δεν μεταβάλλεται από τη στατική τυποθεώρησή της. Π.χ. οι εντολές:

```
double x( 123.457 );
int    i;
i = static_cast<int>( x );
cout << "  x = " << x << "    i = " << i << endl;
```

δίνουν:

```
x = 123.457    i = 123
```

Αντί για `i = static_cast<int>( x )` μπορούμε να γράψουμε:

```
i = int( x );
```

και να πάρουμε το ίδιο αποτέλεσμα. Γενικώς, το να γράφουμε:

$T(v)$ αντί για `static_cast<`T`>(`v`)`

είναι πολύ συνηθισμένο αλλά όχι κατ' ανάγκη καλύτερο.

Πρόσεξε τώρα μια (πολύ συχνή) χρήση της τυποθεώρησης: Από όσα ξέρουμε, μετά τη δήλωση:

```
int k( 10 ), n( 3 );
```

η εντολή:

```
cout << "  k/n = " << k/n << endl;
```

θα δώσει:

```
k/n = 3
```

Πώς μπορούμε να πάρουμε το ακριβές αποτέλεσμα; Έτσι:

```
cout << "  k/n = " << static_cast<double>(k)/n << endl;
```

ή έτσι:

```
cout << "  k/n = " << double(k)/n << endl;
```

που δίνουν:

```
k/n = 3.33333
```

2.9.1 Η Τυποθεώρηση στη C

Η C++ έχει κληρονομήσει από τη C και έναν άλλο τρόπο γραφής της τυποθεώρησης· γράφουμε:

$(T)\,v$ που είναι ισοδύναμο[12] με `static_cast<`T`>(`v`)`

Έτσι, μπορεί να δεις γραμμένο

```
(unsigned long int) -1234567890L
```

αντί για:

```
static_cast<unsigned long int>(-1234567890L)
```

---

[11] Υπάρχουν και άλλα είδη τυποθεώρησης, που θα τα δούμε αργότερα.

[12] Ε, όχι ακριβώς· ο συμβολισμός αυτός καλύπτει και άλλες περιπτώσεις τυποθεώρησης, που θα μάθουμε αργότερα.

2.10 * Οι «Συντομογραφίες» της Εκχώρησης

Η C++ μας δίνει μερικές «συντομογραφίες» πολύ συνηθισμένων εκχωρήσεων. Πολύ συχνά στα προγράμματά μας θέλουμε να γράψουμε: αύξησε το *x* κατά *b*· αυτό το γράφουμε, σύμφωνα με όσα ξέρουμε:

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

δηλαδή: πάρε την τιμή της *x* και την τιμή της *b*, πρόσθεσέ τις και το αποτέλεσμα να το αποθηκεύσεις ως νέα τιμή της *x*. Η C++ μας δίνει τη δυνατότητα να το γράψουμε ως:

```
   x += b;
```

Παρόμοιες συντομογραφίες υπάρχουν και για τις άλλες πράξεις:

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

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

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

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

Τέλος, υπάρχει και μια παραλλαγή της τελευταίας συντομογραφίας: οι `x++` και `x--` αυξάνουν/μειώνουν την τιμή της *x* κατά 1. Τη διαφορά τους από τις `++x` και `--x` θα τη μάθουμε αργότερα.[13]

Στο παρακάτω παράδειγμα μπορείς να δεις όλα τα παραπάνω:

```
#include <iostream>
using namespace std;
int main()
{
   int x, y;
   int a = 1, b = 2;

   x = 5; x += a+b; cout << x << "  ";
   x = 5; x -= a+b; cout << x << endl;
   x = 5; x *= a+b; cout << x << endl;
   x = 5; x /= a+b; cout << x << "  ";
   x = 5; x %= a+b; cout << x << endl;

   x = 5;  ++x;   cout << x << "  ";
   x = 5;  --x;   cout << x << endl;
   x = 5;  x++;   cout << x << "  ";
   x = 5;  x--;   cout << x << endl;
}
```

Αποτέλεσμα:

```
8  2
15
1  2
6  4
6  4
```

Θα αρχίσουμε να χρησιμοποιούμε τις συντομογραφίες αυτές στο Β' Μέρος του βιβλίου.

---

[13] Τώρα προσπάθησε να καταλάβεις τη σχέση που έχει η C++ με τη «μητρική» της γλώσσα C!

2.11 * Υπολογισμός Παράστασης

Στον πίνακα του Παρ. Ε βλέπεις τα χαρακτηριστικά των πράξεων της C++. Ας δούμε αυτές που έχουμε μάθει μέχρι τώρα:

- Πρώτα εκτελούνται υπολογισμοί μέσα σε παρενθέσεις (προτ. 0).
- Στη συνέχεια υπολογίζονται οι κλήσεις συναρτήσεων (προτ. 2).
- Μετά γίνονται πράξεις με τα πρόσημα (προτ. 3) με προσεταιριστικότητα από δεξιά προς τα αριστερά. Δηλαδή; Στη C++ μπορείς να γράψεις:

  `+ - - +12` ή `- + - -x`

  που υπολογίζονται ως:

  `+(-(-(+12)))` και `-(+(-(-x)))`

  αντιστοίχως.
- Μετά γίνονται πολλαπλασιασμοί, διαιρέσεις και υπολογισμοί υπολοίπου (προτ. 5) με προσεταιριστικότητα από αριστερά προς τα δεξιά. Αυτό σημαίνει ότι η παράσταση: `2*x/y*z/5` υπολογίζεται ως: `(((2*x)/y)*z)/5`. Αλλά προσοχή: *δεν σημαίνει* ότι αν έχεις την `(x*y)/(w/u)` θα γίνει πρώτα ο πολλαπλασιασμός `x*y` και μετά η διαίρεση `w/u`.
- Στη συνέχεια γίνονται προσθέσεις και αφαιρέσεις (προτ. 6) από αριστερά προς τα δεξιά, με την ίδια έννοια όπως παραπάνω.

Όταν υπολογίζεται μια αριθμητική παράσταση γίνονται και ορισμένες μετατροπές τύπων, που θα τις ονομάζουμε *Συνήθεις Αριθμητικές Μετατροπές* (ΣΑΜ). Ας πούμε ότι έχουμε κάποια πράξη *a θ b*. Τότε:

1. Αν κάποια από τις τιμές *a*, *b* είναι τύπου `long double`, τότε μετατρέπεται στον τύπο `long double` και η άλλη τιμή.
2. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `double`, τότε μετατρέπεται στον τύπο `double` και η άλλη τιμή.
3. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `float`, τότε μετατρέπεται στον τύπο `float` και η άλλη τιμή.
4. Αλλιώς, τα *a*, *b* (είναι ακέραιου τύπου) υφίστανται *προώθηση ακεραίων*, δηλαδή: αν κάποια (ή και οι δύο) από τις τιμές *a*, *b* είναι «μικρού ακέραιου τύπου» (`char`, `short int`, `signed` ή `unsigned`), αυτή μετατρέπεται σε τύπο `int` ή `unsigned int` (αν δεν μπορεί να παρασταθεί στον `int`) και στη συνέχεια

   αν κάποια από τις τιμές *a*, *b* είναι τύπου `unsigned long`, τότε

   μετατρέπεται στον τύπο `unsigned long` και η άλλη τιμή.
5. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `long int` και η άλλη `unsigned int`, τότε αν κάθε τιμή `unsigned int` μπορεί να παρασταθεί σε `long int`

   η τιμή `unsigned int` μετατρέπεται στον τύπο `long int`

   αλλιώς

   και οι δύο μετατρέπονται σε `unsigned long int`.
6. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `long`, τότε μετατρέπεται σε `long` και η άλλη τιμή.
7. Αλλιώς, αν κάποια από τις τιμές *a*, *b* είναι τύπου `unsigned`, τότε μετατρέπεται στον τύπο `unsigned` και η άλλη τιμή.
8. Αλλιώς και οι δυο τιμές είναι τύπου `int`.

Οι ΣΑΜ εφαρμόζονται όταν γίνονται οι πράξεις `+`, `-`, `*`, `/` μεταξύ αριθμητικών τιμών και καθορίζουν τον τύπο του αποτελέσματος. Ακέραιη προώθηση γίνεται στις ενικές πράξεις `+`, `-` (πρόσημα). Ο τύπος του αποτελέσματος είναι αυτός που προκύπτει από την προώθηση.

Δηλαδή, ο μεταγλωττιστής κάνει τις κατάλληλες μετατροπές τύπου ώστε να έχει τον ακριβέστερο υπολογισμό. Αλλά, στο τέλος, αυτό μπορεί να «χαλάσει» από κάποια εκχώρηση που μπορεί να ακολουθεί. Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
   short int a;

   a = SHRT_MAX + 10;
   cout << SHRT_MAX+10 << "  " << a << endl;
   cout << sizeof(SHRT_MAX+10) << "  " << sizeof(a) << endl;
}
```

που δίνει:

```
32777  -32759
4  2
```

Η πρώτη τιμή είναι ακριβής αλλά η δεύτερη όχι. Το τι έγινε το καταλαβαίνεις αν πάρεις υπόψη σου και τη δεύτερη γραμμή: Η τιμή `SHRT_MAX + 10` χρειάζεται 4 ψηφιολέξεις για να παρασταθεί αλλά η *a* έχει μόνο 2...

2.12 *scanf()*: Η Δίδυμη της *printf()*

Στη C, όπως γράφουμε με την *printf().*, διαβάζουμε με τη *scanf()*. Η αλλιώς: *ό,τι κάνει ο "`>>`" στο cin κάνει η scanf() στο stdin*, που είναι για τη C το ρεύμα από το πληκτρολόγιο προς το πρόγραμμά μας.

Ξαναγράφουμε το πρόγραμμα του Παραδ. 1 της §2.3 χρησιμοποιώντας τις *scanf()* και *printf()*:

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

   scanf( "%d %d %lf %lf", &k, &j, &x, &y );
   printf( "%d %d %f %f\n", k, j, x, y );
}
```

Η σύνταξη της *scanf* μοιάζει με αυτήν της *printf* ως προς το ότι και οι δύο παίρνουν πρώτο όρισμα έναν *ορμαθό μορφοποίησης*. Τα υπόλοιπα ορίσματα διαφέρουν ως προς τον τύπο: η *scanf* περιμένει (παραστάσεις που οι τιμές τους είναι) *βέλη*, δηλαδή *διευθύνσεις στη μνήμη*. Σε κάθε προδιαγραφή μορφοποίησης αντιστοιχεί μια διεύθυνση. Πηγαίνοντας από αριστερά προς τα δεξιά βρίσκουμε

- Την προδιαγραφή "`%d`" που αντιστοιχεί στην πρώτη διεύθυνση "`&k`". Αυτό σημαίνει: θα διαβάσεις ψηφία που σχηματίζουν μια *ακέραιη* σταθερά· θα αποθηκεύσεις την τιμή (τύπου `int`) που θα προκύψει από τη μετατροπή σε εσωτερική παράσταση στη διεύθυνση `&`*k*.
- Παρόμοια ισχύουν για τη δεύτερη "`%d`" που αντιστοιχεί στη δεύτερη διεύθυνση "`&j`".
- Η πρώτη προδιαγραφή "`%lf`" που αντιστοιχεί στην τρίτη διεύθυνση "`&x`" και σημαίνει: θα διαβάσεις ψηφία που σχηματίζουν μια *πραγματική* σταθερά· θα αποθηκεύσεις την τιμή τύπου `double` που θα προκύψει από τη μετατροπή σε εσωτερική παράσταση στη διεύθυνση `&`*x*.
- Παρόμοια ισχύουν και για τη δεύτερη "`%lf`" που αντιστοιχεί στην τέταρτη διεύθυνση "`&y`".

Οι προδιαγραφές "**%d**" είναι γενικώς για ακέραιες τιμές. Μπροστά από το '**d**' μπορεί να υπάρχει ένας **τροποποιητής** (modifier) '**h**' αν θέλουμε η εσωτερική παράσταση να είναι **short int** ή '**l**' για να είναι **long int** (χωρίς τροποποιητή για **int**).[14]

Για πραγματικές τιμές έχουμε τις προδιαγραφές "**%f**" (ή "**%e**" ή "**%E**" ή "**%f**" ή "**%g**" ή "**%G**"). Μπροστά από το '**f**' μπορεί να υπάρχει τροποποιητής '**l**' αν θέλουμε η εσωτερική παράσταση να είναι **double** ή '**L**' για να είναι **long double** (χωρίς τροποποιητή για **float**).

Παρατήρηση: ▶

Το παραπάνω πρόγραμμα δεν είναι γραμμένο σε C, αλλά πρόγραμμα C++ που χρησιμοποιεί βιβλιοθήκες της C. Ο μεταγλωττιστής της C *δεν* θα το δεχτεί.

Να τι θα δεχθεί:

```
#include <stdio.h>

int main()
{
   int    k, j;
   double x, y;

   scanf( "%d %d %lf %lf", &k, &j, &x, &y );
   printf( "%d %d %f %f\n", k, j, x, y );
}
```

Φύλαξέ το στο αρχείο **test_C_IO.c** και δώσε το στον μεταγλωττιστή της C. Δεν θα σου φέρει αντίρρηση.◀

Ας ξαναγράφουμε και το πρόγραμμα του Παραδ. 3 της §2.3 με τις *scanf* και *printf*:

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

   printf( "Δώσε τρεις πραγματικούς αριθμούς\n" );
   scanf( "%lf %lf %lf", &a, &b, &c );
   printf( " a = %f   b = %f   c = %f\n", a, b, c );
   delta = b*b - 4*a*c;
   x1 = (-b + sqrt(delta))/(2*a);
   x2 = (-b - sqrt(delta))/(2*a);
   printf( " Λύση1 = %f   Λύση2 = %f\n", x1, x2 );
}
```

Πρόσεξε ότι εδώ προτιμήσαμε να βάλουμε όλα τα σταθερά κείμενα στον ορμαθό μορφόποιησης. Δηλαδή:

- Αντι να γράψουμε

```
   printf( "%s\n", "Δώσε τρεις πραγματικούς αριθμούς" );
```

γράφουμε

```
   printf( "Δώσε τρεις πραγματικούς αριθμούς\n" );
```

- Αντι να γράψουμε

```
   printf( "%s %f %s %f %s %f\n",
           " a =", a, " b =", b, " c =", c );
```

γράφουμε

```
   printf( " a = %f   b = %f   c = %f\n", a, b, c );
```

[14] Για ακέραιες τιμές χωρίς πρόσημο έχουμε την προδιαγραφή "**%u**" με τους ίδιους τροποποιητές.

2.13 Λάθη, Λάθη ...

Αν ακολουθείς τις οδηγίες μας και περνάς τα προγράμματα και πειραματίζεσαι θα πρέπει να έχεις αγανακτήσει ήδη με τον μεταγλωττιστή σου και τα λάθη που βρίσκει. Κάνε υπομονή. Όσο προχωράς θα ελαττωθούν τα λάθη απροσεξίας καθώς και τα σοβαρότερα.

- ***Είναι βασικό να μάθεις να βρίσκεις και να διορθώνεις μόνος/η σου τα λάθη των προγραμμάτων σου.*** [15]

Μόνο έτσι θα μάθεις προγραμματισμό! (Ε, στην αρχή μπορεί να χρειαστείς και λίγη βοήθεια...)

Αλλά, ας τα πάρουμε από την αρχή:

- Το ";" είναι για τη C++ **τερματιστής εντολής** (statement terminator). Με αυτό τελειώνουν οι εντολές και οι δηλώσεις.
- Όλες οι μεταβλητές και οι σταθερές (με όνομα) πρέπει να δηλώνονται υποχρεωτικώς πριν χρησιμοποιηθούν· καλύτερα να τις δηλώνεις στην αρχή.
- Μη χρησιμοποιείς σε παραστάσεις μεταβλητές που δεν τους έχεις δώσει τιμή πιο πριν είτε με εντολή εκχώρησης, είτε με εντολή `cin >>` ....

Συνηθισμένα λάθη εκτέλεσης είναι αυτά της ανάγνωσης στοιχείων, όταν μάλιστα πρόκειται για αριθμούς:

- Αν πληκτρολογείς τιμή μεταβλητής ακέραιου τύπου, δώσε σταθερά του τύπου αυτού: πρόσημο, αν χρειάζεται, και στη συνέχεια ψηφία. Το 17.0 *δεν είναι* σταθερά ακέραιου τύπου, ούτε το 17,0 ούτε το 1.7e1. Το 17 είναι! Και, για όνομα του Θεού, μην πληκτρολογήσεις *INT_MAX*.
- Αν πληκτρολογείς τιμή μεταβλητής πραγματικού τύπου, δώσε μια σταθερά του τύπου αυτού. Η υποδιαστολή είναι "." όχι ",".

Ξανακοίταξε τώρα τα προγράμματα με τα ανεξήγητα λάθη, διόρθωσέ τα και συνέχισε την προσπάθεια. Καλή τύχη!

2.14 Τι (Πρέπει να) Έμαθες Μέχρι Τώρα

Η βασική έννοια αυτού του κεφαλαίου είναι η *μεταβλητή*, το βασικό εργαλείο για τη διαχείριση της κύριας μνήμης του ΗΥ. Πρέπει να έμαθες

- Να *δηλώνεις* μεταβλητές και –αν θέλεις– να *ορίζεις* την αρχική τιμή τους.
- Να ορίζεις/αλλάζεις την τιμή τους με την *εντολή εκχώρησης*.
- Να ορίζεις/αλλάζεις την τιμή τους με *εντολή εισόδου* από το πληκτρολόγιο.

Ακόμη, πρέπει να έμαθες μερικά πράγματα για τους βασικούς (αριθμητικούς) *τύπους* της C++ και πώς να «μετατρέπεις τον τύπο» μιας τιμής.

Τι προγράμματα μπορείς να γράψεις; Αυτά που λύνουν τις ασκήσεις της Β Ομάδας είναι χαρακτηριστικά.

---

[15] Ξέρεις πως λέγεται αυτή η δουλειά στα αγγλικά; «debugging», που θα μπορούσε να μεταφραστεί «απεντόμωση», ενώ σημαίνει «διόρθωση λαθών». Ρώτησε να μάθεις γιατί λέγεται έτσι και γιατί τα λάθη προγραμματισμού λέγονται «bugs» (ζωϋφια, κοριοί). Έχει ιστορική και χιουμοριστική αξία.

Ασκήσεις

Α Ομάδα

2-1 Δίνεται το πρόγραμμα:

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

   i = 5;  j = 7;  k = 10;
   cout << i << ' ' << (i+1) << ' ' << (i+j)
        << ' ' << (i + j*k) << endl;
   k = i + j;         cout << k << endl;
   j = j + 1;         cout << j << endl;
   i = j + 2*i + j;   cout << i << endl;
}
```

Εκτέλεσε το πρόγραμμα (*εσύ*, όχι ο υπολογιστής), όπως κάναμε στο παράδειγμα της αντιμετάθεσης.

2-2 Το ίδιο για το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{  double first, second, realResult;
   int    third, intResult;

   cin >> first >> second >> third;
   realResult = first + second + third;
   intResult = first + second - third;
   realResult = realResult - intResult;
   intResult = intResult - realResult;
   cout << realResult << intResult << endl;
}
```

Δίνεται μια γραμμή με στοιχεία εισόδου:

1.5e1 8.1 5 7.41e-5 33.0

2-3 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο μια τιμή –ας πούμε x– και θα υπολογίζει και θα τυπώνει τις τιμές των παραστάσεων για την τιμή που διαβάστηκε:

$$\frac{1}{1+\sqrt{x}} \qquad 1+e^{-x/2} \qquad x^x + x^{x/2}$$

$$\frac{\sigma\upsilon\nu x - \eta\mu x}{1+\sqrt{2x}} \qquad \frac{e^{-x/2}+e^{x/2}}{2} \qquad x^{x/3} + x^{3x}$$

όπου e η βάση των φυσικών λογαρίθμων.

2-4 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή μιας γωνίας σε μοίρες –ας πούμε x– και θα υπολογίζει και θα τυπώνει τα:

$$\eta\mu x,\ \sigma\upsilon\nu x,\ \varepsilon\varphi x,\ \sigma\varphi x,\ \tau\varepsilon\mu x,\ \sigma\tau\varepsilon\mu x$$

Β Ομάδα

2-5 Γράψε πρόγραμμα που διαβάζει από το πληκτρολόγιο δυο αριθμούς: β (θετικό πραγματικό, $\neq 1$) και x (θετικό πραγματικό) και θα υπολογίζει και θα τυπώνει τα $\log_\beta x$ και β^x.
Υπόδ.: Χρησιμοποίησε την ιδιότητα: $\log_\beta x = \ln x / \ln \beta$.

2-6 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή κάποιου αντικειμένου, χωρίς ΦΠΑ και τον συντελεστή ΦΠΑ (%). Θα υπολογίζει και θα γράφει την τιμή με ΦΠΑ.

2-7 Γράψε πρόγραμμα που θα διαβάζει τις καρτεσιανές συντεταγμένες, x, y, ενός σημείου στο επίπεδο και θα υπολογίζει και θα τυπώνει τις πολικές r, ϕ. Υπόθεσε ότι $x > 0$. Υπολόγισε τη ϕ: α) με την *atan* β) με την *atan2*.

Υπόδ.: $r = \sqrt{x^2 + y^2}$, $\phi = τοξεφ(y/x)$.

2-8 Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 2%, επί του βασικού μισθού, για κάθε χρόνο υπηρεσίας. Γράψε πρόγραμμα που θα διαβάζει τον βασικό μισθό και τα χρόνια υπηρεσίας ενός υπαλλήλου και θα υπολογίζει το χρονοεπίδομα και το συνολικό μισθό.

2-9 Αν συνδέσουμε δυο αντιστάσεις R_1 και R_2 σε σειρά, η συνολική αντίσταση είναι $R = R_1 + R_2$. Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 και θα υπολογίζει και θα τυπώνει την R.

2-10 Αν συνδέσουμε δυο αντιστάσεις R_1 και R_2 παραλλήλως, η συνολική αντίσταση είναι:

$$R = \frac{R_1 R_2}{R_1 + R_2}$$

Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 και θα υπολογίζει και θα τυπώνει την R.

2-11 Ένα κύκλωμα αποτελείται από πυκνωτή χωρητικότητας C και αντιστάτη αντίστασης R συνδεμένα παράλληλα. Στα κοινά τους άκρα συνδέουμε πηγή εναλλασσόμενης τάσης $U = U_0 ημ(2πνt)$.

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο τις τιμές των U_0 (V), $ν$ (Hz), C (F), R ($Ω$) και θα υπολογίζει και θα γράφει στην οθόνη:

- την ενεργή τιμή της τάσης $U_{εν} = U_0 / \sqrt{2}$,
- την κυκλική συχνότητά της $ω = 2πν$,
- την εμπέδηση $Z = \sqrt{\frac{1}{R^2} + (ωC)^2}$ του κυκλώματος,
- το πλάτος i_0 και την ενεργή τιμή $i_{εν}$ του ρεύματος,
- τη διαφορά φάσης μεταξύ U και i, $\phi = τοξεφ(ωCR)$.

2-12 Στην §2.5 είδαμε ένα πρόγραμμα που μας δίνει τη μέγιστη και την ελάχιστη τιμή του τύπου `int`. Αν στις εντολές του προγράμματος αντικαταστήσεις το `"INT"` με `"LONG"`, `"SHRT"`, `"CHAR"`, `"UINT"`, `"ULONG"`, `"USHRT"`, `"UCHAR"`, θα πάρεις τα στοιχεία του Πίν. 2-1 για τους τύπους `long int`, `short int`, `char`, `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char`, αντιστοίχως. Γράψε πρόγραμμα που θα βγάζει αυτά τα στοιχεία.

Στην ίδια παράγραφο είδαμε και ένα πρόγραμμα που μας δίνει τις χαρακτηριστικές τιμές του τύπου `double`. Αν αλλάξεις το `"DBL"` σε `"FLT"` και `"LDBL"` θα πάρεις τα στοιχεία των τύπων `float` και `long double` αντιστοίχως. Γράψε πρόγραμμα που θα βγάζει τα στοιχεία του Πιν. 2-2.

Γ Ομάδα

2-13 Γράψε πρόγραμμα που θα λύνει το πρόβλημα της ελεύθερης πτώσης, στην περίπτωση που η αρχική ταχύτητα δεν είναι μηδέν και έχει συνιστώσες στον κατακόρυφο και τον οριζόντιο άξονα.

κεφάλαιο

3

* Το Σωστό Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσουμε την έννοια της επαλήθευσης προγράμματος και τις βασικές σχετικές διαδικασίες.
Προσδοκώμενα αποτελέσματα:
Να μπορείς να αποδείξεις την ορθότητα απλών προγραμμάτων.
Έννοιες κλειδιά:

- *προδιαγραφές προγράμματος*
- *απόδειξη ορθότητας ή επαλήθευση προγράμματος*
- *συνθήκες επαλήθευσης*
- *αξίωμα της εκχώρησης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Γράφαμε στην §0.4:

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

Αν αυτά που λέγαμε στις §0.3 και 0.4 σου φάνηκαν αφηρημένα και «ακαταλαβίστικα» ήρθε η ώρα να τα δεις σε εφαρμογή σε συγκεκριμένα παραδείγματα, να δεις πώς γίνεται η **επαλήθευση** (verification) ή **απόδειξη ορθότητας** (correctness proof) προγράμματος.

Ακόμη, στην §2.2 γράφαμε:

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

Πολλοί συγχέουν τις **δοκιμές προγράμματος** (program testing) με την απόδειξη ορθότητας· καμία σχέση! Μπορείς να κάνεις δοκιμές στο πρόγραμμά σου για να βρεις και να διορθώσεις λάθη που ενδεχομένως έχει. Το να περάσει το πρόγραμμα επιτυχώς όλες τις δοκιμές δεν σημαίνει ότι δεν έχει άλλα λάθη. Θα πεις «Και αν το δοκιμάσω επιτυχώς για όλες τις τιμές που μπορεί να πάρουν οι μεταβλητές του;» Ε, τότε εντάξει, αλλά τι πρόγραμμα είναι αυτό;

3.1 Ξεκινώντας με τη Δήλωση

Οι συνθήκες που ισχύουν μετά από τις δηλώσεις είναι αυτές που συνάγονται από τους Πιν. 2-1 και 2-2. Ας πούμε ότι κάνουμε τις δηλώσεις:

```
int          number;
unsigned int uNumber;
double       rNumber;
```

Όπως είπαμε «*έχουμε τη σιγουριά ότι η number έχει πάντοτε ακέραιη τιμή ανάμεσα στη μεγαλύτερη και τη μικρότερη που μπορεί να παρασταθεί στον τύπο* `int`» που μπορεί να γραφεί ώς εξής:

$$(number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

Αυτή είναι η **αναλλοίωτη** (invariant) του τύπου `int`.

Παρομοίως, για τον *uNumber* θα έχουμε:

$$(uNumber \in \mathbb{N}) \;\&\&\; (0 \le uNumber \le UINT\_MAX)$$[1]

και για τον *rNumber*:

$$(rNumber \in \mathbb{Q}) \;\&\&\; (-DBL\_MAX \le rNumber \le DBL\_MAX)$$

Οι συνθήκες αυτές ισχύουν «όσο ζουν» οι μεταβλητές.

Αν δώσουμε και αρχική τιμή; Π.χ.:

```
int number( 37 );
```

τότε θα έχουμε:[2]

$$(number == 37) \;\&\&\; (number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

Εδώ έχουμε πλεονασμό: αφού *number* `==` 37 προφανώς θα έχουμε και $number \in \mathbb{Z}$ και $INT\_MIN \le number \le INT\_MAX$. Δεν πειράζει· όπως είπαμε, αυτές που έχουν σχέση με τη δήλωση ισχύουν «όσο ζουν» οι μεταβλητές ενώ η "*number* `==` 37" μετά από λίγο μπορεί να μην ισχύει.

Πρόσεξε όμως και το εξής: αν δώσεις

```
int number( 3.0/2 );
```

μπορεί να πάρεις από τον μεταγλωττιστή μια *ειδοποίηση* (warning) της μορφής «`converting to 'int' from 'double'`» αλλά η *number* θα πάρει αρχική τιμή "`1`". Μπορούμε λοιπόν να πούμε ότι μετά την

```
int number( Π );
```

θα έχουμε:

---

[1] *UINT_MAX;;;* Έλυσες την άσκ. 2-12;

[2] Τι είναι πάλι αυτό το "`==`"; Επειδή το "`=`" στη C++ έχει συγκεκριμένο νόημα, αυτό που καθορίζεται στην εκχώρηση, χρησιμοποιούμε το "`==`" για τη σύγκριση «είναι ίσο με». Αυτό θα το ξανασυζητήσουμε.

(*number* == static_cast<int>(*Π*)) &&
(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Γενικώς, μπορούμε να πούμε ότι μετά την

```
T x( Π );
```

και αν ορίζεται η static_cast<*T*>(*Π*), θα έχουμε:

(*x* == static_cast<*T*>(*Π*)) && I_T(*x*)

όπου I_T είναι η *αναλλοίωτη* του τύπου *T*, δηλαδή η συνθήκη που ισχύει για όλες τις τιμές τύπου *T*.

3.2 Εντολές Εισόδου και Εξόδου

Εδώ τα πράγματα είναι απλά:

- Μια εντολή εξόδου δεν έχει επίδραση στις τιμές των μεταβλητών. Έτσι, αν ισχύει η *P* πριν από την

```
cout << number;
```

θα ισχύει και μετά την εκτέλεσή της.

- Μετά από μια εντολή εισόδου –αν η τιμή που εισάγεται μπορεί να παρασταθεί– δεν είναι δυνατόν να ξέρουμε κάτι περισσότερο από αυτό που ξέρουμε από τις δηλώσεις. Για παράδειγμα, με τις δηλώσεις της προηγούμενης παραγράφου, μετά την:

```
cin >> number;
```

θα έχουμε:

(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Οτιδήποτε άλλο ίσχυε για τη *number* (π.χ. κάτι σαν *number* == 37) παύει να ισχύει.

3.3 Οι Σταθερές στις Αποδείξεις

Πώς χειριστήκαμε τη σταθερά *g* στο παράδειγμα του προηγούμενου κεφαλαίου. *Είχαμε τη συνθήκη g* == 9.81 *αναλλοίωτη σε ολόκληρο το πρόγραμμα*.

Αυτό ακριβώς θα κάνουμε και με όλες τις σταθερές: αν έχουμε δηλώσει:

```
const T cn( c );
```

σε ολόκληρο το πρόγραμμά[3] μας ισχύει η *cn* == *c* και η αναλλοίωτη I_T(*cn*) του τύπου *T*:

(*cn* == *c*) && I_T(*cn*)

3.4 Το Αξίωμα της Εκχώρησης

Να δούμε τώρα πώς μπορούμε να διατυπώσουμε με ακρίβεια το νόημα της εντολής εκχώρησης. Ας πούμε ότι έχουμε δηλώσει

```
double x;
int y;
```

και έχουμε στο πρόγραμμά μας τις εντολές:

```
x = 7;      // x == 7.0
y = x + 4;  // y == 11
```

Σε σχόλια έχουμε βάλει το τι ξέρουμε ότι ισχύει σε εκείνο το σημείο της εκτέλεσης σύμφωνα με αυτά που είπαμε: «υπολογίζεται η τιμή της παράστασης, η τιμή μετατρέπεται στον τύπο της μεταβλητής, η τιμή φυλάγεται ως τιμή της μεταβλητής».

[3] Για την ακρίβεια: σε ολόκληρη την *εμβέλεια* της δήλωσης, όπως θα μάθουμε αργότερα.

Πάντως, στα προγράμματα που θα δούμε στη συνέχεια, το συνηθισμένο θα είναι να μην ξέρεις την τιμή της παράστασης· θα ξέρεις όμως μερικές ιδιότητές της, π.χ.: ας υποθέσουμε ότι σε κάποιο πρόγραμμα έχουμε δηλώσει

```
double x, a, b;
```

και έχουμε την εντολή:

```
x = pow( a + b, 2 ) + 1;
```

Πριν από την εκτέλεση της εντολής έχουμε a == a_0, b == b_0, αλλά τα a_0, b_0 μας είναι άγνωστα όταν γράφουμε το πρόγραμμα· ξέρουμε όμως ότι:

$$|a_0 + b_0| \le ½$$

Τι ξέρουμε για την τιμή της x μετά την εκτέλεση της εντολής; Η τιμή Π του δεξιού μέρους όταν εκτελείται η εκχώρηση είναι:

$$(a_0 + b_0)^2 + 1 = (|a_0 + b_0|)^2 + 1$$

και αφού ξέρουμε ότι: $|a_0 + b_0| \le ½$, θα έχουμε:

$$\Pi = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Μπορούμε λοιπόν να πούμε ότι μετά την εκτέλεση της εντολής εκχώρησης θα έχουμε:

$$x = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Να λοιπόν τι μπορούμε να πούμε γενικά:

- Αν
  - η `v` είναι τύπου T και
  - πριν από την εκτέλεση της `v` = Π ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$ και
- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά

τότε

- Μετά την εκτέλεση της `v` = Π ισχύει η $P(v)$, δηλαδή η συνθήκη που προκύπτει από στην $P($ `static_cast<`T`>(`Π`)` $)$ αν αντικαταστήσουμε με τη v τη `static _cast<`T`>(` Π `)`.

Αυτό είναι το **αξίωμα της εκχώρησης**.

Παρατηρήσεις: ▶

1. Η $P($ `static_cast<`T`>(`Π`)` $)$ μπορεί να περιλαμβάνει και συνθήκες που δεν περιλαμβάνουν τη v. Αυτές δεν επηρεάζονται από την εντολή εκχώρησης, αλλά παραμένουν αναλλοίωτες από αυτήν. Στο παράδειγμά μας, συμφώνως με όσα είπαμε, θα έχουμε πριν από την εντολή:

$$(a == a_0) \;\&\&\; (b == b_0) \;\&\&\; (|a_0 + b_0| \le ½)$$

Όλα αυτά δεν επηρεάζονται από την εκτέλεση της

```
x = pow( a + b, 2 ) + 1;
```

και ισχύουν και μετά από αυτήν.

2. Ό,τι ίσχυε για τη v πριν από την εκτέλεση της "`v` = Π" δεν ισχύει μετά από αυτήν. Γυρνώντας στο παράδειγμά μας, ας πούμε ότι πριν από την εκτέλεση της `x = pow(a + b, 2) + 1` έχουμε x == 0. Μετά την εκτέλεσή της, το μόνο από τα «παλιά» που μένει να ισχύει είναι η $-DBL\_MAX \le x \le DBL\_MAX$ που απορρέει από τη δήλωση `double x`. Η x == 0 παύει να ισχύει και αντικαθίσταται από την $x = (a_0 + b_0)^2 + 1 \le {}^5/_4$. ◀

Μπορούμε να δούμε το αξίωμα της εκχώρησης και με έναν άλλο τρόπο:

- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά,
- Για να ισχύει μετά την εκτέλεση της `v` = Π η $P(v)$

θα πρέπει

- πριν από την `v` = Π να ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$, δηλαδή η συνθήκη που προκύπτει αν στην $P(v)$ βάλουμε όπου v τη `static_cast<`T`>(` Π `)`.

Όπως θα δεις στη συνέχεια, αυτόν τον τρόπο θα χρησιμοποιούμε συνήθως.

3.5 Συνθήκες "true" και "false"

Πολύ συχνά θέλουμε να γράψουμε τη συνθήκη: «για οποιαδήποτε τιμή των μεταβλητών μας». Αυτή γράφεται ως εξής: `true`. Να ένα παράδειγμα τέτοιας συνθήκης:

$(x \geq 0)$ `||` $(x < 0)$

Όπως υπάρχει συνθήκη `true`, υπάρχει και συνθήκη `false` και σημαίνει: «δεν ισχύει όποιες και αν είναι οι τιμές των μεταβλητών μας.» Π.χ.

$(x \geq 0)$ `&&` $(x < 0)$

3.6 Το Πρόγραμμα Μαζί με την Απόδειξη

Και τώρα να δούμε, με παραδείγματα, πώς χρησιμοποιούμε το αξίωμα της εκχώρησης για να αποδείξουμε την ορθότητα ενός προγράμματος. Στην πραγματικότητα θα δούμε τρεις φορές, από τρεις διαφορετικές σκοπιές, το παράδειγμα της αντιμετάθεσης. Όπως είπαμε όμως, η ορθότητα του προγράμματος αναφέρεται σε συγκεκριμένες **προδιαγραφές** (program specifications). Το πρώτο πράγμα που πρέπει να κάνουμε είναι να διατυπώσουμε αυτές τις προδιαγραφές για το πρόβλημά μας. Αυτό είναι απλό:

Προϋπόθεση: $(a == a_0)$ `&&` $(b == b_0)$

Απαίτηση: $(a == b_0)$ `&&` $(b == a_0)$

που σημαίνουν: Αν πριν από την πρώτη εντολή ισχύει η συνθήκη $(a == a_0)$ `&&` $(b == b_0)$ τότε μετά την εκτέλεση όλων των εντολών θα ισχύει: $(a == b_0)$ `&&` $(b == a_0)$.

Πώς γίνεται *πρακτικά* να πάρουμε την απόδειξη ορθότητας ενός προγράμματος; Θα δούμε πώς γίνεται αυτό από τρεις διαφορετικές «απόψεις»:

- Γράφουμε το πρόγραμμα και –*παράλληλα*– καταγράφουμε τις συνθήκες που προκύπτουν από την εκτέλεση της κάθε εντολής.
- Μας δίνεται το πρόγραμμα (ή το γράφουμε εμείς) και μετά αποδεικνύουμε ότι είναι σωστό.
- Γράφουμε το πρόγραμμα προσπαθώντας να βάζουμε τις εντολές που θα μας δώσουν αυτά που ζητούν οι προδιαγραφές.

Θα δοκιμάσουμε αυτές τις τρεις «απόψεις» στο πρόγραμμα της *αντιμετάθεσης*. Και στις τρεις περιπτώσεις, θα παρεμβάλουμε, μέσα σε σχόλια, τις συνθήκες που μας ενδιαφέρουν.

3.6.1 Απόδειξη – Πρόγραμμα Παραλλήλως

Ας δούμε αρχικά την απόδειξη που γίνεται παραλλήλως με το γράψιμο. Στην περίπτωση αυτή θα πρέπει να βάζουμε μετά από κάθε εντολή τη συνθήκη που προκύπτει από την εκτέλεσή της. Θα τη βρίσκουμε με την εξής συνταγή που βγάζουμε από το αξίωμα της εκχώρησης:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(v) ως εξής:***
  1. ***Θα αντιγράψεις την P(`static_cast<T>(Π)`).***
  2. ***Θα σβήσεις όποιαδήποτε συνθήκη περιλαμβάνει τη v. Η τιμή της v άλλαξε και ό,τι ίσχυε γι' αυτήν παύει να ισχύει.***
  3. ***Οτιδήποτε υπάρχει για την Π θα γραφεί άλλη μια φορά –με `&&`– με αντικατάσταση της Π από την v.***

Ξεκινούμε λοιπόν με:

```
// (a == a0) && (b == b0)
```

Το πρώτο που κάναμε ήταν να φυλάξουμε κάπου (στην *s*) την αρχική τιμή της *a*. Αυτό γίνεται με την εντολή εκχώρησης `s = a`. Τι θα ισχύει μετά την εκτέλεση αυτής της εντολής;

Ας βρούμε τη συνθήκη που θα ισχύει μετά την εντολή. Στην περίπτωσή μας μεταβλητή είναι η *s* και παράσταση είναι η *a*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0).
2. Αφού δεν υπάρχει τίποτε για την *s*, δεν κάνουμε διαγραφές.
3. Για την *a* υπάρχει η a == a_0· την αντιγράφουμε αντικαθιστώντας την *a* με *s*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
```

Ύστερα από αυτό, μπορούμε να βάλουμε στην *a* την τιμή της *b* (`a = b`). Πώς παίρνουμε τη συνθήκη που θα ισχύει μετά την εντολή. Εδώ, μεταβλητή είναι η *a* και παράσταση είναι η *b*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0) && (s == a_0).
2. Διαγράφουμε από αυτήν ό,τι ισχύει για την *a*: (b == b_0) && (s == a_0).
3. Για τη *b* υπάρχει η b == b_0· την αντιγράφουμε αντικαθιστώντας τη *b* με *a*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
```

Τέλος, δίνουμε στη *b* την τιμή της *a*, που έχουμε «φυλάξει» στην *s*. Ό,τι ίσχυε για τη *b* (b == b_0) παύει να ισχύει· από εδώ και πέρα για την *b* ισχύει ό,τι ίσχυε μέχρι τώρα την *s* (b == a_0):

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
   b = s
// (a == b0) && (b == a0) && (s == a0)
```

Αποδείξαμε ότι το πρόγραμμα είναι σωστό; Η συνθήκη που πήραμε τελικώς δεν είναι η απαίτηση. Αλλά... για να δούμε: Αποδείξαμε ότι αν το πρόγραμμά μας εκτελεσθεί με προϋπόθεση (a == a_0) && (b == b_0), θα καταλήξουμε στη συνθήκη: (a == b_0) && (b == a_0) && (s == a_0). Στο Παρ. Α, §Α.4, βλέπουμε ότι (P && Q) ⇒ P· στην περίπτωσή μας, αν πάρουμε ως P την (a == b_0) && (b == a_0) και ως Q την (s == a_0) έχουμε:

$$((a == b_0)\ \&\&\ (b == a_0))\ \&\&\ (s == a_0) \Rightarrow ((a == b_0)\ \&\&\ (b == a_0))$$

Αλλά ο συμπερασματικός κανόνας (*E*1), που είδαμε στην §0.4.1, λέει ότι αποδείξαμε την ορθότητα του προγράμματός μας.

3.6.2 Απόδειξη εκ των Υστέρων (από το Τέλος προς την Αρχή)

Τώρα, ας έρθουμε στη δεύτερη περίπτωση: Γράψαμε (ή μας δίνεται) το πρόγραμμα και πρέπει να αποδείξουμε ότι είναι σωστό σε σχέση με τις προδιαγραφές του. Μπορούμε να κάνουμε την απόδειξη είτε *από την αρχή προς το τέλος* είτε *από το τέλος προς την αρχή*.

Για το παράδειγμα της αντιμετάθεσης, αν κάνουμε την απόδειξη από την αρχή προς το τέλος, θα πάρουμε την απόδειξη που είδαμε παραπάνω.

Για να κάνουμε μια απόδειξη *από το τέλος προς την αρχή* θα πρέπει να μπορούμε να απαντούμε στο εξής ερώτημα: Αν ξέρω τη συνθήκη Q που ισχύει μετά την εκτέλεση της εντολής E, πώς μπορώ να βρω τη συνθήκη που πρέπει να ισχύει πριν από αυτήν; Χρησιμοποιώντας τη δεύτερη διατύπωση του αξιώματος της εκχώρησης, που είδαμε στο τέλος της §3.4, βγάζουμε την εξής συνταγή:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(`static_cast<T>(Π)`) ως εξής:***

1. *Θα αντιγράψεις την P(v).*
2. *Οπου υπάρχει η v θα την αντικαταστήσεις με τη* `static_cast<T>(Π)`.

🡒🡒🡒

Στο παράδειγμά μας θα πρέπει να αποδείξουμε:

```
// (a == a0) && (b == b0)
   s = a;  a = b;  b = s;
// (a == b0) && (b == a0)
```

Ξεκινούμε από την:

```
   b = s;
// (a == b0) && (b == a0)
```

Πώς παίρνουμε τη συνθήκη που θα ισχύει πριν από την εντολή; Εδώ, μεταβλητή είναι η *b* και παράσταση είναι η *s*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (b == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *b* την *s*: (a == b_0) && (s == a_0).

Φθάνουμε λοιπόν στο εξής:

```
   s = a;  a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Με τον ίδιο τρόπο βρίσκουμε τη συνθήκη που πρέπει να ισχύει πριν από την `a = b`. Μεταβλητή είναι η *a* και παράσταση είναι η *b*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (s == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *a* τη *b*: (b == b_0) && (s == a_0).

Έχουμε λοιπόν:

```
   s = a;
// (b == b0) && (s == a0)
   a = B;
// (a == b0) && (s == a0)
   B = s;
// (a == b0) && (b == a0)
```

Τέλος, ας βρούμε τη συνθήκη που θα πρέπει να ισχύει πριν από την `s = a` ώστε μετά από αυτήν να έχουμε (b == b_0) && (s == a_0). Μεταβλητή μας είναι η *s* και παράσταση η *a*. Άρα:

1. παίρνουμε την (b == b_0) && (s == a_0) και
2. αντικαθιστούμε το *s* με το *a* και παίρνουμε την (b == b_0) && (a == a_0).

Φθάνουμε λοιπόν στο εξής:

```
// (b == b0) && (a == a0)
   s = a;
// (b == b0) && (s == a0)
   a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Για να είναι το πρόγραμμά μας σωστό θα πρέπει: η συνθήκη (b == b_0) && (a == a_0) –που θέλουμε να ισχύει πριν από την πρώτη εντολή– να συνάγεται από την προϋπόθεση. Αλλά αυτό συμβαίνει: Επειδή η πράξη && είναι αντιμεταθετική, από την προϋπόθεση (a == a_0) && (b == b_0) παίρνουμε τη (b == b_0) && (a == a_0). Άρα το πρόγραμμά μας είναι σωστό.

🡐🡐🡐

3.6.3 Το Πρόγραμμα από τις Προδιαγραφές

Τέλος, ας έρθουμε στην τρίτη «άποψη»: Να γράψουμε το πρόγραμμα με οδηγό τις προδιαγραφές; Θα ρωτήσεις: Δηλαδή μπορούμε να το γράψουμε και αλλιώς; Όχι βέβαια·

αυτό που εννοούμε είναι να δουλέψουμε πιο συστηματικά και από τις απαιτήσεις να βρίσκουμε τις εντολές.

Θα δουλέψουμε και πάλι από το τέλος προς την αρχή.

✎✎✎

Στο παράδειγμά μας ξεκινούμε με την ερώτηση: με ποια εντολή μπορούμε να καταλήξουμε στη συνθήκη:

```
// (a == b0) && (b == a0)
```

Σύμφωνα με το αξίωμα της εκχώρησης μπορούμε να πάρουμε την b `==` a_0, αν εκτελεσθεί η εντολή `b = s` και πριν από αυτήν ισχύει η s `==` a_0. Παρόμοια, μπορούμε να φτάσουμε στην a = b_0, αν εκτελεσθεί η εντολή `a = t` και πριν από αυτήν ισχύει η t `==` b_0:

```
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Πώς μπορούμε να φτάσουμε στην (t `==` b_0) `&&` (s `==` a_0); Σύμφωνα με το αξίωμα της εκχώρησης και πάλι, μπορούμε να φτάσουμε στη συνθήκη αυτή αν εκτελεσθούν οι εντολές `s = a; t = b` και πριν από αυτές ισχύει η (a `==` a_0) `&&` (b `==` b_0), που είναι η προϋπόθεσή μας. Άρα, το παρακάτω πρόγραμμα είναι σωστό:

```
// (a == a0) && (b == b0)
   s = a;  t = b;
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Εδώ χρησιμοποιήσαμε δύο βοηθητικές μεταβλητές, αλλά αυτό δεν είναι κάτι τραγικό.

✎✎✎

3.7 Διά Ταύτα ...

Πριν κάνουμε μια σύγκριση των τριών απόψεων να προλάβουμε κάποιους που μπορεί να κάνουν ήδη απαισιόδοξες σκέψεις του εξής τύπου: «Και αν έχω ένα πρόγραμμα των εκατό γραμμών τι γίνεται; Θα έχω συνθήκες που πιάνουν σελίδες! Για να μη πούμε τι θα γίνει όταν, όπως γίνεται στην πραγματικότητα, έχουμε χιλιάδες γραμμές προγράμματος!» Η απάντηση είναι: τα προγράμματα γράφονται με τη διαδικασία της **βήμα-προς-βήμα ανάλυσης** (§0.5). Έτσι τα κομμάτια προγράμματος που πρέπει να γραφούν και να αποδειχτούν έχουν μέγεθος και «δυσκολία» που μπορούμε να διαχειριστούμε[4]. Αυτά ισχύουν και για τον αντικειμενοστραφή προγραμματισμό· εκεί όμως έχουμε διαφορετικό τρόπο καταμερισμού της «δυσκολίας».

Ποιος από τους τρεις τρόπους εργασίας είναι η πιο συνηθισμένος; Ο δεύτερος και μάλιστα με απόδειξη από το τέλος προς την αρχή! Δηλαδή, ο προγραμματιστής –με τη βήμα-προς-βήμα ανάλυση ή με άλλον τρόπο– έχει φτάσει σε ένα πρόβλημα που από τις προδιαγραφές του «φαίνεται» η λύση. Γράφει λοιπόν τη λύση και μετά αποδεικνύει ότι είναι σωστή.

Ο πρώτος, όπως και ο δεύτερος, αλλά με απόδειξη από την αρχή προς το τέλος, είδες ότι έχουν πιο πολύπλοκη συνταγή για την εύρεση των συνθηκών που μας χρειάζονται. Έχει όμως και το εξής πρόβλημα: Οι συνθήκες «φουσκώνουν» πολύ γρήγορα –με κομμάτια που είναι άχρηστα στη συνέχεια– και γίνονται δύσχρηστες. Στο παράδειγμά μας είδες ότι καταλήξαμε με την (άχρηστη) s `==` a_0 μετά το τελευταίο βήμα.

Ο τρίτος έχει ενδιαφέρον όταν προσπαθούμε να λύσουμε το εξής πρόβλημα: Να γραφεί ένα πρόγραμμα που θα τροφοδοτείται με τις προδιαγραφές ενός προγράμματος και θα το γράφει, ας πούμε σε C++ (αυτόματη σύνθεση προγράμματος). Φυσικά, αν μπορέσουμε να

[4]Πάντως, συνθήκες που πιάνουν αρκετές γραμμές είναι συνηθισμένες.

γράψουμε αλγόριθμο που να κάνει κάτι τέτοιο, θα μπορούμε, πολύ πιο εύκολα να διατυπώσουμε κανόνες που να τους μάθει ένας άνθρωπος.

Εμείς θα χρησιμοποιούμε κατά κύριο λόγο τον δεύτερο τρόπο, με απόδειξη από το τέλος προς την αρχή και σπανιότερα τους άλλους.

Κάποιοι θα αναρωτηθούν «όταν τελειώσει η απόδειξη, θα σβήσουμε τα σχόλια με τις συνθήκες;» Καλύτερα να τις κρατάμε, τουλάχιστον ορισμένες από αυτές. Οι **συνθήκες επαλήθευσης** (verification conditions) είναι απαραίτητες σε περίπτωση που θα θελήσουμε να αλλάξουμε κάτι αργότερα.

3.8 Τα Προβλήματα των Τύπων Κινητής Υποδιαστολής

Πριν προσπαθήσουμε να κάνουμε μια απόδειξη προγράμματος με πραγματικούς αριθμούς (αυτού για την ελεύθερη πτώση της §2.7) ας δούμε μερικά (ακόμη) προβλήματα των τύπων κινητής υποδιαστολής.

Όπως λέγαμε στην §2.5 «*Η παράσταση των τιμών τύπου* `double` *(και* `float` *και* `long double`*) γίνεται προσεγγιστικώς.*» και «*οι πράξεις ... στους πραγματικούς τύπους (όπως ο* `double`*) δεν είναι* [ακριβείς]». Ήδη, στην §1.7.1 είδαμε ότι η:

```
cout.precision(20);
cout << 12.34567890123456789 << endl;
```

μας δίνει:

```
12.345678901234567884
```

Μια ματιά στον Πίν. 2-2 μας λέει ότι η ακρίβεια που μπορούμε να ζητήσουμε από τον `double` δεν υπερβαίνει τα 16 ψηφία. Φυσικά και στην περίπτωση αυτή έχουμε:

```
12.34567890123457
```

Ας προσπαθήσουμε να υπολογίσουμε δυνάμεις με τη χρήση της pow:

```
v = pow(0.3,4.0);   cout << " 0.3^4 = " << v << endl;
v = pow(0.7,2.0);   cout << " 0.7^2 = " << v << endl;
```

Αποτελέσματα:

```
0.3^4 = 0.0081
0.7^2 = 0.4899999999999999
```

Το δεύτερο είναι εντυπωσιακό.

Αν προσπαθήσουμε να κάνουμε το ίδιο πράγμα με τιμές τύπου `float` (εδώ βέβαια η ακρίβεια δεν μπορεί να είναι μεγαλύτερη από 8 ψηφία):

```
cout.precision(8);
v = pow(0.3f,4.0);  cout << " 0.3^4 = " << v << endl;
v = pow(0.7f,2.0);  cout << " 0.7^2 = " << v << endl;
```

θα πάρουμε:

```
0.3^4 = 0.0081000013
0.7^2 = 0.48999998
```

Τι συμπέρασμα βγαίνει από τα παραπάνω; Στις προδιαγραφές, όπου έχουμε τιμές πραγματικού τύπου (π.χ. `double`), δεν μπορούμε να βάζουμε ισότητα.

Στο παράδειγμα της ελεύθερης πτώσης γράψαμε:

Απαίτηση: $(tP == \sqrt{2h/g}\,)$ && $(vP == -\sqrt{2gh}\,)$

αλλά πιο πολύ νόημα θα είχε αν γράφαμε:

Απαίτηση: $(tP \approx \sqrt{2h/g}\,)$ && $(vP \approx -\sqrt{2gh}\,)$

Αν θέλουμε να γράψουμε κάτι πιο ποσοτικό θα μπορούσαμε να βάλουμε φράγματα στο απόλυτο σφάλμα:

Απαίτηση: $(|tP - \sqrt{2h/g}\,| < \varepsilon_{TA})$ && $(|vP + \sqrt{2gh}\,| < \varepsilon_{VA})$

όπου ε_{TA} και ε_{VA} θετικοί αριθμοί, *φράγματα στο απόλυτο σφάλμα*. Τι σημαίνει; Η τιμή της *tP* πρέπει να διαφέρει από την ακριβή τιμή της $\sqrt{2h/g}$ λιγότερο από ε_{TA} και η τιμή της *vP* να διαφέρει από την ακριβή τιμή της, $-\sqrt{2gh}$, λιγότερο από ε_{VA}.

Παρομοίως, μπορούμε να βάλουμε φράγμα στο σχετικό σφάλμα:

Απαίτηση: $(|tP - \sqrt{2h/g}| < \varepsilon_{TR}\sqrt{2h/g})$ **&&** $(|vP + \sqrt{2gh}| < \varepsilon_{VR}\sqrt{2gh})$

Τα ε_{TA}, ε_{VA}, ε_{TR} και ε_{VR} εξαρτώνται όχι μόνο από τον τύπο κινητής υποδιαστολής που χρησιμοποιούμε αλλά και (κυρίως) από το πρόβλημα που λύνουμε. Στο παράδειγμά μας δεν έχει νόημα ακρίβεια μεγαλύτερη από τρία ψηφία. Αυτό μεταφράζεται σε $\varepsilon_{TR} = \varepsilon_{VR} = 5\times10^{-4}$. Φυσικά, ακόμη και ο `float` μας εγγυάται πολύ καλύτερη ακρίβεια.

Και τι θα κάνουμε εδώ; Θα συνεχίσουμε να δίνουμε παραδείγματα στον τύπο `double`, αλλά στις προδιαγραφές θα βάζουμε είτε φράγματα στο σφάλμα, όπως παραπάνω, είτε το "≈" αντί για "`==`". Εσύ θα πρέπει να θυμάσαι ότι *οι πραγματικοί τύποι θέλουν ιδιαίτερη προσοχή*.

3.9 Μια Απόδειξη με Πραγματικούς

Και τώρα να δούμε: Είναι σωστό το πρόγραμμά για την ελεύθερη πτώση; Θα πάρουμε ως προδιαγραφές:

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

Απαίτηση: $(tP \approx \sqrt{2h/g})$ **&&** $(vP \approx -\sqrt{2gh})$

Θα πρέπει δηλαδή να αποδείξουμε:

```
// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   tP = sqrt(2*h/g);
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Θα κάνουμε την απόδειξη από το τέλος προς την αρχή και ξεκινάμε από την:

```
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Ποια συνθήκη θα πρέπει να ισχύει πριν από την εκτέλεση της εντολής; Σύμφωνα με τη συνταγή μας, καταλήγουμε στην:

$$(tP \approx \sqrt{2h/g}) \;\&\&\; (-g\cdot tP \approx -\sqrt{2hg})$$

Φτάσαμε λοιπόν στο:

```
   tP = sqrt(2*h/g);
// (tP ≈ √(2h/g)) && (gTp ≈ √(2hg))
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Τι θα πρέπει να έχουμε πριν από την `tP = sqrt(2*h/g)`; Και πάλι σύμφωνα με τη συνταγή μας βρίσκουμε:

$$(sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

Εδώ όμως πρέπει να προσέξουμε και κάτι άλλο:

- έχουμε διαίρεση δια *g* και θα πρέπει να έχουμε $g \ne 0$,
- έχουμε τετραγωνικές ρίζες και για να υπολογίζονται θα πρέπει να έχουμε: $2h/g \ge 0$ και $2hg \ge 0$ (η μια από τις δύο φτάνει).

  Θα πρέπει λοιπόν να έχουμε:

$$(g \ne 0) \;\&\&\; (hg \ge 0) \;\&\&\; (sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

  Από την προϋπόθεση, $(g == 9.81)$ **&&** $(h \ge 0)$

- έπεται η $hg \ge 0$ και
- από τη $g == 9.81$ έπεται η $g \ne 0$

Αν πούμε ότι η *sqrt* υπολογίζει μια, όσο γίνεται καλύτερη, *προσέγγιση* της τετραγωνικής ρίζας, ισχύει και η $(sqrt(2h/g) \approx \sqrt{2h/g})$ && $(g \cdot sqrt(2h/g) \approx \sqrt{2hg})$.

Πρόσεξε ακόμη ότι αν δεν έχουμε τη g == 9.81 οι υπολογισμοί μας χάνουν το φυσικό τους νόημα, αφού τα αποτελέσματα που έχουμε από τη Φυσική θέλουν αυτήν την τιμή της g.

Προσοχή! Το αν θα ισχύει η $h \geq 0$ εξαρτάται από την καλή πρόθεση του χρήστη. Αργότερα θα μάθουμε πώς μπορούμε να κάνουμε *έλεγχο* της προϋπόθεσης και να ενεργήσουμε ανάλογα.

Αλλά τώρα μας έρχονται και άλλες ιδέες: ας πούμε ότι το h δεν είναι αρνητικό· μήπως υπάρχει περίπτωση να είναι πολύ μεγάλο και να έχουμε υπερχείλιση; Ας το σκεφτούμε. Για το h θα έχουμε: $0 \leq h \leq DBL\_MAX$. Όταν πάει να εκτελέσει την εντολή:

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

για το πολλαπλασιασμό θα έχουμε:

$$0 \leq 2h \leq 2DBL\_MAX$$

Να μια περίπτωση υπερχείλισης[5] και μάλιστα χωρίς λόγο, διότι αν κάνουμε πρώτα τη διαίρεση δια g όλα πάνε καλά:

$$(2/g)h \leq (2/g)DBL\_MAX < DBL\_MAX, \text{ αφού } 2/g < 1$$

Θα πρέπει λοιπόν να υποχρεώσουμε, π.χ. με χρήση παρενθέσεων, τον υπολογιστή να κάνει τις πράξεις όπως εμείς θέλουμε:

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

Ύστερα από τα παραπάνω, η τιμή της *tP* θα είναι:

$$0 \leq tP == sqrt((2/g)h) \leq \sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Ερχόμαστε τώρα στην:

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

Από την σχέση για το *tP*, πολλαπλασιάζοντας επί g, παίρνουμε:

$$0 \leq g \cdot sqrt((2/g)h) \leq g\sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Αλλά, η $g\sqrt{2/g}\sqrt{DBL\_MAX} == \sqrt{2g}\sqrt{DBL\_MAX}$ είναι μικρότερη από *DBL_MAX* όταν έχουμε: $2g < DBL\_MAX$. Αυτό ισχύει για οποιονδήποτε τύπο `double`, αφού οι συνηθισμένες τιμές της *DBL_MAX* είναι της τάξης του 10^{308}. Άρα και η $-DBL\_MAX \leq g \cdot sqrt(2h/g) \leq DBL\_MAX$ ισχύει.

Βλέπουμε κοιπόν ότι η μελέτη της ορθότητας του προγράμματος μας οδηγεί σε διορθώσεις όπως αυτή της εντολής: `tP = sqrt(2*h/g)` που την αλλάξαμε σε: `tP = sqrt((2/g)/h)`. Και πρόσεξε και κάτι ακόμη: αν δεν γράφαμε τη `vP = -g*tP` για λόγους «οικονομίας», αλλά τη βάζαμε `vP = -sqrt(2*g*h)`, θα είχαμε και εδώ προβλήματα «υπερχείλισης χωρίς λόγο».

3.10 Τι Να Κάνουμε

Σε γενικές γραμμές έχουμε τρεις τρόπους για να αποδείξουμε την ορθότητα ενός προγράμματος. Συνήθως την αποδεικνύουμε πηγαίνοντας από το τέλος προς την αρχή. Αργότερα θα δούμε ότι και οι άλλοι δύο τρόποι είναι χρήσιμοι σε πολλές περιπτώσεις.

Τα εργαλεία που χρησιμοποιούμε στην απόδειξη είναι

- *αξιώματα* (όπως αυτό της εκχώρησης) και

[5] «Σιγά που θα έχουμε υπερχείλιση!» σκέφτεσαι «Για να γίνει κάτι τέτοιο θα πρέπει να έχουμε $h > DBL\_MAX/2$ και για τέτοιες τιμές δεν ισχύουν αυτοί οι τύποι.» Σωστό! Αλλά αυτά είναι δουλειά της Φυσικής· εδώ κοιτάμε μόνο τις προδιαγραφές μας.

- *συμπερασματικοί κανόνες* όπως αυτοί που είδαμε στο Κεφ. 0.

Ιδιαίτερη προσοχή θα πρέπει να δίνουμε σε «κακοτοπιές» όπως η υπερχείλιση, η κλήση συνάρτησης με όρισμα εκτός πεδίου ορισμού (π.χ. αρνητικό υπόριζο) και άλλα παρόμοια.

Θα πρέπει να αποδεικνύουμε την ορθότητα όλων των προγραμμάτων που γράφουμε; Κατ' αρχήν «Ναι!» αλλά αυτό δεν είναι και τόσο εύκολο. Αυτό που κάνουμε συνήθως είναι να επαληθεύουμε ορισμένα κρίσιμα κομμάτια του κάθε προγράμματος.

Ασκήσεις

Α Ομάδα

3-1 Απόδειξε ότι: Αν

```
   int k, sum;
```

τότε:

```
// sum == (k - 1)*k/2
   sum = sum + k;
   k = k + 1;
// sum == (k - 1)*k/2
```

3-2 Απόδειξε ότι: Αν

```
   double x, y, z;
```

τότε:

```
// z == y/2 - 1
   x = z + 1;   x = x + y/2;   z = z + 1;
// (x == y) && (z == y/2) }
```

Β Ομάδα

3-3 Αν, στο πρόβλημα της αντιμετάθεσης, ο τύπος των μεταβλητών είναι αριθμητικός, υπάρχει και άλλη λύση χωρίς βοηθητική μεταβλητή:

```
   a = a - b;  b = b + a;  a = b - a;
```

Απόδειξε ότι είναι σωστός. Αγνόησε την περίπτωση υπερχείλισης.

3-4 Απόδειξε ότι: Αν

```
   int d1, d2, q, r;
```

τότε:

```
// (r ≥ d2) && (r ≥ 0) && (d1 = q*d2 + r)
   q = q + 1;
   r = r - d2;
// (r ≥ 0) && (d1 == q*d2 + r)
```

3-5 Απόδειξε ότι: Αν

```
   int k, x, y;
```

τότε:

```
// true
   k = 0;  x = 1;  y = 1;
// (y = 2k+1) && (x == (k+1)²)
```

και

```
// (y == 2k+1) && (x == (k+1)²)
   k = k + 1;  y = y + 2;  x = x + y;
// (y == 2k+1) && (x == (k+1)²)
```

Γ Ομάδα

3-6 Τι σχέση υπάρχει μεταξύ (`double`) x και `static_cast<long>` (x); Διατύπωσε τις προδιαγραφές της συνάρτησης `static_cast<long>` (ή, τουλάχιστον, προσπάθησε).
Υπόδ.: Ξεχώρισε τις περιπτώσεις $x \geq 0$ και $x < 0$.

3-7 Δίνεται το πρόβλημα: Γράψε εντολές που να μεταθέτουν κυκλικώς τις τιμές τριών μεταβλητών a, b, c ($a \leftarrow b \leftarrow c \leftarrow a$). Διατύπωσε τις προδιαγραφές του και γράψε τις εντολές που το λύνουν μαζί με την απόδειξη ορθότητας.

κεφάλαιο

4

bool, char και Άλλοι Παρόμοιοι Τύποι

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσεις τις ιδιαιτερότητες των «ειδικών» ακέραιων τύπων `bool` και `char`. Μαζί με αυτούς θα δούμε και τους παρόμοιους *απαριθμητούς* τύπους.

Προσδοκώμενα αποτελέσματα:
Θα είσαι έτοιμος για αυτά που θα μάθεις στα επόμενα δύο κεφάλαια που στηρίζονται στον τύπο `bool`. Θα μπορείς να χειριστείς στα προγράμματά σου −εκτός από αριθμούς− και χαρακτήρες (και −αργότερα− κείμενα).

Έννοιες κλειδιά:

- *τύπος* `bool`
- *συνθήκες*
- *assert*
- *λογικές παραστάσεις*
- *τύποι* `char`, `signed char`, `unsigned char`
- *απαριθμητοί τύποι*
- *μετονομασία τύπου*
- `typedef`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Οι «Άλλοι» Ακέραιοι Τύποι:
Μέχρι τώρα γράφουμε τις συνθήκες επαλήθευσης σε σχόλια, για να παρακολουθούμε τη λογική του προγράμματός μας. Τώρα θα δούμε ότι μπορούμε:

- να τυπώνουμε την τιμή (1 για `true` ή 0 για `false`) μιας συνθήκης, με μια `"cout << …"`,
- να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή,

και (στο επόμενο κεφάλαιο):

- να ρυθμίζουμε την εκτέλεση του προγράμματός μας με βάση τις τιμές συνθηκών.

Αυτά έχουν να κάνουν με τον τύπο `bool`.
Θα δούμε ακόμη τους τύπους `char` και `wchar_t`.

Στο Κεφ. 2 είπαμε ότι όλοι αυτοί είναι ακέραιοι τύποι. Τι θα πει αυτό; Ας πούμε ότι έχουμε:

```
bool b( true );
char c1( ' ' ), c2( '!' ), c3;
int j;
```

και δίνουμε:

```
j = 32*b;
c3 = c1 + c2;
cout << j << "  " << c3 << endl;
```

Αποτέλεσμα:

```
32  A
```

Στη συνέχεια θα καταλάβεις πώς βγαίνουν αυτά τα αποτελέσματα αλλά και γιατί δεν πρέπει να γράφεις τέτοιες εντολές!

Πριν αρχίσεις τη μελέτη της §4.1 ρίξε μια ματιά στο Παρ. Α και μη διστάζεις να το συμβουλεύεσαι όσο θα μελετάς τις τρεις πρώτες παραγράφους.

4.1 Οι Συνθήκες στο Πρόγραμμα

Σε όσα είπαμε μέχρι τώρα είδαμε και μερικές συνθήκες, π.χ.:

$$(number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

$$(a == a_0) \;\&\&\; (b == b_0) \;\&\&\; (|a_0 + b_0| \le ½)$$

$$(|tP - \sqrt{2h/g}\,| < \varepsilon_{TR}\sqrt{2h/g}\,) \;\&\&\; (|vP + \sqrt{2gh}\,| < \varepsilon_{VR}\sqrt{2gh}\,)$$

Σε όλες τις περιπτώσεις (εκτός από την πρώτη) οι συνθήκες που είδαμε είναι συγκρίσεις που συνδέονται μεταξύ τους με λογικές πράξεις. Όποτε βάλαμε τις συνθήκες μέσα στο πρόγραμμά μας ήταν σε σχόλια. Η C++ μπορεί να «κατανοήσει» και να χρησιμοποιήσει μια συνθήκη αν γραφεί σύμφωνα με τους κανόνες που βάζει η γλώσσα:

- Στις συγκρίσεις, αντί των συμβόλων που χρησιμοποιούν τα μαθηματικά, χρησιμοποίησε αυτά που θέλει η C++. Δες το Πλ. 4.1.
- Στις λογικές πράξεις γράψε τους τελεστές όπως δίνονται στο Πλ. 4.2 αλλά πρόσεξε μερικές «επιφυλάξεις» που παραθέτουμε στη συνέχεια. Μπορείς να χρησιμοποιείς τα "`&&`", "`||`", "`!`", αντί των "$\wedge$", "$\vee$", "$\neg$" αντιστοίχως, αλλά αντί του "$\Leftrightarrow$" θα χρησιμοποιείς το "`==`", αντί του "$\Rightarrow$" το "`<=`" και αντί του "`xor`" το "`!=`". Ο Πίν. 4-1 είναι στην πραγ-

Τιμές Ορισμάτων		Αποτελέσματα Πράξεων					
P	Q	!P (not P)	P && Q (P and Q)	P || Q (P or Q)	P <= Q (P ⇒ Q)	P == Q (P ⇔ Q)	P != Q (P xor Q)
false	false	true	false	false	true	true	false
false	true	true	false	true	true	false	true
true	false	false	false	true	false	false	true
true	true	false	true	true	true	true	false

Πίν. 4-1 Οι πίνακες αλήθειας όλων των λογικών πράξεων που μας δίνει η C++.

Πλαίσιο 4.1

Τελεστές Συσχέτισης (Σύγκρισης)

Μαθηματικά	C++	Όνομα και σημασία του τελεστή συσχέτισης
=	`==`	ίσο με (π.χ. `x == 5`)
≠	`!=`	διάφορο
<	`<`	μικρότερο (π.χ. `x < 12`)
>	`>`	μεγαλύτερο
≤	`<=`	μικρότερο ή ίσο
≥	`>=`	μεγαλύτερο ή ίσο

ματικότητα σύντμηση των πινάκων του Παρ. Α.

- Αντικατάστησε τις διπλές (ή πολλαπλές) συγκρίσεις με απλές,

  π.χ. *μην γράφεις* *INT_MIN ≤ number ≤ INT_MAX*

  αλλά `INT_MIN <= number && number <= INT_MAX`

Η C++ σου επιτρέπει αντί των "`&&`", "`||`", "`!`" να χρησιμοποιείς τα "`and`", "`or`", "`not`" αντιστοίχως. Εμείς θα χρησιμοποιούμε εδώ τα "`&&`", "`||`", "`!`" μια και είναι αυτά που χρησιμοποιούνται συνήθως (από τον καιρό της C) και αυτά θα συναντάς όποτε διαβάζεις άλλα βιβλία. Άλλωστε, είναι πολύ πιθανό, η C++ που χρησιμοποιείς (π.χ. η Borland C++) να μη δέχεται τα "`not_eq`" (αντί για το "`!=`"), "`and`", "`or`", "`not`".

Τα "`==`", "`<`", "`>`", "`<=`", "`>=`", "`!=`", "`not_eq`", "`&&`", "`and`", "`||`", "`or`", "`!`", "`not`" είναι λεξικά σύμβολα (tokens) της C++ και δεν θα πρέπει να διαχωρίζεις τους χαρακτήρες τους, με οποιονδήποτε τρόπο, όταν τα γράφεις ενώ από την άλλη θα πρέπει να τα διαχωρίζεις από τα προηγούμενα και τα επόμενά τους.

Οι **λογικές παραστάσεις** (logical expressions), που η τιμή τους είναι "`true`" ή "`false`", κατασκευάζονται με κανόνες παρόμοιους με αυτούς που είδαμε για τις αριθμητικές παραστάσεις. Μπορούμε να πούμε, ότι μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές ("`true`" ή "`false`") και από συσχετίσεις (π.χ. συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους λογικούς τελεστές.

Στο Πλ.4.2 βλέπεις ένα συντακτικό κανόνα *για τις παρενθέσεις*. Μας λέει ότι μπορούμε να απλουστεύουμε τη γραφή λογικών παραστάσεων παραλείποντας παρενθέσεις.

Δηλαδή: *Μπορείς να γράψεις*

```
!(a > 0) && b <= 0 || c == 1
```

που είναι το ίδιο με το

```
((!(a > 0)) && (b <= 0)) || (c == 1)
```

Πάντως η δεύτερη μορφή είναι προτιμότερη.

Αν όμως θέλεις να γράψεις

```
(!(a > 0) && b <= 0) xor (c == 1)
```

θα *πρέπει να κρατήσεις τις παρενθέσεις*. Διότι αν γράψεις:

```
!(a > 0) && b <= 0 != c == 1
```

θα υπολογιστεί η `!(a > 0) && (((b <= 0) != c) == 1)` (δες τον Πίν. 4-2).

Ο ίδιος κανόνας υπάρχει, με άλλον τρόπο, και στον πίνακα του Παρ. Ε, και μας λέει ότι:

- πρώτα υπολογίζονται οι αρνήσεις "`!`" ("`not`") (προτ. 3),
- υπολογίζονται οι αριθμητικές παραστάσεις,
- στη συνέχεια γίνονται οι συγκρίσεις, πρώτα οι "`<`", "`<=`", "`>`", "`>=`" (προτ. 5) και μετά οι "`==`", "`!=`" (προτ. 6),

Πλαίσιο 4.2

Λογικοί Τελεστές

Λογική	C++
∧, **and**, **και**	`&&` `and`
∨, **or**, **ή**	`||` `or`
¬, ~, **όχι**	`!` `not`
⊻, **xor**	`!=`
⇒, ⊃	`<=`
⇔	`==`

Συντακτικός κανόνας:

- ***Στις λογικές πράξεις επιτρέπεται να μη βάλεις παρενθέσεις, αλλά να ξέρεις ότι οι πράξεις γίνονται με την εξής σειρά: πρώτα οι not (!), μετά οι and (&&), και τέλος οι or (||). Για τις "!=" (xor), "<=" (⇒) και "==" (⇔) χρειάζονται παρενθέσεις.***

- μετά γίνονται οι λογικές πράξεις με τη σειρά: "`&&`" ("`and`") (προτ. 10) και "`||`" ("`or`")[1] (προτ. 11),
- τέλος γίνονται οι εκχωρήσεις.

Μερικά παραδείγματα λογικών παραστάσεων:

```
(x > 0) || (y > 0)
(x > 0) == (y > 0)
(x < 0) == (y < 0)
(k <= l) && (l < m + 5)
(a – 2 > b) && (b < c) || (c == 0)
(a – 2 > b) && ((b < c) || (c == 0))
```

Βέβαια η τιμή όλων αυτών των παραστάσεων, `true` ή `false`, εξαρτάται από τη συγκεκριμένη τιμή των λογικών ορισμάτων. Αν π.χ. υποθέσουμε ότι τα αριθμητικά ορίσματα (τύπου `int`) έχουν τιμές: $x = 5$, $y = 4$, $a = -1$, $b = c = 0$, $k = -2$, $l = 1$ και $m = -4$, τότε οι προηγούμενες λογικές παραστάσεις θα πάρουν τις εξής τιμές:

`(x > 0) || (y > 0)`: `true` (επειδή $x > 0$ και $y > 0$),
`(x > 0) == (y > 0)`: `true` (επειδή $(x > 0)$ `true` και $(y > 0)$ `true`),
`(x < 0) == (y < 0)`: `true` (επειδή $(x < 0)$ `false` και $(y > 0)$ `false`),
`(k <= l) && (l < m + 5)`: `false` (επειδή: $(k <= l)$ `true` ενώ $(l < m + 5)$ `false`),
`(a – 2 > b) && (b < c) || (c == 0)`: `true` (επειδή c `==` 0),
`(a – 2 > b) && ((b < c) || (c == 0))`: `false` (επειδή $(a - 2 > b)$ `false`).

Πρόσεξε τη 2η και την 3η: το `==` υλοποιεί την αμοιβαία συνεπαγωγή (⇔). Και οι δύο παραστάσεις έχουν τιμή `true` διότι και στις δύο περιπτώσεις οι ανισότητες έχουν την ίδια τιμή: στη 2η και οι δύο είναι `true`, στην 3η και οι δύο είναι `false`.

Πρόσεξε ακόμη την 5η και την 6η:

- Αφού η `&&` υπολογίζεται πριν από την `||`, η 5η ισοδυναμεί με: `((a – 2 > b) && (b < c)) || (c == 0)` και επειδή `true ||` οτιδήποτε μας κάνει `true` και η `c == 0` είναι `true`, όλη η παράσταση είναι `true`.
- Στην 6η, με τις παρενθέσεις ζητούμε να υπολογιστεί πρώτα η `||` και μετά η `&&`. Επειδή `false &&` οτιδήποτε μας κάνει `false` και η `a – 2 > b` είναι `false`, όλη η παράσταση είναι `false`.

[1] Εδώ υπάρχει ένα λεπτό σημείο που θα το δούμε στο επόμενο κεφάλαιο.

Παίρνοντας υπόψη την προτεραιότητα των πράξεων μπορούμε να βγάλουμε μερικές παρενθέσεις και τα παραδείγματά μας να γραφούν:

```
x > 0 || y > 0
x > 0 == y > 0
x < 0 == y < 0
k <= l && l < m + 5
a - 2 > b && b < c || c == 0
a - 2 > b && (b < c || c == 0)
```

Και τώρα πρόσεξε:

- ***Σε μια εντολή εξόδου (`cout << …`) μπορείς να βάζεις ως όρισμα μια συνθήκη, μέσα σε παρενθέσεις. Αυτό που θα γίνει είναι το εξής: θα υπολογισθεί η τιμή της και αν είναι "`true`" (αν ισχύει) θα τυπωθεί η τιμή "1" (ένα), ενώ αν είναι "`false`" (αν δεν ισχύει) θα τυπωθεί η τιμή "0" (μηδέν).***

Με άλλα λόγια, αν *Π* μια λογική παράσταση τότε η εντολή:

```
cout << (Π) …
```

ισοδυναμεί με:

```
cout << 1 …
```
αν η *Π* έχει τιμή `true`, και με
```
cout << 0 …
```
αν η *Π* έχει τιμή `false`.

Παράδειγμα

Αν έχουμε δηλώσει:

```
int  n1, n2;
```

τότε οι εντολές:

```
n1 = 12;  n2 = 13;
cout << n1 << " < " << n2 << " : " << (n1 < n2) << endl;
cout << n1 << " > " << n2 << " : " << (n1 > n2) << endl;
```

θα δώσουν:

```
12 < 13 : 1
12 > 13 : 0
```

Στην επόμενη παράγραφο βλέπεις πώς χρησιμοποιούμε αυτή τη δυνατότητα για να ελέγχουμε την ορθότητα ενός προγράμματος.

Προς το παρόν όμως δες κάτι που μπορεί να σου συμβεί: Είπαμε πιο πάνω ότι για να «κατανοήσει» η C++ μια λογική παράσταση πρέπει να αντικαταστήσεις «*τις διπλές (ή πολλαπλές) συγκρίσεις με απλές*», π.χ. αντί για `0 <= x < 10` γράψε `(0 <= x) && (x < 10)`. Αν δεν το κάνεις, ο μεταγλωττιστής θα δεχθεί τη διπλή σύγκριση αλλά θα την «κατανοήσει» ως `(0 <= x) < 10` που έχει πάντοτε τιμή `true` διότι το `(0 <= x)` είναι μικρότερο από 10 είτε έχει τιμή `true` (1) είτε έχει τιμή `false` (0).

4.1.1 Το Λάθος που θα Κάνεις Συχνά!

Ο τίτλος της παραγράφου είναι απαισιόδοξος, αλλά είναι παρατηρημένο: οι αρχάριοι στη C++ (και στη C) κάνουν το ίδιο *σοβαρότατο* σφάλμα: προσπαθούν (μάταια) να συγκρίνουν για ισότητα με το "`=`" αντί για το "`==`". Κοίταξε όμως τι μπορεί να συμβεί:

Ας πούμε ότι έχεις: *a* `==` –1, *b* `==` *c* `==` 0 και θέλεις να τυπώσεις την τιμή της παράστασης:

```
(a – 2 > b) && (b < c) || (c == 0)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 0)) << endl;
```

Αποτέλεσμα: **0** (**false**)! Πώς έγινε αυτό; Η C++ υπολογίζει την τιμή της *εκχώρησης* [2] **c = 0** που είναι μηδέν και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 0 ως **false**.

Ας πούμε τώρα ότι θέλεις την τιμή της παράστασης

```
(a – 2 > b) && (b < c) || (c == 44)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 44)) << endl;
```

Αποτέλεσμα: **1** (**true**)! Εδώ τι έγινε; Η C++ υπολογίζει την τιμή της *εκχώρησης* **c = 44** που είναι 44 και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 44 ≠ 0 ως **true**. Εκτός από αυτό όμως, η *c*, για το υπόλοιπο πρόγραμμά σου, έχει τιμή 44 αντί για 0 που θα ήθελες να έχει!

Και στις δύο περίπτώσεις η Borland C++ θα σου δώσει *προειδοποίηση*: «**Possibly incorrect assignment**» (πιθανόν εσφαλμένη εκχώρηση). Καλό είναι λοιπόν *να προσέχεις*, όχι μόνον τα *λάθη*, *αλλά* και *τις προειδοποιήσεις*.

4.2 Οι Τιμές των Συνθηκών Επαλήθευσης

Αν ένα πρόγραμμά μας είναι σωστό τότε, κάθε φορά που εκτελείται, όλες οι συνθήκες επαλήθευσης θα (πρέπει να) έχουν τιμή **true**. Αν αυτό δεν συμβαίνει τότε κάτι δεν πάει καλά.

Όπως καταλαβαίνεις, σύμφωνα με όσα είπαμε στην προηγούμενη παράγραφο μπορούμε να ζητήσουμε από τον υπολογιστή να υπολογίσει και να μας τυπώσει τις τιμές των συνθηκών επαλήθευσης.

Δες ένα παράδειγμα με το πρόγραμμα της αντιμετάθεσης:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   S = a;
// (b == b0) && (S == a0)
   cout << "   1: " << ((b == b0) && (S == a0)) << endl;
   a = b;
// (a == b0) && (S == a0)
   cout << "   2: " << ((a == b0) && (S == a0)) << endl;
   b = S;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Πρόσεξε τα εξής:

- Κάτω από κάθε σχόλιο με συνθήκη επαλήθευσης υπάρχει μια "**cout << …**" που τυπώνει την τιμή της συνθήκης. Οι εκτυπώσεις αρχίζουν με ένα χαρακτηριστικό: « **Προ:**» για την προϋπόθεση, « **Απτ:**» για την απαίτηση και αριθμούς 1, 2, 3,... για τις ενδιάμεσες συνθήκες.

[2] «τιμή της *εκχώρησης*»!; Ναι, υπάρχει τέτοιο πράγμα και θα το δούμε αργότερα· προς το παρόν: είναι η τιμή που αποθηκεύεται στη μεταβλητή.

- Πρόσεξε πώς έγινε η μετάφραση των συνθηκών σε C++, σύμφωνα με τους κανόνες που είπαμε.

  Να ένα παράδειγμα εκτέλεσης:

```
Προ: 1
  1: 1
  2: 1
Απτ: 1
a = 20   b = 10
```

  Ας δούμε τώρα το πρώτο, εσφαλμένο, πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   a = b;  b = a;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Παράδειγμα εκτέλεσης:

```
Προ: 1
Απτ: 0
a = 20   b = 20
```

Προσοχή! Να υπενθυμίσουμε ότι με τύπους κινητής υποδιαστολής μπορεί να έχεις και μερικές εκπλήξεις όταν οι συνθήκες σου έχουν ισότητες[3].

Βέβαια, θα πρέπει να (ξανα)τονίσουμε ότι το να πάρεις μερικά παραδείγματα εκτέλεσης με όλες τις συνθήκες `true` δεν σημαίνει ότι το πρόγραμμά σου είναι σωστό.

4.3 Έλεγχος Συνθηκών Επαλήθευσης: *assert()*

Στο επόμενο κεφάλαιο θα μάθουμε πώς να ελέγχουμε τη ροή της εκτέλεσης του προγράμματος. Προς το παρόν όμως θα δούμε έναν τρόπο για να αποφασίζουμε αν θα *διακόψουμε* την εκτέλεση του προγράμματος σε σχέση με τις τιμές των συνθηκών επαλήθευσης.

Δες έναν άλλον τρόπο για να γράψουμε το δεύτερο πρόγραμμα της προηγούμενης παραγράφου:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   a = b;  b = a;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

[3] Για δοκίμασε να κάνεις τα ίδια με το πρόγραμμα της ελεύθερης πτώσης. Χα!

Η εκτέλεση του προγράμματος δίνει:

```
Assertion failed: a == b0 && b == a0, file test02a.cpp, line 13
```

Τι είναι αυτή η «παράξενη» εντολή "`assert(`*συνθήκη*`)`"; Να τη σκέφτεσαι ως κλήση συνάρτησης, όπως αυτές που μάθαμε στο Κεφ. 1, που όμως:

- παίρνει όρισμα μια *συνθήκη* (ή μια τιμή τύπου `bool`, όπως θα μάθουμε στην επόμενη παράγραφο) και
- *δεν επιστρέφει* κάποια *τιμή*.

Για τέτοιες συναρτήσεις θα μιλήσουμε αργότερα.

Για να χρησιμοποιήσεις την *assert* θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <cassert>`".

Ας δούμε τη λειτουργία της:

- Την πρώτη φορά που καλείται, στη γραμμή 11, η συνθήκη-όρισμα ισχύει (`true`). Ως προς τα αποτελέσματα: είναι σαν να μην υπάρχει.
- Τη δεύτερη φορά καλείται, στη γραμμή 13, η συνθήκη-όρισμα δεν ισχύει (`false`). Ως αποτέλεσμα διακόπτεται η εκτέλεση του προγράμματος και παίρνουμε αυτά που είδες.

Ξαναγράφουμε και το πρώτο πρόγραμμα ως εξής:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   S = a;
   assert( (b == b0) && (S == a0) );
   a = b;
   assert( (a == b0) && (S == a0) );
   b = S;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

Αποτέλεσμα:

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

Καμιά ένδειξη για την ύπαρξη της *assert*![4]

Τέλος, ας δούμε πώς μπορούμε να χρησιμοποιήσουμε την *assert* για τη διασφάλιση της προϋπόθεσης του προγράμματος της ελεύθερης πτώσης (§2.7). Στο αρχείο `eleptw.cpp` έχουμε το πρόγραμμά μας που τώρα, με την εισαγωγή της "`assert( h >= 0 )`" στη 14η γραμμή, έχει την εξής μορφή:

```
#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
```

[4] Παρ' όλα αυτά η εκτέλεση του προγράμματος γίνεται πιο αργή. Θα δούμε αργότερα ότι αυτό διορθώνεται.

```
// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
   tP = sqrt( 2*h/g );
   vP = -sqrt(2*h*g);
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
// Τύπωσε τα tP, vP
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << vP << " m/sec" << endl;
}
```

Δες ένα παράδειγμα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: -80
Assertion failed: h >= 0, file eleptw.cpp, line 14
```

Φυσικά, αν δίναμε αρχικό ύψος **80** το αποτέλεσμα θα ήταν ακριβώς αυτό που ξέραμε από το αρχικό πρόγραμμα.

4.4 Ο Τύπος bool

Ας δούμε τώρα πώς μπορούμε «*να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή*», όπως λέγαμε στην §4.1.

Στα προηγούμενα κεφάλαια γνωρίσαμε τον τρόπο γραφής και δήλωσης δύο τύπων στοιχείων: **int** (ακέραιοι) και **double** (πραγματικοί). Στην παράγραφο αυτή θα γνωρίσουμε έναν τρίτο τύπο στοιχείων, που λέγεται τύπος **bool**. Οι τιμές του τύπου **bool** είναι οι **false** και **true** και είναι διαταγμένες:

```
false < true
```

Μεταξύ τιμών τύπου **bool** μπορεί να γίνονται οι λογικές πράξεις που μάθαμε: η "**&&**", η "**||**", η "**!**". Μα είπαμε ότι μπορεί να γίνονται και οι άλλες: ⇒, ⇔, **xor**! Ναι, αλλά να καταλάβουμε τι συμβαίνει: η υλοποίησή τους στηρίζεται στη διάταξη που δώσαμε παραπάνω. Πήγαινε στο Παρ. Α και

- Δες τον αληθοπίνακα της "⇒" (Πίν. Α-3)· στις γραμμές 1, 2 και 4 για τις οποίες η $P \Rightarrow Q$ έχει τιμή **true**, με βάση τη διάταξη **false** < **true**, ισχύει η P **<=** Q. Στη γραμμή 3, όπου η $P \Rightarrow Q$ έχει τιμή **false**, έχουμε $P > Q$ ή αλλιώς **!**(P **<=** Q).
- Δες τον αληθοπίνακα της "**xor**" (Πίν. Α-4)· στις γραμμές 2 και 3 για τις οποίες η P **xor** Q έχει τιμή **true**, έχουμε P **!=** Q, ενώ στις 1 και 4 όπου η P **xor** Q έχει τιμή **false**, P **==** Q ή **!**(P **!=** Q).
- Τέλος, δες τον αληθοπίνακα της "⇔" (Πίν. Α-5)· στις γραμμές 1 και 4 για τις οποίες η $P \Leftrightarrow Q$ έχει τιμή **true**, έχουμε P **==** Q, ενώ στις 2 και 3 όπου η $P \Leftrightarrow Q$ έχει τιμή **false**, P **!=** Q ή **!**(P **==** Q).

Μπορούμε λοιπόν να συμπληρώσουμε αυτό που είπαμε πιο πάνω: μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές (**true** ή **false**), μεταβλητές τύπου **bool** και από συσχετίσεις (ή συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους τελεστές "**&&**", "**||**", "**!**", "**<=**", "**==**" και "**!=**". Ακόμη, σε μια μεταβλητή τύπου **bool** μπορείς να αποθηκεύεις την τιμή μιας συνθήκης (λογικής παράστασης).

Οι μεταβλητές τύπου **bool** δηλώνονται στο πρόγραμμα, όπως και οι μεταβλητές των άλλων τύπων που γνωρίσαμε, στο μέρος δήλωσης μεταβλητών. Π.χ.:

```
   bool   x, y, logical, signal, ok, lgc;
   int    i, k;
```

```
   double a;
```

Η παραπάνω δήλωση σημειώνει ότι κατά την εκτέλεση του προγράμματος οι μεταβλητές *x*, *y*, *logical*, *signal*, *ok* και *lgc* θα παίρνουν τις τιμές **true** ή **false**. Έτσι π.χ., μπορούμε να γράψουμε μέσα στο πρόγραμμα τις παρακάτω εντολές εκχώρησης, όπου η δεξιά πλευρά είναι μια λογική παράσταση (ή μια λογική σταθερά), ενώ η αριστερή πλευρά είναι μια μεταβλητή τύπου **bool**.

```
    x = true;                       y = !signal || x;
    logical = (x || y) && signal;   ok = (a <= 18) && (k != 2);
    lgc = false;                    signal = k < -1;
```

Παράδειγμα

Το παρακάτω πρόγραμμα τυπώνει το περιεχόμενο του Πίν. 4-1, μόνο που αντί για **false** και **true** έχει 0 και 1 αντίστοιχα.

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q" << endl;
   P = false;  Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = false;  Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
}
```

Αποτέλεσμα:

```
P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q
0   0   1   0    0    1    1    0
0   1   1   0    1    1    0    1
1   0   0   0    1    0    0    1
1   1   0   1    1    1    1    0
```

Αφού όταν ζητάμε να τυπωθεί **true** ή **false** παίρνουμε 1 ή 0 αντιστοίχως καταλαβαίνεις και γιατί (στο πρόγραμμα της εισαγωγής) το **32*true** μας δίνει 32! Ο **bool** είναι ένας ακέραιος τύπος με δύο τιμές 0 και 1. Αυτό θα *πρέπει να προσπαθείς να το ξεχάσεις* αν και θα υπάρχουν πολλά για σου το θυμίζουν.

- Να θυμάσαι μόνον ότι **false < true** (και όχι τα 0 και 1).
- Να θυμάσαι ότι **!false == true** και **!true == false** (και όχι ότι **1-false == true** και **1-true == false**).

Να δούμε τώρα τις κληρονομιές από τη C: Στη C δεν υπάρχει τύπος **bool** και *οποιαδήποτε τιμή δεν είναι μηδέν θεωρείται ως "***true***", ενώ το "***0***" θεωρείται ως "***false***"*. Η C++ είναι συμβατή με αυτά. Αν έχεις δηλώσει:

```
   bool b;
```

τότε οι εντολές:

```
   b = static_cast<bool>( 37.5 );  cout << b << "  ";
```

```
   b = 37.5;                        cout << b << "  ";
   b = static_cast<bool>( 0 );      cout << b << "  ";
   b = 0;                           cout << b << endl;
```

δίνουν:

```
1  1  0  0
```

Δηλαδή, η μη μηδενική τιμή (τύπου **double**) γίνεται **true** με ανοικτή ή εννοούμενη τυποθεώρηση ενώ το 0 γίνεται **false**.

Καλώς ή κακώς, εργαλεία από τη C θα χρησιμοποιούμε συχνά και δεν γίνεται να ξεφύγουμε από χρήσεις μη λογικών τιμών σε θέσεις όπου περιμένουμε τιμή τύπου **bool**. Εμείς πάντως θα αποφεύγουμε παρόμοιες χρήσεις. Όσο είναι δυνατό.

Σημείωση: ▶

Για τις μεταβλητές τύπου **bool** θα δεις πολύ συχνά τον όρο **σημαία** (flag) που μπορεί να είναι *ανεβασμένη* (**true**) ή *κατεβασμένη* (**false**). ◀

4.4.1 Για να Γράφουμε "false" και "true"

Αν τα "**0**" και "**1**" σε ενοχλούν πολύ και θέλεις να γράφεις "**false**" και "**true**" μπορείς να στείλεις προς το *cout* το μήνυμα "**boolalpha**" και όλα θα αλλάξουν με μιας: Το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
  cout << false << "  " << true << endl;
  cout << boolalpha;
  cout << false << "  " << true << endl;
}
```

θα δώσει:

```
0  1
false  true
```

4.5 Οι Τύποι char

Στην §2.3 μεταξύ των άλλων ακέραιων τύπων είδαμε και τον τύπο **char**. Στην πραγματικότητα η C++ έχει τρεις διαφορετικούς τύπους:

- **char**,
- **signed char**,
- **unsigned char**.

Ας δούμε ένα μικρό αλλά χαρακτηριστικό

Παράδειγμα 1 ➴

```
#include <iostream>
using namespace std;
int main()
{
   char c;     unsigned char u;     signed char s;

   c = 195;  u = 195;  s = 195;
   cout << c << "  " << u << "  " << s << endl;
   cout << static_cast<int>(c) << "  " << static_cast<int>(u)
        << "  " << static_cast<int>(s) << endl;
}
```

Αποτέλεσμα:

```
Γ  Γ  Γ
-61  195  -61
```

Ας τα πάρουμε από την αρχή:

- Με τις εντολές: `c = 195;  u = 195;  s = 195` δίνουμε και στις τρεις μεταβλητές την ίδια ακέραιη τιμή: 195. Να τονίσουμε βέβαια ότι αυτές οι εντολές εκτελούνται ως:

```
c = static_cast<char>(195);
u = static_cast<unsigned char>(195);
s = static_cast<signed char>(195) .
```

- Το αποτέλεσμα της `cout << c << "  " << u << "  " << s << endl` είναι λίγο αναπάντεχο: `Γ  Γ  Γ`. Πώς εξηγείται; Στο Παρ. Δ, Πίν. Δ-4 (Δ-3) μπορείς να δεις ότι στο σύνολο χαρακτήρων για τα Windows, στη θέση 195, υπάρχει το ελληνικό γράμμα Γ.
- Η δεύτερη γραμμή είναι εντελώς ανεξήγητη για όποιον δεν ξέρει αριθμητική modulo (mod 256 στην περίπτωσή μας). Απλώς παρατήρησε ότι 61 = 256 – 195.

&

Τώρα, ξεχνούμε για λίγο το ότι βάλαμε τον `char` στους ακέραιους τύπους και ξεκινούμε από το εξής: *στον τύπο* `char` *παριστάνουμε χαρακτήρες*· όλους τους χαρακτήρες που περιλαμβάνει το σύνολο χαρακτήρων του υπολογιστή.

Παράδειγμα 2

Μέσα στο πρόγραμμά μας μπορεί να δηλώσουμε:

```
const char plus = '+', space = ' ', aLower = 'a', aUpper = 'A';
char letter, digit, spcChar;
```

και στη συνέχεια να δώσουμε:

```
letter = aLower;
digit = '7';
spcChar = plus;
cout << ' ' << letter << space << digit << spcChar << endl;
```

Αποτέλεσμα:

```
 a 7+
```

ή, για να το δεις καλύτερα:

```
␣a␣7+
```

&

Οι σταθερές τύπου `char` έχουν τη μορφή που βλέπεις στα παραδείγματα: ένας χαρακτήρας μεταξύ δύο αποστρόφων:

```
'+'   ' '   'a'   'A'   '7'
```

Όπως είναι φανερό, έχεις πρόβλημα να παραστήσεις την απόστροφο· η λύση είναι αυτή που λέγαμε και για τους ορμαθούς χαρακτήρων: αν θέλεις να γράψεις σταθερές τύπου `char` που έχουν ως τιμή κάποιον από τους χαρακτήρες: `\`, `?`, `'`, `"` θα πρέπει να γράψεις: `'\\'`, `'\?'`, `'\''`, `'\"'`. Με παρόμοιο τρόπο (Πίν. 4-2) μπορείς να παραστήσεις και *μη εκτυπώσιμους* χαρακτήρες.

Στο παράδειγμά μας είδαμε ότι σε μια μεταβλητή τύπου `char` μπορούμε να δίνουμε τιμές με εντολές εκχώρησης. Μπορούμε να διαβάζουμε και από το πληκτρολόγιο; Βεβαίως.

LF	`'\n'`
HT	`'\t'`
VT	`'\v'`
BS	`'\b'`
CR	`'\r'`
FF	`'\f'`
BEL	`'\a'`
\	`'\\'`
?	`'\?'`
'	`'\''`
"	`'\"'`

Πίν. 4-2 Πώς γράφουμε ως σταθερές τύπου `char` μερικούς χαρακτήρες εκτυπώσιμους και μη. Για τους μη εκτυπώσιμους δες το Παρ. D.

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3;
   :
cin >> c1 >> c2 >> c3;
cout << c1 << c2 << c3 << endl;
```

Πληκτρολογούμε:

a b c<Enter>

και παίρνουμε:

```
abc
```

Θαυμάσια! Αλλά… Δεν είπαμε ότι το κενό (διάστημα) είναι χαρακτήρας; Γιατί δεν έγινε τιμή της *c2*; Γι' αυτό φταίει ο "`>>`" του *cin* που «τρώει» τα διαστήματα, τις αλλαγές γραμμής και άλλα παρόμοια. Αν θέλεις να διαβάζεις τα πάντα χρησιμοποίησε τη

```
cin.get(a);
```

που διαβάζει από το πληκτρολόγιο έναν χαρακτήρα και τον αποθηκεύει ως τιμή της *a*, τύπου `char`. Αλλά εδώ προσοχή: *πρέπει να διαβάζεις τα πάντα*!

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3, c4;

cin.get(c1);  cin.get(c2);
cin.get(c3);  cin.get(c4);
cout << " c1: " << c1 << endl;
cout << " c2: " << c2 << endl;
cout << " c3: " << c3 << endl;
cout << static_cast<int>(c4) << endl;
```

Πληκτρολογούμε:

```
a b<enter>
```

και παίρνουμε:

```
 c1: a
 c2:
 c3: b
10
```

Στη δεύτερη γραμμή έχουμε στην πραγματικότητα: "`c2: ␣`". Να το διάστημα που λέγαμε. Το 10 στην τέταρτη γραμμή είναι αποτέλεσμα της: `cout << static_cast<int>(c4)`. Στη *c4* αποθηκεύτηκε το <Enter> (LF, '`\n`') που δώσαμε στο τέλος της γραμμής!

Από το τελευταίο παράδειγμα θα πρέπει να καταλαβαίνεις ότι αν στο ίδιο πρόγραμμα ανακατέψεις "`cin >> …`" και "`cin.get(v)`" θα έχεις προβλήματα.

- ***Σε ένα πρόγραμμα θα δουλεύεις είτε με "`cin >> …`" είτε με "`cin.get(v)`". Στη δεύτερη περίπτωση θα παίρνεις ό,τι στέλνει το πληκτρολόγιο προς το πρόγραμμά σου. Για να τις ανακατέψεις θα πρέπει να ξέρεις πώς ακριβώς δουλεύει η "`cin >> …`".***

4.5.1 Το Σύνολο Χαρακτήρων και οι Τύποι `char`

Η θέση του χαρακτήρα μέσα στο σύνολο χαρακτήρων τον καθορίζει απόλυτα. Π.χ. στη θέση 65 του ASCII υπάρχει το κεφαλαίο γράμμα '`A`' του λατινικού αλφαβήτου.

Οι γλώσσες προγραμματισμού βάζουν μερικές, πολύ χαλαρές, προδιαγραφές στα σύνολα χαρακτήρων που χρησιμοποιούν:

- Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη.
- Αν τα πεζά γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.
- Αν τα κεφαλαία γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.

Αν έχουμε μια τιμή c τύπου `unsigned char`, με τυποθεώρηση με τη `static_cast<int>` μπορούμε να βρούμε τη θέση (order) της στο σύνολο χαρακτήρων. Π.χ. οι:

```
unsigned char c;

c = 'D';  cout << static_cast<int>(c);
```

```
c = 'Γ';  cout << "   " << int(c);
cout << "   "
 << static_cast<int>(static_cast<unsigned char>('ώ')) << endl;
```

θα δώσουν:

```
68   195    254
```

Αντιστρόφως, τυποθεώρηση τιμής τύπου `int` από τον `unsigned char` μας δίνει το χαρακτήρα που υπάρχει στη θέση που καθορίζει το όρισμα. Π.χ. οι:

```
cout << static_cast<unsigned char>(68) << "   ";
cout << static_cast<unsigned char>(195) << "   ";
cout << static_cast<unsigned char>(254) << endl;
```

θα δώσουν:

```
D   Γ   ώ
```

Μπορούμε λοιπόν να πούμε ότι: για οποιαδήποτε τιμή *c*, τύπου `unsigned char` έχουμε:

`static_cast<unsigned char>(static_cast<int>(c)) == c`

και αντιστρόφως (`int k`, $0 \le$ `k` ≤ 255):

`static_cast<int>(static_cast<unsigned char>(k)) == k`

Αλλά, πώς αποθηκεύεται στη μνήμη μια τιμή τύπου `unsigned char`; Αποθηκεύεται η «ζωγραφιά»; Όχι βέβαια. Μετά την εκτέλεση της εντολής: `c = 'D'` αυτό που θα αποθηκευτεί στη μνήμη, στη θέση *c*, είναι ο *ακέραιος* που δίνεται από την `static_cast<int>('D')`, σε δυαδική παράσταση.

Το να εμφανιστεί η ζωγραφιά στην οθόνη σου ή στον εκτυπωτή σου είναι ένα άλλο πρόβλημα που έχει να κάνει με το μέσο και την τεχνολογία του. Παλιότερα, στον τύπο `unsigned char` ήταν δυνατή η παράσταση όλων των χαρακτήρων που μπορούσε να εμφανίσει ένας ΗΥ. Σήμερα τα πράγματα είναι πιο πολύπλοκα. Οι ΗΥ έχουν δυνατότητα παράστασης πάρα πολλών χαρακτήρων –ο κατάλληλος τύπος είναι πια ο `wchar_t` που θα δούμε στη συνέχεια– ενώ ο `unsigned char` παριστάνει ένα επιλεγμένο υποσύνολο που μπορεί να αλλάζει.

Οι τύποι `signed char` και `char` έχουν και αυτοί τη δυνατότητα να παραστήσουν όλους τους χαρακτήρες που μπορεί να παραστήσει ο `unsigned char`, αλλά έχουμε μια διαφορά με το αποτέλεσμα της `int` για τέτοιες τιμές: αν το 8ο (ή το 1ο αν προτιμάς) δυαδικό ψηφίο είναι 1 ερμηνεύεται ως πρόσημο (–). Π.χ. οι:

```
unsigned char uc;
char c;

c = 'ώ';  uc = 'ώ';
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
c = c - 1;  uc = c;
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
```

θα δώσουν:

```
ώ   ώ
-2   254
ύ   ύ
-3   253
```

Ας δούμε τι ενδιαφέροντα υπάρχουν εδώ:

1. Και στις δύο μεταβλητές δώσαμε την ίδια τιμή, `'ώ'`, επιτυχώς, όπως φαίνεται και από την εκτύπωση των τιμών τους.
2. Οι `static_cast<int>(c)` και `static_cast<int>(uc)` προκάλεσαν την εκτύπωση των –2 και 254. Τι συνέβη; Πρόκειται για δύο διαφορετικές ερμηνείες της ίδιας δυαδικής παράστασης: 11111110_2. Στην πρώτη περίπτωση, που περιμένουμε προσημασμένο

αριθμό, το 1ο "1" θεωρείται ότι δείχνει αρνητική τιμή. Αυτά θα τα μάθουμε αργότερα. Πάντως προς το παρόν μπορείς να θυμάσαι τον εξής κανόνα:

Έστω ότι οι *c* (`int`) και *uc* (`unsigned int`) έχουν ίδια τιμή (στην περίπτωση αυτή: τον ίδιο χαρακτήρα)

αν `static_cast<int>(c)` ≥ 0 τότε
`static_cast<int>(c) == static_cast<int>(uc)`
αλλιώς (αν `static_cast<int>(c)` < 0)
`static_cast<int>(c) == (static_cast<int>(uc) - 256)`

3. Μειώσαμε την τιμή της *c* με τη `c = c - 1`. Έχουμε αυτό το δικαίωμα, αφού οι `char` θεωρούνται ακέραιοι τύποι.
4. Εκχωρήσαμε την τιμή της *c* στη *uc*· μην ξεχνάς ότι το νόημα της εκχώρησης είναι `uc = static_cast<unsigned char>(c)`. Μετά την εκχώρηση οι δύο μεταβλητές έχουν το ίδιο περιεχόμενο (`'ϋ'`).

Στα Windows, στους τύπους `char` και `signed char` όλοι οι χαρακτήρες παριστάνονται με ακέραιες τιμές από −128 μέχρι 127.

4.6 Ο Τύπος char στο Πρόγραμμα

Ας δούμε τώρα το εξής πρόβλημα: Σε κάποιο πρόγραμμά σου έχεις δηλώσει:

```
int k;
```

και στη συνέχεια, την εντολή:

```
cin >> k;
```

Όταν η εντολή αυτή εκτελείται, ας πούμε ότι θέλεις να δώσεις την τιμή 375. Πιέζεις λοιπόν τα πλήκτρα <3>, το <7> και το <5>. Όπως ήδη ξέρεις, το *k* θα πάρει την τιμή 375, αλλά ας δούμε πώς.

Αυτό που «φεύγει» από το πληκτρολόγιο προς τον υπολογιστή είναι η πληροφορία ότι πιέσθηκε το πλήκτρο <3> και στη συνέχεια το <7> και τέλος το <5>. Μέχρι εδώ λοιπόν το πρόγραμμά σου έχει «παραλάβει» τρεις *χαρακτήρες*, τρεις τιμές τύπου `char`. Πώς θα βγει τώρα το 375; Αυτό θα γίνει, με την εκτέλεση μερικών εντολών που παίρνουν ως στοιχεία εισόδου τους τρεις χαρακτήρες και βγάζουν ως αποτέλεσμα την τιμή 375 στη θέση μνήμης *k* του προγράμματός μας. Στη συνέχεια με ένα απλοϊκό προγραμματάκι θα προσπαθήσουμε να σου δείξουμε πώς γίνεται αυτή η δουλειά.

Κατ' αρχάς να επαναλάβουμε τον περιορισμό που είπαμε για τα σύνολα χαρακτήρων: «Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη». Π.χ. στον ASCII οι θέσεις αυτές είναι από το 48 (`'0'`) μέχρι το 57 (`'9'`). Έτσι, αν από τη σειρά ενός ψηφίου στον πίνακα αφαιρέσουμε την σειρά που έχει ο χαρακτήρας '0' παίρνουμε την αριθμητική τιμή του ψηφίου αυτού. Π.χ. (στο ASCII):

`static_cast<int>('7')-static_cast<int>('0')` = 55 − 48 = 7

Με βάση τις παραπάνω παρατηρήσεις, μπορούμε να δούμε πώς περίπου διαβάζει τα αριθμητικά στοιχεία η C++. Ας πούμε ότι θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει τριψήφιους ακέραιους χωρίς πρόσημο· ή καλύτερα, θα διαβάζει τρεις χαρακτήρες - ψηφία και θα τους μεταφράζει σε ακέραιο αριθμό. Από τα τρία ψηφία: το πρώτο θα είναι το ψηφίο των εκατοντάδων, το δεύτερο των δεκάδων και το τρίτο των μονάδων. Αφού υπολογίσουμε την τιμή που αντιστοιχεί σε κάθε ψηφίο, τις προσθέτουμε πολλαπλασιασμένες με την αντίστοιχη δύναμη του 10. Να το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int ord0 = static_cast<int>('0');
// η θέση του '0' στο σύνολο χαρακτήρων
```

```
   unsigned char  digit1, digit2, digit3;
// τρία ψηφία (χαρακτήρες) που διαβάζονται από το πληκτρολόγιο
   unsigned char  ceol;  // εδώ θα πάει το <enter>
   int value1, value2, value3;
// οι ακέραιοι που αντιστοιχούν στα τρία ψηφία
   int integer;
// Ο ακέραιος που φιλοδοξούμε να διαβάσουμε

   cin.get(digit1);  cin.get(digit2);  cin.get(digit3);
   cin.get(ceol);
   value1 = static_cast<int>(digit1) - ord0;
   value2 = static_cast<int>(digit2) - ord0;
   value3 = static_cast<int>(digit3) - ord0;
   integer = value1*100 + value2*10 + value3;
   cout << " Ακέραιος = " << integer << endl;
}
```

Αν δώσουμε σ' αυτό το πρόγραμμα:

375

θα πάρουμε απάντηση:

```
 Ακέραιος = 375
```

Αν δώσουμε:

007

θα πάρουμε:

```
 Ακέραιος = 7
```

Το πρόγραμμα περιμένει **τρία δεκαδικά ψηφία.** Με διαφορετικά στοιχεία εισόδου δεν θα πάρεις την απάντηση που περιμένεις. Αν δώσουμε:

␣␣*7*

θα πάρουμε λάθος απάντηση:

```
 Ακέραιος = -1753
```

Και η εκτύπωση πώς γίνεται; Η οθόνη σου ή ο εκτυπωτής σου περιμένουν χαρακτήρες. Αν θέλουμε να γράψουμε τον αριθμό 375 θα πρέπει πρώτα να γράψουμε το ψηφίο των εκατοντάδων (`'3'`), μετά το ψηφίο των δεκάδων (`'7'`) και τέλος το ψηφίο των μονάδων (`'5'`).

Οι εκατοντάδες υπολογίζονται ως πηλίκο της *ακέραιης* διαίρεσης 375 `/` 100. Αν πάρουμε το υπόλοιπο αυτής της διαίρεσης (375 `%` 100 `==` 75) και το διαιρέσουμε δια 10, το πηλίκο (75 `/` 10) μας δίνει τις δεκάδες, ενώ το υπόλοιπο (75 `%` 10) μας δίνει τις μονάδες.

Οι παρακάτω εντολές κάνουν αυτή τη δουλειά για τριψήφιους μη-αρνητικούς ακέραιους.

```
   cin >> integer;
   value1 = integer / 100;
   digit1 = static_cast<unsigned char>(ord0 + value1);
   integer = integer % 100;  value2 = integer / 10;
   digit2 = static_cast<unsigned char>(ord0 + value2);
   integer = integer % 10;  value3 = integer;
   digit3 = static_cast<unsigned char>(ord0 + value3);
   cout << digit1 << digit2 << digit3 << endl;
```

Η C++ σου δίνει συναρτήσεις για επεξεργασία τιμών τύπου `char`. Στον Πίν. 4-3 βλέπεις μερικές από αυτές, που σου επιτρέπουν να δεις τι είδους είναι κάποιος χαρακτήρας.

Οι συναρτήσεις επιστρέφουν τιμή τύπου `int` για συμβατότητα με τη C, από την οποίαν και κληρονομήθηκαν. Θα τις χειριζόμαστε ως συναρτήσεις τύπου `bool` (κατηγορήματα). Όταν τις χρησιμοποιείς θα πρέπει να βάζεις στο πρόγραμμά σου την οδηγία: "`#include <cctype>`". Ας δούμε ένα παράδειγμα. Το πρόγραμμα:

```
#include <iostream>
#include <cctype>
```

Όνομα	Δίνει αποτέλεσμα `true` (≠ 0) αν το όρισμα είναι:
`isalpha`	Λατινικό γράμμα (`'a'..'z'`, `'A'..'Z'`)
`isupper`	Κεφαλαίο λατινικό γράμμα (`'A'..'Z'`)
`islower`	Πεζό λατινικό γράμμα (`'a'..'z'`)
`isdigit`	Δεκαδικό ψηφίο (`'0'..'9'`)
`isxdigit`	Δεκαεξαδικό ψηφίο (`'0'..'9'`, `'a'..'f'`, `'A'..'F'`)
`isspace`	Κάποιος από τους `' '`, `'\t'`, `'\r'`, `'\n'`, `'\f'`
`iscntrl`	Χαρακτήρας ελέγχου: `static_cast<char>(0) .. static_cast<char>(31)`, `static_cast<char>(127)`
`ispunct`	Τίποτε από τα παραπάνω
`isalnum`	Λατινικό γράμμα ή δεκαδικό ψηφίο
`isprint`	Εκτυπώσιμος: `static_cast<char>(32)` (= `' '`) .. `static_cast<char>(126)` (= `'~'`)
`isgraph`	`isalpha || isdigit || ispunct`
`isascii`	Χαρακτήρας ASCII (ISO 646) `static_cast<char>(0) .. static_cast<char>(127)`

Πίν. 4-3 Συναρτήσεις της C++ για επεξεργασία χαρακτήρων. Παίρνουν ένα όρισμα τύπου `char`. Επιστρέφουν τιμή `true` αν το όρισμα έχει την ιδιότητα που περιγράφεται στη δεύτερη στήλη. Αλλιώς `false`. Για να τις χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου: `#include <ctype>`.

```
using namespace std;
int main()
{
   char c1 = 'f', c2 = 'ϕ', c3 = '3';

   cout << isalpha(c1) << "  " << isalpha(c2) << endl;
   cout << (isxdigit(c1) && !isdigit(c1)) << endl;
   cout << (isxdigit(c3) && !isdigit(c3)) << endl;
}
```

μας δίνει:

```
8  0
1
0
```

Το `'f'` είναι γράμμα του λατινικού αλφαβήτου και η `isalpha(c1)` μας δίνει τιμή `8` ≠ 0 δηλαδή `true`. Το `'ϕ'` δεν είναι γράμμα του λατινικού αλφαβήτου· η `isalpha(c2)` μας δίνει τιμή `0` δηλαδή `false`. Στη συνέχεια ζητάμε την τιμή της πρότασης «το `'f'` είναι δεκαεξαδικό ψηφίο και το `'f'` δεν είναι δεκαδικό ψηφίο». Η απάντηση είναι `true` (1). Αλλά, στη συνέχεια μας λέει ότι η τιμή της πρότασης «το `'3'` είναι δεκαεξαδικό ψηφίο και το `'3'` δεν είναι δεκαδικό ψηφίο» είναι `false` (0).

Α, και κάτι ακόμη: αν σε ενοχλεί το 8 άλλαξε την πρώτη εντολή σε:

```
   cout << static_cast<bool>(isalpha(c1)) << "  "
        << static_cast<bool>(isalpha(c2)) << endl;
```

Πάντως, οι συνθήκες με τις λογικές πράξεις υπολογίζονται σωστά· δεν υπάρχει πρόβλημα.

Στο `cctype` δηλώνεται ακόμη η συνάρτηση *tolower* που αν τροφοδοτηθεί με κεφαλαίο γράμμα του λατινικού αλφαβήτου μας δίνει το αντίστοιχο πεζό, π.χ. η `tolower('Q')` θα δώσει `'q'`. Η *toupper*, που δηλώνεται επίσης στο `cctype`, αν τροφοδοτηθεί με πεζό θα μας δώσει το αντίστοιχο κεφαλαίο, π.χ. η `toupper('q')` θα δώσει `'Q'`.

4.7 Ο Τύπος wchar_t

Ο τύπος **wchar_t** είναι ένα εργαλείο της C++[5] που χρησιμοποιείται για να αντιμετωπισθεί το πρόβλημα της παράστασης στον υπολογιστή πολλών χαρακτήρων. Έχει 65536 διαφορετικές τιμές, αντί για τις 256 του **char** και χρησιμοποιείται για την υλοποίηση του προτύπου Unicode.

Μια σταθερά τύπου **wchar_t** είναι μια σταθερά τύπου **char** με το πρόθεμα **L**, π.χ.:

```
L'a'   L'%'  L'ξ'
```

Όπως λέγαμε και στον Πίν. 2-1, υλοποιείται σε 16 δυαδικά ψηφία (2 ψηφιολέξεις). Μπορείς να δοκιμάσεις την:

```
cout << sizeof('A') << "  " << sizeof(L'A') << endl;
```

που θα σου δώσει:

```
1  2
```

Δεν μπορείς να διαβάσεις τιμές τύπου **wchar_t** από το *cin* ενώ μπορείς να στείλεις τιμές στο ρεύμα *cout*, αλλά θα δεις να τυπώνονται οι αντίστοιχοι αριθμοί. Π.χ. αν

```
wchar_t a, b;
```

τότε οι

```
a = L'a';  b = L'\n';
cout << a << "  " << b << endl;
```

θα δώσουν:

```
97  10
```

Αν περιλάβεις στο πρόγραμμά σου το **cwctype**[6] μπορείς να χρησιμοποιήσεις τις συναρτήσεις επεξεργασίας τιμών τύπου **wchar_t**, αντίστοιχες αυτών που έχουμε στον Πίν. 4-3. Τα ονόματά τους διαφέρουν κατά το ότι αντί για **is** έχουν πρόθεμα **isw**. Π.χ., για τις παραπάνω τιμές των *a*, *b*, η

```
cout << iswalnum(a) << "  " << iswprint(b)  << endl;
```

θα δώσει:

```
256  0
```

(ο **L'a'** είναι αλαφαριθμητικός ενώ ο **L'\n'** δεν είναι εκτυπώσιμος.)

Παρατήρηση: ▶

Σύμφωνα με αυτά που είπαμε, στην *a* μπορούμε να αποθηκεύσουμε το ελληνικό γράμμα "πεζό άλφα" στην κωδικοποίηση Unicode, που είναι η τιμή **0x03b1** (945) γράφοντας "**a = 0x03b1**" ή "**a = 945**". Όχι όμως γράφοντας "**a = L'α'**"· στην περίπτωση αυτή αποθηκεύεται η τιμή 225 (Windows).◀

4.8 Τακτικοί Τύποι

Ξεκινούμε με την εξής παρατήρηση:

- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **bool** στο υποσύνολο { 0, 1 } του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **unsigned char** στο υποσύνολο 0 .. 255 του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές των τύπων **signed char** και **char** στο υποσύνολο −128 .. 127 του **int**.

---

[5] Στη C ορίζεται στο stddef.h ως (Borland C):
```
typedef unsigned short wchar_t;
```

[6] Αν δεν το έχεις δοκίμασε το **cctype**.

Η απεικόνιση –και στις τρεις περιπτώσεις– γίνεται με την `int`. Οι τύποι αυτοί λέγονται **τακτικοί τύποι** (ordinal types)[7]. Φυσικά και ο `int`, που μπορεί να απεικονισθεί στον εαυτό του, είναι επίσης τακτικός τύπος. Και στην περίπτωση αυτή η απεικόνιση γίνεται με την `int`, αλλά, φυσικά για τον `int`, η `int` απεικονίζει κάθε αριθμό στον εαυτό του.

Οι τακτικοί τύποι είναι **διαταγμένοι** (ordered). Για κάθε τιμή ενός τακτικού τύπου υπάρχει μια **προηγούμενη** (predecessor) και μια **επόμενη** (successor). Π.χ. προηγούμενος του 0 είναι ο –1 και επόμενος ο 1, προηγούμενος του `'C'` είναι ο `'B'` και επόμενος ο `'D'`. Φυσικά, δεν υπάρχει επόμενος του *INT_MAX* και προηγούμενος του *INT_MIN*. Στον τύπο `char` δεν υπάρχει προηγούμενος του `char(-128)` και επόμενος του `char(127)`. Στον `bool` δεν υπάρχει προηγούμενη της `false` ούτε επόμενη της `true`.

4.9 Απαριθμητοί Τύποι (που Ορίζονται από τον Χρήστη)

Η C++ δίνει τη δυνατότητα στον προγραμματιστή να ορίζει δικούς του τύπους. Μια τέτοια κατηγορία τύπων είναι και τακτικοί ή **απαριθμητοί τύποι** (enumerated types). Μπορείς, για παράδειγμα, στο πρόγραμμά σου να ορίσεις:

```
   enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
                  friday, saturday };
   enum EurUn { Austria, Belgium, Bulgaria, Cyprus, CzechRepublic,
                Denmark, Ellas, Estonia, Finland, France, Germany,
                Hungary, Ireland, Italy, Latvia, Lithuania,
                Luxembourg, Malta, Netherlands, Poland, Portugal,
                Romania, Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom };
```

στη συνέχεια να δηλώσεις:

```
   WeekDay day1, day2;
   EurUn country;
```

και να δώσεις τιμές:

```
   day1 = sunday;  day2 = wednesday;
// . . .
   country = Ellas;
```

Ο ορισμός ενός απαριθμητού τύπου αρχίζει με το λεξικό σύμβολο "`enum`". Στη συνέχεια γράφουμε το όνομα του τύπου και μετά, μέσα σε άγκιστρα, τα *ονόματα των τιμών* που περιλαμβάνονται στον τύπο.

Φυσικά, ο ορισμός τύπου μπαίνει, στο πρόγραμμά μας, *πριν* από τις δηλώσεις μεταβλητών.

Ο ορισμός του τύπου είναι συγχρόνως και ορισμός των τιμών που περιέχει. Για παράδειγμα, οι τιμές που μπορεί να πάρει κάθε μεταβλητή τύπου *WeekDay*, όπως η *day*1, είναι οι: *sunday*, *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday*.

Οι τιμές του κάθε τύπου είναι διαταγμένες σύμφωνα με τον ορισμό του τύπου. Π.χ.:

sunday < *monday* < *tuesday* < *wednesday* < *thursday* < *friday* < *saturday*

Austria < *Belgium* < *Bulgaria* < . . . < *Sweden* < *UnitedKingdom*

Η `static_cast<int>` δέχεται ως όρισμα τιμή τέτοιων τακτικών τύπων και μας δίνει ως τιμή τη θέση (τάξη) του ορίσματος στον τύπο:

`static_cast<int>(sunday) == 0`,
`static_cast<int>(monday) == 1`,
`static_cast<int>(tuesday) == 2` κ.ο.κ.
`static_cast<int>(Austria) == 0`,
`static_cast<int>(Belgium) == 1`,
`static_cast<int>(Denmark) == 2` κ.ο.κ.

[7] Ο όρος από την Pascal.

Μαζί με κάθε τέτοιο τύπο ορίζεται και η αντίστοιχη αντίστροφη συνάρτηση που μας δίνει τιμές του τύπου αυτού:

`static_cast<WeekDay>(0) == sunday`,
`static_cast<WeekDay>(1) == monday`,
`static_cast<WeekDay>(2) == tuesday` κ.ο.κ.
`static_cast<EurUn>(0) == Austria`,
`static_cast<EurUn>(1) == Belgium`,
`static_cast<EurUn>(2) == Bulgaria` κ.ο.κ.

Μπορείς να χρησιμοποιείς τέτοιες τιμές σε εντολές εξόδου· η "`cout << v …`" θα λειτουργήσει σαν την: "`cout << static_cast<int>(v)...`"

Δεν μπορείς να χρησιμοποιείς μεταβλητές τέτοιων τύπων σε εντολές εισόδου (`cin >> v`). Το πρόβλημα παρακάμπτεται με τη χρήση μιας μεταβλητής "`int k`" και με τις εξής εντολές:

```
cin >> k;
v = static_cast<T>( k );
```

όπου *T* ο τύπος της *v*.

Μπορείς αν θέλεις να αλλάζεις τις τιμές του `int` στους οποίους απεικονίζονται οι τιμές του τύπου σου. Π.χ. με τον ορισμό:

```
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };
```

θα έχεις:

```
static_cast<int>( zero) == 48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αν θέλεις μπορείς να δίνεις περισσότερα από ένα ονόματα στην ίδια τιμή. Με τους παρακάτω ορισμούς:

```
enum Digit { miden = 48, zero = 48, one, two, three, four,
             five, six, seven, eight, nine };
enum Digit { zero = 48, one, two, three, four,
             five, six, seven, eight, nine, miden = 48 };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) ==48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αλλά προσοχή· αν δώσεις:

```
enum Digit { zero = 48, one, two, three, four,
             miden = 48, five, six, seven, eight, nine };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) == 48,
static_cast<int>(one) == static_cast<int>(five) == 49,
static_cast<int>(two) == static_cast<int>(six) == 50,
. . .
```

4.10 Μετονομασία Τύπου

Θα μπορούσαμε να ορίσουμε τους τύπους *WeekDay* και *EurUn* και ως εξής:

```
typedef enum { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday }                        WeekDay;
typedef enum { Austria, Belgium, Bulgaria, Cyprus,
               CzechRepublic, Denmark, Ellas, Estonia, Finland,
               France, Germany, Hungary, Ireland, Italy,
               Latvia, Lithuania, Luxembourg, Malta,
```

```
                Netherlands, Poland, Portugal, Romania,
                Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom }                          EurUn;
```

Το **typedef** (*type def*inition) είναι λεξικό σύμβολο. Αν σκεφτούμε ότι ο τύπος είναι στην παργαματικότητα το **enum { sunday, monday, tuesday, wednesday, thursday, friday, saturday }**, η **typedef** είναι στην πραγματικότητα εντολή *μετονομασίας* τύπου. Με αυτήν μπορείς να μετονομάσεις και άλλους τύπους που είναι ήδη ορισμένοι. Π.χ.

```
typedef unsigned int       Natural;
typedef unsigned long int  LongNatural;
typedef unsigned char      byte;
typedef bool               Logical;
```

Οι «νέοι» τύποι, *Natural*, *LongNatural*, *Logical* και *byte*, είναι στην πραγματικότητα οι αρχικοί τύποι και, επομένως, έχουν όλες τις ιδιότητές τους. Μετά τους παραπάνω ορισμούς μπορείς να δηλώσεις:

```
Natural     n, eN;
LongNatural athr, aAth;
Logical     yparxei;
byte        c1, c2;
```

Ασκήσεις

Α Ομάδα

4-1 Εξήγησε πώς βγήκε το αποτέλεσμα –1753 όταν δώσαμε στο πρόγραμμα της §4.5, στοιχείο εισόδου: ␣␣**7**.

Β Ομάδα

4-2 Γράψε πρόγραμμα που θα αποδεικνύει την ισοδυναμία: ((*A* **||** *B*) **&&** *B*) ≡ *B*.

4-3 Το ίδιο για τις ταυτολογίες: (*A* **&&** *B*) ⇒ *A* και (*A* **&&** *B*) ⇒ *B*.

Γ Ομάδα

4-4 Με βάση το πρόγραμμα της §4.5, γράψε πρόγραμμα που θα διαβάζει *δυαδικούς ακέραιους*.

κεφάλαιο

5

Επιλογές

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κάνεις το πρώτο βήμα στη χρήση συνθηκών για τον έλεγχο εκτέλεσης ενός προγράμματος: Να μπορείς να επιλέγεις ομάδες εντολών που θα εκτελεσθούν ή δεν θα εκτελεσθούν.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις προγράμματά πιο ευέλικτα, που η πορεία της εκτέλεσής τους θα εξαρτάται από τιμές συνθηκών και των μεταβλητών που περιλαμβάνονται σε αυτές. Θα μπορείς να ελέγχεις αν ισχύει η προϋπόθεση όταν αρχίζει η εκτέλεση του προγράμματος.

Έννοιες κλειδιά:

- *εντολές επιλογής*
- *εντολή* `if`
- *εντολή* `ifelse`
- *σύνθετη εντολή*
- *πολλαπλές επιλογές*
- *λογική εντολών* `if` *και* `ifelse`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Οι εντολές που γνωρίσαμε στα προηγούμενα κεφάλαια, δηλ. οι εντολές εισόδου/εξόδου, οι εντολές δήλωσης και η εντολή εκχώρησης μας δίνουν τη δυνατότητα να γράφουμε απλά προγράμματα υπολογισμών, που όμως δεν έχουν δυνατότητες για επιλογή «διαδρομών», στη διάρκεια της εκτέλεσης. Αυτό σημαίνει ότι, όταν εκτελείται το πρόγραμμα, δεν υπάρχει δυνατότητα «παράκαμψης» κάποιων εντολών του προγράμματος και επομένως *όλες οι*

εντολές εκτελούνται υποχρεωτικά με την ίδια σειρά που είναι γραμμένες στο πρόγραμμα, η μια μετά την άλλη.

Φυσικά, αυτός ο περιορισμός δεν είναι βολικός στην πράξη και στις γλώσσες προγραμματισμού υπάρχουν ειδικές εντολές, που μας δίνουν τη δυνατότητα να ζητούμε την εκτέλεση ή την παράκαμψη εντολών που υπάρχουν στο πρόγραμμά μας.

Στη C++ τέτοιες εντολές είναι οι **εντολές επιλογής** (selection statements). Στο κεφάλαιο αυτό, θα ασχοληθούμε ειδικά με τέτοιες εντολές.

5.1 Επιλογή – Οι Εντολές if

Στο προηγούμενο κεφάλαιο λέγαμε ότι «*μπορούμε ... να ρυθμίζουμε την εκτέλεση του προγράμματός μας με βάση τις τιμές συνθηκών.*» Εδώ θα μάθουμε έναν τρόπο για να κάνουμε κάτι τέτοιο. Ξεκινούμε με ένα παράδειγμα.

Θέλουμε ένα πρόγραμμα που θα διαβάζει δύο πραγματικούς αριθμούς, από το πληκτρολόγιο, και θα γράφει τον μεγαλύτερο από αυτούς μαζί με το μήνυμα `" Μεγαλύτερος είναι ο αριθμός: "`. Ας πούμε ότι δηλώνουμε:

```
   double x, y, maxxy;
```

και διαβάζουμε:

```
   cin >> x >> y;
```

Στο τέλος θα γράψουμε:

```
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
```

Το πρόβλημά μας είναι να δώσουμε στη *maxxy* τη σωστή τιμή. Ένας απλός τρόπος, που θα σκέφτονταν πολλοί, είναι ο εξής:

αν $x > y$ **τότε** { **βάλε** *maxxy* ← *x*
αλλιώς { **βάλε** *maxxy* ← *y*

Τα «**βάλε** *maxxy* ← *x*» και «**βάλε** *maxxy* ← *y*» ξέρουμε να τα γράψουμε σε C++. Τα υπόλοιπα; Σχεδόν μεταφράζουμε στα αγγλικά:

```
   if ( x > y )  maxxy = x;
          else  maxxy = y;
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   if (x > y)  maxxy = x;
         else  maxxy = y;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

Η εντολή **ifelse** της C++ έχει γενικώς τη μορφή:

```
if ( S )  Et  else  Ef
```

όπου *S* μια συνθήκη (λογική παράσταση) και E_t, E_f εντολές. Η εντολή εκτελείται ως εξής:

- υπολογίζεται η τιμή της λογικής παράστασης *S*,
- αν η τιμή είναι `true` (αν η συνθήκη ισχύει) τότε α) εκτελείται η εντολή E_t και στη συνέχεια β) αγνοείται η εντολή E_f και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**,
- αν η τιμή είναι `false` (αν η συνθήκη δεν ισχύει) τότε α) αγνοείται η εντολή E_t, β) εκτελείται η εντολή E_f και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**.

Αυτά φαίνονται και στο λογικό διάγραμμα της **ifelse** στο Σχ. 5-1. Ο ρόμβος δείχνει το σχήμα «απόφασης» –επιλογής διαδρομής– που λαμβάνεται ανάλογα με την τιμή (`true` ή `false`) της παράστασης *S*. Από το σχήμα μπορείς να καταλάβεις ακόμη γιατί οι εντολές **ifelse** (και οι **if** που θα δούμε στη συνέχεια) λέγονται και εντολές **διακλάδωσης** (branching).

Ερχόμαστε στο παράδειγμά μας και ας πούμε ότι όταν εκτελείται η "`cin >> x >> y`" απαντούμε με:

```
1.6   0.6
```

Το "`1.6`" γίνεται τιμή της *x* και το "`0.6`" γίνεται τιμή της *y*. Στη συνέχεια εκτελείται η **ifelse** και κατ' αρχάς υπολογίζεται η συνθήκη "`x > y`" ή "1.6 > 0.6", που είναι `true`. Ύστερα από αυτό εκτελείται η εντολή "`maxxy = x`" και στη συνέχεια η εντολή που ακολουθεί την **ifelse**, δηλαδή η "`cout << ...`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 1.6
```

Η "`maxxy = y`" είναι σαν να μην υπάρχει.

Αν στη "`cin >> ...`" απαντήσουμε με:

```
1.6   2.6
```

η "`x > y`" (1.6 > 2.6) θα πάρει τιμή `false`. Στην περίπτωση αυτή θα εκτελεσθεί η εντολή "`maxxy = y`" και στη συνέχεια η "`cout << ...`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 2.6
```

Το ίδιο θα συμβεί και στην περίπτωση που απαντούμε στη "`cin >> ...`" με:

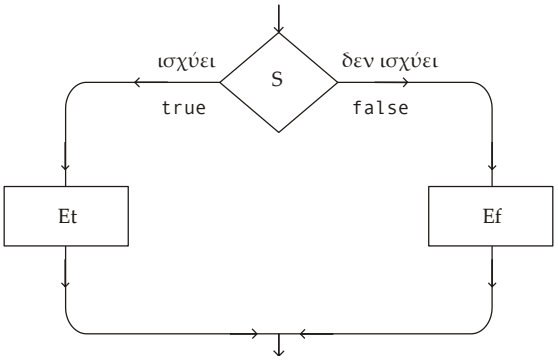
```
2.6   2.6
```

αφού και πάλι η "`x > y`" (2.6 > 2.6) θα πάρει τιμή `false`.

Ας ξαναδούμε τώρα το παράδειγμα 3 της §2.3. Κάναμε ένα πρόγραμμα που έβρισκε τις ρίζες της εξίσωσης $ax^2 + bx + c = 0$. Αλλά όταν η διακρίνουσα ήταν αρνητική είχαμε πρόβλημα. Τώρα μπορούμε να κάνουμε ένα καλύτερο πρόγραμμα: οι εντολές

```
x1 = (-b + sqrt(delta))/(2*a);
x2 = (-b - sqrt(delta))/(2*a);
cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
```

θα πρέπει να εκτελούνται μόνον αν *delta* ≥ 0 αλλιώς θα πρέπει να βγαίνει ένα μήνυμα που



Σχ. 5-1 Λογικό διάγραμμα για την εντολή **ifelse**.

θα λέει ότι η εξίσωση δεν έχει πραγματικές λύσεις. Θα βάλουμε λοιπόν μια **ifelse**, αλλά… Μάθαμε παραπάνω ότι μετά τη συνθήκη μπορούμε να γράψουμε μια εντολή και εδώ έχουμε τρεις· τι κάνουμε τώρα;

Η C++ μας προσφέρει την εξής δυνατότητα: Να δημιουργήσουμε μια **σύνθετη εντολή** (compound statement) που θα περιλαμβάνει τις τρεις εντολές που μας ενδιαφέρουν. Αυτό γίνεται ως εξής: «συσκευάζουμε» τις εντολές μας με ένα άγκιστρο (**{**) στην αρχή και ένα άγκιστρο (**}**) στο τέλος. Στην περίπτωσή μας:

```
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
```

Να πώς γίνεται το πρόγραμμά μας:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;    // οι συντελεστές της εξίσωσης
   double  x1, x2;     // οι ρίζες της εξίσωσης
   double  delta;      // η διακρίνουσα

   cout << "Δώσε τρεις πραγματικούς αριθμούς" << endl;
   cin >> a >> b >> c;
   cout << " a = " << a << "   b = " << b
        << "   c = " << c << endl;
   delta = b*b - 4*a*c;
   if ( delta >= 0 )
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
   else
      cout << " Δεν υπάρχουν Πραγματικές Λύσεις" << endl;
   cout << " ΤΕΛΟΣ" << endl;
}
```

Αν τροφοδοτήσουμε το πρόγραμμα με τους παρακάτω τρείς αριθμούς (συντελεστές της εξίσωσης):

```
            1   3   -10
```

θα πάρουμε το εξής αποτέλεσμα:

```
 a = 1   b = 3   c = -10
 Λύση1 = 2   Λύση2 = -5
 ΤΕΛΟΣ
```

Αν όμως πληκτρολογήσουμε τους αριθμούς:

```
            2   3   4
```

θα πάρουμε τα εξής:

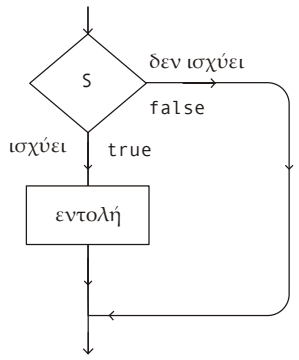
```
 a = 2   b = 3   c = 4
 Δεν υπάρχουν Πραγματικές Λύσεις
 ΤΕΛΟΣ
```

Ζητάμε άλλη μια εκτέλεση, δίνοντας για στοιχεία εισόδου:

```
    0   1   2
```

Αποτέλεσμα: διακοπή εκτέλεσης του προγράμματος και κάποιο «κακό» μήνυμα. Γιατί; Το "**0**" έγινε τιμή της *a* και στον υπολογισμό των *x1* και *x2* διαιρούμε δια 2*a*. Χρειαζόμαστε λοιπόν άλλον έναν έλεγχο, άλλη μια **if**. Θα επανέλθουμε στη συνέχεια.

5.2 Η Εντολή if



Σχ. 5-2 Λογικό διάγραμμα για την εντολή **if**.

Ξαναγυρνάμε στο πρόβλημα του μεγίστου· πολλοί θα προτιμούσαν να δώσουν μια κάπως διαφορετική λύση:

αν *x* > *y* **τότε** { **βάλε** *maxxy* ← *x*

αν *x* ≤ *y* **τότε** { **βάλε** *maxxy* ← *y*

Αυτό πώς το γράφουμε σε C++; Μια πρώτη σκέψη είναι η εξής:

```
if (x > y)  maxxy = x; else;
if (x <= y)  maxxy = y; else;
```

που στην πραγματικότητα μεταφράζει το:

αν *x* > *y* **τότε** { **βάλε** *maxxy* ← *x*

αλλιώς { μη κάνεις τίποτε

αν *x* ≤ *y* **τότε** { **βάλε** *maxxy* ← *y*

αλλιώς { μη κάνεις τίποτε

Μετά το `else` θεωρείται ότι υπάρχει η **κενή** (empty) ή **μηδενική** (null) εντολή που αντιπροσωπεύει στη C++ αυτό το «*μη κάνεις τίποτε*» του ψευδοκώδικα.

Πάντως μας δίνεται και άλλη δυνατότητα, χωρίς εκείνο το "`else;`":

```
if (x > y)  maxxy = x;
if (x <= y)  maxxy = y;
```

Εδώ χρησιμοποιούμε την απλή **if** που έχει γενικώς τη μορφή:

`if ( S )  Et`

όπου *S* μια συνθήκη (λογική παράσταση) και E_t εντολή. Η εντολή εκτελείται ως εξής:

- υπολογίζεται η τιμή της λογικής παράστασης *S*,
- αν η τιμή είναι `true` (αν η συνθήκη ισχύει) τότε α) εκτελείται η εντολή E_t και στη συνέχεια β) η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **if**,
- αν η τιμή είναι `false` (αν η συνθήκη δεν ισχύει) τότε αγνοείται η εντολή E_t και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί την **ifelse**.

Στο Σχ. 5-2 βλέπεις και το λογικό διάγραμμα της **if**.

Να πώς γράφεται το πρόγραμμά μας:

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   if (x > y)  maxxy = x;
   if (x <= y)  maxxy = y;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

και να πώς εκτελείται. Στη `cin >> x >> y` απαντούμε με:

```
1.6   0.6
```

Το 1.6 γίνεται τιμή της *x* και το 0.6 γίνεται τιμή της *y*. Στη συνέχεια εκτελείται η "`if (x > y) maxxy = x`" και πρώτα απ' όλα υπολογίζεται η συνθήκη "`x > y`" ή "1.6 > 0.6", που είναι `true`. Ύστερα από αυτό εκτελείται η εντολή "`maxxy = x`" και στη συνέχεια η εντολή που ακολουθεί την **if**, δηλαδή η "`if (x <= y) maxxy = y`". Και εδώ υπολογίζεται η συνθήκη "`x <= y`" ή 1.6 <= 0.6 που είναι `false`· έτσι η "`maxxy = y`" *δεν* εκτελείται και πηγαίνουμε στην "`cout << ...`" που δίνει:

```
 Μεγαλύτερος είναι ο αριθμός: 1.6
```

Η "`maxxy = y`" είναι σαν να μην υπάρχει.

Αν στη "`cin >> …`" απαντήσουμε με:

Πλαίσιο 5.1

Η Εντολή if

"`if`", "`(`", *συνθήκη*, "`)`", *εντολή*

Συστατικά:

- το `if` είναι λεξικό σύμβολο,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή* είναι μια οποιαδήποτε εντολή της C++.

Εκτέλεση της εντολής **if**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη

Αν η τιμή είναι `false` η εντολή που ακολουθεί τη συνθήκη αγνοείται.

Η Εντολή ifelse

"`if`", "`(`", *συνθήκη*, "`)`", *εντολή*-1, "`else`", *εντολή*-2

Συστατικά:

- τα `if` και `else` είναι λεξικά σύμβολα,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή*-1 (περιοχή του if) είναι μια οποιαδήποτε εντολή της C++.
- η *εντολή*-2 (περιοχή του else) είναι μια οποιαδήποτε εντολή της C++.

Εκτέλεση της εντολής **ifelse**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η *εντολή*-1 που βρίσκεται μεταξύ της συνθήκης και του **else**.

Η *εντολή*-2 αγνοείται.

αλλιώς (αν η τιμή είναι `false`)

εκτελείται η *εντολή*-2 που βρίσκεται μετά το **else**.

Η *εντολή*-1 αγνοείται.

1.6 2.6

η "`x > y`" (1.6 > 2.6) θα πάρει τιμή `false` και η "`maxxy = x`" δεν θα εκτελεσθεί. Στην επόμενη **if** η συνθήκη "`x <= y`" ή 1.6 <= 2.6 είναι `true`· στην περίπτωση αυτή θα εκτελεσθεί η εντολή "`maxxy = y`" και στη συνέχεια η "`cout << …`" που δίνει:

```
Μεγαλύτερος είναι ο αριθμός: 2.6
```

Πριν προχωρήσουμε ας δούμε έναν ακόμη τρόπο για να λύσουμε το πρόβλημα του μεγίστου· ο τρόπος αυτός θα μας είναι πολύ χρήσιμος στη συνέχεια.

```
#include <iostream>
using namespace std;
int main()
{
   double x, y, maxxy;

   cin >> x >> y;
   maxxy = y;
   if (x > maxxy)  maxxy = x;
   cout << " Μεγαλύτερος είναι ο αριθμός: " << maxxy << endl;
}
```

Εδώ *θεωρούμε αυθαιρέτως* ότι ο μεγαλύτερος αριθμός είναι ο δεύτερος (δηλ. ο *y*). Στην επόμενη εντολή όμως ελέγχουμε μήπως έχουμε κάνει λάθος (`x > maxxy`)· στην περίπτωση αυτή κάνουμε τη διόρθωση: `maxxy = x`.

Από τους τρεις τρόπους, ο πρώτος είναι ο πιο οικονομικός. Ας δούμε γιατί.

Στο πρώτο πρόγραμμα θα γίνει μια σύγκριση –η `x > y`– και αναλόγως θα εκτελεσθεί μια μόνο από τις εντολές: "`maxxy = x`" ή "`maxxy = y`". Δηλαδή θα έχουμε 1 σύγκριση και 1 εκχώρηση.

Στο δεύτερο θα εκτελεστούν και οι δυο συγκρίσεις (`x > y`, `x <= y`), μια και θα εκτελεσθούν και οι δυο εντολές **if**. Επειδή όμως οι δυο συνθήκες είναι αντίθετες, η μια θα είναι `true` και η άλλη `false`. Έτσι, από τις εντολές "`maxxy = x`" και "`maxxy = y`" θα εκτελεστεί μόνο η μια. Έχουμε λοιπόν: 2 συγκρίσεις και 1 εκχώρηση.

Στην τρίτη περίπτωση η εκχώρηση "`maxxy = y`" θα γίνει οπωσδήποτε. Το ίδιο και η σύγκριση "`x > maxxy`". Η εκχώρηση "`maxxy = x`" θα εκτελεσθεί μόνο στην περίπτωση που $x > y$. Αν δεχτούμε ότι η πιθανότητα για κάτι τέτοιο είναι 50%, το πρόγραμμα αυτό θα εκτελεί 1 σύγκριση και 1.5 εκχώρηση κατά μέσο όρο.

5.3 Φωλιασμένες if – Πολλαπλές Επιλογές

Στις θέσεις των εσωτερικών εντολών της **ifelse** (ή της **if**) μπορεί να δοθεί, όπως ειπώθηκε, οποιαδήποτε εντολή της C++. Επομένως μπορεί να δοθεί μια άλλη εντολή **if**, όπως δείχνει το παρακάτω παράδειγμα:

```
if ( S1 )
   if ( S2 ) εντολή-1
        else εντολή-2
```

Εδώ μπορεί να αναρωτηθείς: που κολλάει το `else`; Στη C++ ισχύει η πρόσθετη σύμβαση που λέει ότι το παραπάνω γενικό σχήμα της **if** (μέσα στην **if**) είναι ισοδύναμο με το σχήμα:

```
if ( S1 )
{
   if ( S2 )  εντολή-1
        else  εντολή-2
}
```

Δηλαδή, η δεύτερη **if** είναι η περιοχή του **if** για την `if (S1)...`. Το `else` είναι της `if (S2)...` . Για να εκτελεσθεί η *εντολή*-1 θα πρέπει να είναι αληθείς και η *S*1 και η *S*2 συγχρόνως. Για να εκτελεσθεί η *εντολή*-2 θα πρέπει να είναι αληθής η *S*1 και να είναι ψευδής η *S*2.

Μπορούμε δηλαδή να πούμε γενικότερα ότι το λεξικό σύμβολο `else` (και η εντολή που το ακολουθεί) αναφέρεται πάντα στο αμέσως προηγούμενο `if`. Αν όμως θέλουμε η *εντολή*-2 να εκτελεστεί σύμφωνα με την τιμή λογικής παράστασης *S*1 μόνον, θα πρέπει να δώσουμε υποχρεωτικώς την παρακάτω μορφή:

```
if ( S1 )  {  if ( S2 )  εντολή-1  }
     else  εντολή-2
```

Στην περίπτωση που μια εντολή **if** εμφανίζεται μέσα σε μιά άλλη **if**, λέγεται **φωλιασμένη** (nested) εντολή **if**.

Με φωλιασμένες **if** λύνουμε προβλήματα στα οποία έχουμε να επιλέξουμε μεταξύ εντολών που είναι περισσότερες από δύο. Στην περίπτωση αυτή βάζουμε δύο ή περίσσότερες **ifelse** τη μία μέσα στην άλλη· συνήθως βάζουμε την επόμενη **ifelse** στην περιοχή **else** της προηγούμενης. Ας ξεκινήσουμε με το

Παράδειγμα 1

Η ΔΕΗ χρεώνει την κατανάλωση ημερήσιου ρεύματος ώς εξής[1]:

- Για τις πρώτες 800 *kWh* (0 .. 800]: 0.088 €/*kWh*
- για τις επόμενες 400 *kWh* (800 .. 1200]: 0.112 €/*kWh*

[1] Η χρέωση γίνεται έτσι, με κλιμακούμενη αύξηση. Για τις τιμές... μην τα συζητάς. Ας πούμε ότι είναι έτσι.

- για τις επόμενες 500 *kWh* (1200 .. 1700]: 0.136 €/*kWh*
- για τις υπόλοιπες *kWh* (1700 .. +∞): 0.28 €/*kWh*

Να γραφεί πρόγραμμα που θα διαβάζει την κατανάλωση σε *kWh* και θα βγάζει το κόστος της ενέργειας που καταναλώθηκε[2].

Για να δούμε τι μας λέει ο παραπάνω πίνακας:

```
αν κατανάλωση <= 800 τότε
  κόστος = κατανάλωση*0.088
αλλιώς (είναι πάνω από 800)
  αν 800 < κατανάλωση <= 1200 τότε
    κόστος = 800*0.088 + (κατανάλωση - 800)*0.112
  αλλιώς (είναι πάνω από 1200)
    αν 1200 < κατανάλωση <= 1700 τότε
      κόστος = 800*0.088 + 400*0.112 + (κατανάλωση - 1200)*0.136
    αλλιώς (είναι πάνω από 1700)
      κόστος = 800*0.088 + 400*0.112 + 500*0.136 +
                    (κατανάλωση - 1700)* 0.28
```

Ας μεταφράσουμε σε **ifelse** την πρώτη σύγκριση:

```
if ( consumption <= 800 )
  cost = consumption*0.088
else
   ...
```

Η περιοχή του **else** θα εκτελεσθεί αν και μόνον αν η "`consumption <= 800`" έχει τιμή `false` ή αλλιώς "`consumption > 800`". Αν λοιπόν γράψουμε την επόμενη **ifelse** ως εξής:

```
if ( 800 < consumption && consumption <= 1200 ) ...
```

η πρώτη σύγκριση (και η λογική πράξη "`&&`") είναι άχρηστη, αφού στη θέση που το βάλαμε έχει σίγουρα τιμή `true`. Στο παρακάτω πρόγραμμα μπορείς να δεις πώς θα γραφούν οι πολλαπλές επιλογές μας.

```
#include <iostream>
using namespace std;
int main()
{
   double  consumption;    // Κατανάλωση σε kWh
   double  cost;

   cout << " Δώσε την κατανάλωση σε kWh: ";  cin >> consumption;
   if (consumption <= 800)
      cost = consumption * 0.088;
   else if (consumption <= (800 + 400))
      cost = 800*0.088 + (consumption - 800)*0.112;
   else if (consumption <= (800 + 400 + 500))
      cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
   else
      cost = 800*0.088 + 400*0.112 + 500*0.136
             + (consumption - 1700)*0.28;
   cout << cost << endl;
}
```

Γενικώς μπορούμε να γράψουμε:

```
if (S1)  E1
else if (S2)  E2
else if (S3)  E3
   :
else if (Sn)  En
else En+1
```

που θα εκτελεσθεί ως εξής:

[2] Ούτε νυκτερινό, ούτε πάγιο, ούτε ΦΠΑ, ούτε ΕΡΤ, ούτε τίποτε άλλο. Μόνον ημερήσιο!

αν η S_1 έχει τιμή `true` τότε

θα εκτελεσθεί η E_1 και θα αγνοηθούν οι E_2, E_3,... E_n,

αλλιώς (S_1 έχει τιμή `false`) αν η S_2 έχει τιμή `true` τότε

θα εκτελεσθεί η E_2 και θα αγνοηθούν οι E_1, E_3,... E_n,

αλλιώς (S_1 και S_2 έχουν τιμή `false`) αν η S_3 έχει τιμή `true` τότε

θα εκτελεσθεί η E_3 και θα αγνοηθούν οι E_1, E_2,... E_n,

. . .

αλλιώς (S_1, S_2,... S_{n-1} έχουν τιμή `false`) αν η S_n έχει τιμή `true` τότε

θα εκτελεσθεί η E_n και θα αγνοηθούν οι E_1, E_2,... $E_{n\text{-}1}$,

αλλιώς (S_1, S_2,... S_n έχουν τιμή `false`)

θα εκτελεσθεί η E_{n+1} και θα αγνοηθούν οι E_1, E_2,... E_n.

Αλλά προσοχή! Η σειρά που θα βάλεις τις συνθήκες των **if** είναι ουσιαστική. Οι εντολές της k-οστής **if** θα εκτελεστούν αν ισχύει η

$$(\texttt{!}\ S_1)\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ (\texttt{!}\ S_{k-1})\ \texttt{\&\&}\ S_k$$

ή αλλιώς:

$$(\texttt{!}\ (S_1\ \texttt{||}\ \ldots\ \texttt{||}\ S_{k-1}))\ \texttt{\&\&}\ S_k$$

Καμιά φορά, αν γράφεις απρόσεκτα, αυτή μπορεί να είναι ισοδύναμη με `false`! Για να δεις ένα (χονδροειδές) παράδειγμα, στο τελευταίο πρόγραμμα βάλε:

```
if ( consumption <= (800 + 400 + 500) )
   cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
else if ( consumption <= (800 + 400) )
   cost = 800*0.088 + (consumption - 800)*0.112;
else if ( consumption <= 800 )
   cost = consumption * 0.088;
else
  :
```

Για να εκτελεσθεί η

```
   cost = 800*0.088 + (consumption - 800)*0.112;
```

θα πρέπει να έχουμε: "`!(consumption <= 1700) && consumption <= 1200`". Η τιμή αυτής της παράστασης είναι `false`, όποια τιμή και αν έχει η *consumption*. Παρομοίως, `false` είναι πάντοτε και η "`!(consumption <= 1700) && !(consumption <= 1200) && consumption <= 100`" και έτσι αποκλείεται να εκτελεσθεί η: "`cost = consumption*0.088`".

Μερικοί προτιμούν να λύνουν αυτό το πρόβλημα με πολλές ανεξάρτητες **if**. Π.χ. για το παράδειγμά μας θα γράψουν:

```
if ( consumption <= 800 )
   cost = consumption * 0.088;
if ( 800 < consumption && consumption <= 1200 )
   cost = 800*0.088 + (consumption - 800)*0.112;
if ( 1200 < consumption && consumption <= 1700 )
   cost = 800*0.088 + 400*0.112 + (consumption - 1200)*0.136;
if ( consumption < 1700 )
   cost = 800*0.088 + 400*0.112 + 500*0.136
          + (consumption - 1700)*0.28;
```

Φυσικά, το πρόγραμμα είναι μια χαρά. Ποιο είναι το πρόβλημά της; Κάνει πολλές πράξεις χωρίς λόγο.[3] Π.χ. αν του δώσεις κατανάλωση 50 τότε θα υπολογίσει τον λογαριασμό από την πρώτη **if**, αλλά συνέχεια θα υπολογίσει τις συνθήκες όλων των υπολοίπων **if** για να τις βγάλει `false` φυσικά.

[3] Χωρίς λόγο; Πρόσεξε ότι είναι σαφώς πιο σίγουρη από τις άλλες μορφές.

Παράδειγμα 2

Θέλουμε να γράψουμε ένα πρόγραμμα που θα προσομοιώνει τη λειτουργία ενός απλού υπολογιστή τσέπης (pocket calculator). Θα διαβάζει μια γραμμή όπου στην *πρώτη* θέση θα δίνεται το σύμβολο της πράξης (`+`, `-`, `*`, `/`) και στη συνέχεια δυο πραγματικοί αριθμοί με τους οποίους θα γίνει η πράξη.

Θα αποθηκεύουμε το σύμβολο της πράξης σε μια μεταβλητή τύπου `unsigned char` με το όνομα *oper*. Τις δυο πραγματικές τιμές θα τις αποθηκεύουμε σε δυο μεταβλητές *x*1, *x*2, τύπου `double`.

Αφού διαβάσαμε μια γραμμή θα πρέπει να ελέγξουμε την *oper*:

```
αν oper == '+' τότε
  Δώσε το άθροισμα
αλλιώς, η oper μπορεί να είναι '-', '*', '/'
  αν oper == '-' τότε
    Δώσε τη διαφορά
  αλλιώς, η oper μπορεί να είναι '*', '/'
    αν oper == '*' τότε
      Δώσε το γινόμενο
    αλλιώς, η oper μπορεί να είναι '/'
      Δώσε το πηλίκο
```

Μεταφράζοντας τα παραπάνω σε C++ θα έχουμε δυο **ifelse** που η κάθε μια τους έχει σαν περιοχή του else μια άλλη **ifelse**. Αλλά μάλλον δεν τελειώσαμε: Θα πρέπει να προβλέψουμε την πιθανότητα λάθους στην πράξη. Δεν είναι καθόλου σίγουρο ότι αν η πράξη δεν είναι '`+`', '`-`', '`*`' θα είναι '`/`'. Μπορεί ο χρήστης να δώσει κατά λάθος '`@`', ας πούμε. Θα πρέπει λοιπόν να ελέγχουμε την πράξη και για τη διαίρεση. Τέλος, δεν θα πρέπει να προσπαθήσουμε να υπολογίσουμε το πηλίκο αν ο διαιρέτης είναι "`0`" (μηδέν). Να το πρόγραμμά μας τελικώς:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;          // Τα ορίσματα της πράξης
   unsigned char oper;     // Το σύμβολο της πράξης

   cin >> oper >> x1 >> x2;
   if (oper == '+')        cout << (x1 + x2) << endl;
   else if (oper == '-')   cout << (x1 - x2) << endl;
   else if (oper == '*')   cout << (x1 * x2) << endl;
   else if (oper == '/')
      if (x2 != 0)          cout << (x1 / x2) << endl;
      else
         cout << " Δεν γίνεται" << endl;
   else
      cout << " Η πράξη (+,-,*,/) στην πρώτη θέση" << endl;
}
```

5.4 * Η Λογική των Εντολών Επιλογής

Ας δούμε τώρα τα εργαλεία που μας χρειάζονται για να κάνουμε αποδείξεις ορθότητας προγραμμάτων με εντολές **ifelse** και **if**.

Υποθέτουμε ότι έχουμε την εξής περίπτωση:

```
// P
   if (S)  Et else Ef
```

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* `&&` *S* και έστω ότι καταλήγουμε στην συνθήκη *Qt*. Αν δεν ισχύει η *S* –δηλαδή ισχύει η `!`*S*– τότε εκτελείται η εντολή *Ef* με

προϋπόθεση *P* **&&** (**!***S*) και έστω ότι καταλήγουμε στην *Qf*. Άρα, η συνθήκη που θα καταλήξουμε μετά την εκτέλεση της **ifelse** θα είναι η: *Qt* **||** *Qf*. Βλέπουμε λοιπόν ότι:

- ***η ifelse είναι ένας τρόπος για να πάρουμε συνθήκες που να περιέχουν*** **||**.

Επειδή όμως, όπως ξέρουμε, δεν γράφουμε εντολές στην τύχη για να δούμε πού θα καταλήξουμε, αλλά τις γράφουμε για να καταλήξουμε σε κάποια συγκεκριμένη συνθήκη, ας δούμε το πρόβλημά μας κάπως διαφορετικά:

```
// P
   if (S)  Et else Ef                                    (1)
// Q
```

Δηλαδή: είμαστε στην κατάσταση *P* και γράφουμε την `if (S) Et else Ef` με στόχο να καταλήξουμε στην *Q*. Τι γίνεται τώρα;

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S*. Αν το πρόγραμμά μας είναι σωστό θα πρέπει να φτάνουμε στην *Q*. Αν πάλι δεν ισχύει η *S* –δηλαδή ισχύει η **!***S*– τότε εκτελείται η εντολή *Ef* με προϋπόθεση *P* **&&** (**!***S*). Αν το πρόγραμμά μας είναι σωστό θα πρέπει και πάλι να φτάνουμε στην *Q*. Δηλαδή:

- ***Για να αποδείξουμε την ορθότητα της* (1) *πρέπει και αρκεί να αποδείξουμε την ορθότητα των:***

```
      P && S { Et } Q
P && (!S) { Ef } Q
```

Αυτός είναι ο συμπερασματικός κανόνας της **ifelse**, που γράφεται συμβολικώς ως εξής:

$$\frac{P \,\&\&\, S\,\{Et\}\,Q,\; P \,\&\&\, (!S)\,\{Ef\}\,Q}{P\,\{\textbf{if}\ (S)\ Et\ \textbf{else}\ Ef\}\,Q}$$

Παρομοίως ισχύουν για την **if**. Αν έχουμε:

```
// P
   if (S)  Et
```

τότε: αν μεν ισχύει η *S* εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S* και έστω ότι καταλήγουμε στην συνθήκη *Qt*. Αν δεν ισχύει η *S* –δηλαδή ισχύει η **!***S*– τότε *δεν εκτελείται* η εντολή *Et* και παραμένει η *P* **&&** (**!***S*). Άρα, η συνθήκη που θα καταλήξουμε μετά την εκτέλεση της **if** θα είναι η: *Qt* **||** (*P* **&&** (**!***S*)).

Ας δούμε και τον συμπερασματικό κανόνα της **if**. Ας πούμε ότι έχουμε:

```
// P
   if (S)  Et                                            (2)
// Q
```

Αν ισχύει η *S* τότε εκτελείται η εντολή *Et* με προϋπόθεση *P* **&&** *S*. Αν το πρόγραμμά μας είναι σωστό θα πρέπει να φτάνουμε στην *Q*. Αν πάλι δεν ισχύει η *S* (ισχύει η **!***S*) τότε θα πρέπει να φτάνουμε στην *Q* χωρίς να εκτελεστεί οποιαδήποτε εντολή. Αφού λοιπόν ισχύει η *P* **&&** (**!***S*) θα πρέπει από αυτήν να συνάγεται η *Q*. Δηλαδή:

- ***Για να αποδείξουμε την ορθότητα της* (2) *πρέπει και αρκεί να αποδείξουμε την ορθότητα των:***

```
P && S { Et } Q
(P && (!S)) ⇒ Q
```

Αυτός είναι ο συμπερασματικός κανόνας της **if**. Συμβολικώς:

$$\frac{P \,\&\&\, S\,\{Et\}\,Q,\; (P \,\&\&\, (!S)) \Rightarrow Q}{P\,\{\textbf{if}\ (S)\ Et\}\,Q}$$

Παράδειγμα

Για παράδειγμα χρήσης του κανόνα της **ifelse** ας αποδείξουμε ότι στο πρώτο πρόγραμμα της §5.2 βρίσκουμε σωστά το μέγιστο. Φυσικά, θα πρέπει να ξεκινήσουμε με τις προδιαγραφές.

Ξεκινούμε με την απαίτηση· θέλουμε να έχουμε τη *maxxy* μεγαλύτερη (ή ίση) και από τη *x* και από τη *y*: (*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*). Αλλά η τιμή της *maxxy* θα είναι ίση είτε με *x* είτε με *y*: (*maxxy* **==** *x*) **||** (*maxxy* **==** *y*). Να λοιπόν η απαίτηση:

((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*))

Τι προϋπόθεση θα έχουμε; Τίποτε ιδιαίτερο. Θα πρέπει να μπορούμε να επεξεργαστούμε *οποιεσδήποτε* τιμές και αν πάρουμε –στις *x*, *y*– από τη "`cin >> x >> y`". Δηλαδή, όπως είπαμε στην §3.7, θα έχουμε για προϋπόθεση:

`true`

Να λοιπόν τι πρέπει να αποδείξουμε:

```
// true
   if (x > y)  maxxy = x;
         else  maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Σύμφωνα με τον κανόνα της **ifelse**, αρκεί να αποδείξουμε ότι:

```
// true && (x > y)
   maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

και

```
// true && (!(x > y))
   maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Αυτές όμως αποδεικνύονται πολύ εύκολα.

1: Για να ισχύει η ((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*)) μετά τη "`maxxy = x`" θα πρέπει πριν από αυτήν να ισχύει η: ((***x*** ≥ *x*) **&&** (***x*** ≥ *y*)) **&&** ((***x*** **==** *x*) **||** (***x*** **==** *y*)), που παίρνουμε από την απαίτηση αν βάλουμε όπου *maxxy* το *x*. Αν πάρουμε υπόψη μας ότι

- ότι οι ***x*** ≥ *x* και ***x*** **==** *x* έχουν τιμή `true` για οποιαδήποτε τιμή της *x*,
- ότι (`true` **&&** A) ≡ A,
- ότι (`true` **||** A) ≡ `true`,

αυτή απλουστεύεται σε: ***x*** ≥ *y*. Αυτή όμως συνάγεται από την προϋπόθεση *x* > *y*.

2: Για να ισχύει η ((*maxxy* ≥ *x*) **&&** (*maxxy* ≥ *y*)) **&&** ((*maxxy* **==** *x*) **||** (*maxxy* **==** *y*)) μετά τη "`maxxy = y`" θα πρέπει πριν από αυτήν να ισχύει η: ((***y*** ≥ *x*) **&&** (***y*** ≥ *y*)) **&&** ((***y*** **==** *x*) **||** (***y*** **==** *y*)). Παρομοίως και εδώ, αν παρουμε υπόψη μας ότι οι ***y*** ≥ *y* και ***y*** **==** *y* έχουν τιμή `true` για οποιαδήποτε τιμή της *y*, καθώς και τις ταυτολογίες που είδαμε παραπάνω, αυτή απλουστεύεται σε: ***y*** ≥ *x*. Αυτή όμως συνάγεται από την προϋπόθεση **!**(*x* > *y*) που σημαίνει: *x* ≤ *y*.

5.4.1 * Από το Τέλος προς την Αρχή

Στις αποδείξεις που κάναμε με την εντολή εκχώρησης, χρησιμοποιούσαμε το αξίωμα της εκχώρησης για βρούμε τη συνθήκη που θα πρέπει να ισχύει *πριν* από μια εντολή ώστε μετά από αυτή να έχουμε μια συγκεκριμένη απαίτηση. Εδώ θα δούμε πώς θα κάνουμε τα ίδια με τις εντολές **if**.

Το πρόβλημα που έχουμε να λύσουμε είναι το εξής: Ποια θα πρέπει να είναι η *P* ώστε να έχουμε:

```
   // P
      if (S)  Et else Ef
   // Q
```

Ας ονομάσουμε *Pt* και *Pf* τις συνθήκες για τις οποίες έχουμε:

```
// Pt                  // Pf
   Et        και          Ef
// Q                   // Q
```

Αν βρούμε τις *Pt* και *Pf* τότε η *P* είναι (δες και την Άσκ. 5-3):

$$(Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))$$

Παράδειγμα 1

Έστω ότι έχουμε να αποδείξουμε:

```
// (-1 ≤ x < 0) || (1 ≤ x < 6)
   if (x < 0)  y = x;
         else  y = x - 2;
// -1 ≤ y ≤ 4
```

Για να αποδείξουμε αυτό που ζητείται θα πρέπει:

1. να βρούμε, όπως είπαμε παραπάνω, την *P* ώστε:

```
// P
   if (x < 0)  y = x;
         else  y = x - 2;
// -1 ≤ y ≤ 4
```

2. να αποδείξουμε ότι η *P* συνάγεται από την προϋπόθεση (σ.κ. (*E*2), §0.4), δηλ.:

$((-1 \le x \le 0)$ `||` $(1 \le x \le 6)) \Rightarrow P$

Ξεκινούμε από το:

1. Οι αντιστοιχίες με το μοντέλο που δώσαμε πιο πάνω είναι:

- *Et* είναι η: `y = x;`
- *Ef* είναι η: `y = x - 2;`
- *Q* είναι η: $-1 \le y \le 4$

Θέλουμε λοιπόν να βρούμε την *Pt* και την *Pf* ώστε:

```
// Pt                        // Pf
   y = x;                       y = x - 2;
// -1 ≤ y ≤ 4                // -1 ≤ y ≤ 4
```

Για να έχουμε $-1 \le y \le 4$ μετά την `y = x` θα πρέπει να έχουμε πριν από αυτήν $-1 \le x \le 4$. Αυτή είναι η *Pt*.

Παρομοίως, για να έχουμε $-1 \le y \le 4$ μετά την `y = x - 2` θα πρέπει, πριν από αυτήν, να έχουμε $-1 \le x-2 \le 4$ ή αλλιώς $1 \le x \le 6$. Αυτή είναι η *Pf*.

Άρα, πριν από την **ifelse** θα πρέπει να ισχύει η: $(Pt$ `&&` $S)$ `||` $(Pf$ `&&` $(!S))$ ή αλλιώς, αν πάρουμε υπόψη μας ότι *S* είναι η $x < 0$ και `!`*S* η $!(x < 0)$ ή $(x \ge 0)$:

$$((-1 \le x \le 4) \;\&\&\; (x < 0)) \;||\; ((1 \le x \le 6) \;\&\&\; (x \ge 0))$$

ή αλλιώς: $(-1 \le x < 0)$ `||` $(1 \le x \le 6)$

Τώρα ερχόμαστε στο

2. Θα πρέπει να απόδείξουμε ότι η $(-1 \le x < 0)$ `||` $(1 \le x \le 6)$ συνάγεται από την προϋπόθεση $(-1 \le x < 0)$ `||` $(1 \le x < 6)$. Για να ισχύει η προϋπόθεση θα πρέπει να ισχύει είτε η $-1 \le x < 0$ είτε η $1 \le x < 6$ (είτε και οι δύο, αλλά στην περίπτωσή μας αυτό είναι αδύνατο). Αν ισχύει η πρώτη τότε η *P* ισχύει. Αν ισχύει η δεύτερη $(1 \le x < 6)$, τότε ισχύει και η $1 \le x \le 6$, άρα ισχύει και η *P*.

Επομένως το πρόγραμμά μας είναι σωστό.

Στην περίπτωση της **if** έχουμε το εξής πρόβλημα: Ποια θα πρέπει να είναι η *P* ώστε να έχουμε:

```
   // P
      if (S)  Et
   // Q
```

Ας ονομάσουμε, όπως παραπάνω, *Pt* τη συνθήκη για την οποία έχουμε:

```
// Pt
   Et
// Q
```

Αν βρούμε την *Pt*, η *P* είναι (δες και την Άσκ. 5-3):

$$(S \texttt{ \&\& } Pt) \texttt{ || } ((\texttt{! } S) \texttt{ \&\& } Q)$$

Παράδειγμα 2

Θα αποδείξουμε, από το τέλος προς την αρχή, την ορθότητα του τρίτου προγράμματος για τον υπολογισμό του μεγίστου, δηλαδή:

```
// true
   maxxy = y;
   if (x > maxxy)  maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Θα πρέπει πρώτα να βρούμε τη συνθήκη που θα πρέπει να ισχύει πριν από την **if**, δηλ. την $(S \texttt{ \&\& } Pt) \texttt{ || } ((\texttt{! } S) \texttt{ \&\& } Q)$. Στην περίπτωσή μας:

- S είναι η $x > maxxy$ και $\texttt{!}S$ είναι η $\texttt{!}(x > maxxy)$ ή $x \le maxxy$,
- Q είναι η $((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y))$

Το πρώτο βήμα είναι να υπολογίσουμε την Pt ώστε:

```
// Pt
   maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Όπως είδαμε και στο παράδ. της §5.5 η Pt είναι: $\boldsymbol{x} \ge y$.

$$((x > maxxy) \texttt{ \&\& } (x \ge y)) \texttt{ ||}$$
$$((x \le maxxy) \texttt{ \&\& } ((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y)))$$

ή, απλούστερα:

$$((x > maxxy) \texttt{ \&\& } (x \ge y)) \texttt{ ||}$$
$$(((maxxy \ge x) \texttt{ \&\& } (maxxy \ge y)) \texttt{ \&\& } ((maxxy \texttt{ == } x) \texttt{ || } (maxxy \texttt{ == } y)))$$

Έχουμε λοιπόν:

```
   maxxy = y;
// ((x>maxxy) && (x≥y)) ||
// ((maxxy≥x) && (maxxy≥y) && ((maxxy==x)||(maxxy==y)))
   if (x > maxxy)  maxxy = x;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

Αντικαθιστώντας στη συνθήκη που βρήκαμε την y στη θέση της $maxxy$ παίρνουμε τη συνθήκη που θα πρέπει να ισχύει πριν από τη `maxxy = y`:

$$((x > y) \texttt{ \&\& } (x \ge y)) \texttt{ || } (((y \ge x) \texttt{ \&\& } (y \ge y)) \texttt{ \&\& } ((y \texttt{ == } x) \texttt{ || } (y \texttt{ == } y)))$$

Αυτή όμως απλουστεύεται:

- η $(x > y) \texttt{ \&\& } (x \ge y)$ ισοδυναμεί με $(x > y)$,
- οι $y \ge y$ και $y \texttt{ == } y$ έχουν τιμή `true`

και η συνθήκη μας γίνεται:

$$(x > y) \texttt{ || } (y \ge x)$$

Αυτή όμως είναι η `true` που είναι και η προϋπόθεσή μας.

5.5 Εξασφάλιση Προϋποθέσεων

Τώρα έχουμε στα χέρια μας και άλλα εργαλεία –εκτός από την *assert*– για να ελέγχουμε τις προϋποθέσεις για την εκτέλεση των προγραμμάτων μας.

Παράδειγμα 1

Οι υπολογισμοί του προγράμματος για το πρόβλημα της ελεύθερης πτώσης απαιτούν να διασφαλίσθεί η συνθήκη $h \ge 0$. Αυτό μπορεί να γίνει πολύ εύκολα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
```

```
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   // Υπολόγισε τα tP, vP
      tP = sqrt( (2/g)*h );
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   // Τύπωσε τα tP, vP
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
}
```

Εκείνο το «**false**» στο σχόλιο, μετά το **else**, τι σημαίνει; Ότι δεν υπάρχουν τιμές των *tP* και *vP* που να πληρούν την απαίτηση όταν $h < 0$.

Σύγκρινέ αυτό το πρόγραμμα με αυτό της §4.4 και διάλεξε αυτό που προτιμάς: *assert* ή **if**. Θα επανέλθουμε...

Παράδειγμα 2

Να ξαναγυρίσουμε στο παράδειγμα του τριωνύμου, για το οποίο, την τελευταία φορά, ανακαλύψαμε ότι ξεχάσαμε τον έλεγχο της *a*.

Πριν κάνουμε οποιονδήποτε άλλον υπολογισμό θα πρέπει να ελέγξουμε αν: $a \neq 0$ και $\Delta \geq 0$. Αυτή είναι η προϋπόθεσή μας:

Προϋπόθεση: $(a \neq 0)$ **&&** $(b^2 - 4ac \geq 0)$

Απαίτηση: $(x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a})$ **&&** $(x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a})$

Το παρακάτω πρόγραμμα, που είναι μια τροποποιημένη μορφή του προγράμματος της §5.2, πριν από όλα ελέγχει αν ισχύει η προϋπόθεση.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double  a, b, c;    // οι συντελεστές της εξίσωσης
   double  x1, x2;     // οι ρίζες της εξίσωσης
   double  delta;      // η διακρίνουσα

   cout << "Δώσε τρεις πραγματικούς αριθμούς" << endl;
   cin >> a >> b >> c;
   cout << " a = " << a << "   b = " << b
        << "   c = " << c << endl;
   delta = b*b - 4*a*c;
   if ( a != 0  &&  delta >= 0 )
   {
      x1 = (-b + sqrt(delta))/(2*a);
      x2 = (-b - sqrt(delta))/(2*a);
      cout << " Λύση1 = " << x1 << "   Λύση2 = " << x2 << endl;
   }
   else // εκτός προδιαγραφών
   {
      if ( a == 0 )
```

```
         cout << " Η εξίσωση είναι 1ου βαθμού" << endl;
      else
         cout << " Δεν υπάρχουν Πραγματικές Λύσεις" << endl;
   }
   cout << " ΤΕΛΟΣ" << endl;
}
```

Τώρα, όπως βλέπεις, τα *x1* και *x2* θα υπολογιστούν μόνο στην περίπτωση που $a \neq 0$ **και** $delta \geq 0$.

Παρατηρήσεις ▶

1. Να υπενθυμίσουμε ότι αυτή η σύγκριση, "`a != 0`", δεν έχει και τόσο νόημα, αφού η *a* είναι τύπου `double`. Αν η τιμή της *a* δεν έρχεται από το πληκτρολόγιο, αλλά από ενδιάμεσους υπολογισμούς, μπορεί να έχει μια πολύ μικρή (απόλυτη) τιμή, που να μην είναι μηδέν, αλλά να δημιουργεί πρόβλημα.
2. Όπως ξέρεις, μπορούμε να συνεχίσουμε την αναζήτηση λύσης ακόμη και στην περίπτωση που *a* `==` 0. Δες την Άσκ. 5-7.
3. Το να γράφουμε τις εντολές εκχώρησης για τις *x1* και *x2* όπως ξέρουμε από τα μαθηματικά μπορεί να φαίνεται όμορφο αλλά δεν είναι και το καλύτερο για το πρόγραμμά μας. Όπως θα μάθουμε αργότερα, η αφαίρεση είναι μια «κακή» πράξη για τους τύπους κινητής υποδιαστολής διότι μπορεί να προκαλέσει *απώλεια σημαντικών ψηφίων*. Θα δούμε λοιπόν έναν άλλον τρόπο που αποφεύγει την αφαίρεση:

```
      if ( b >= 0 )
         x1 = (-b - sqrt(delta))/(2*a);
      else // b < 0
         x1 = (-b + sqrt(delta))/(2*a);
      x2 = c/(a*x1);
```

Αν, ας πούμε, τα *a*, *b*, *c* έχουν τιμές 1, -1, -30 αντιστοίχως, θα εκτελεσθεί η

```
         x1 = (-b + sqrt(delta))/(2*a);
```

όπου έχουμε τον υπολογισμό $-(-1)+\sqrt{(-1)^2 - 4\cdot1\cdot(-30)} = 1 + \sqrt{121} = 1 + 11 = 12$ (*x1* `==` 6). Η άλλη λύση υπολογίζεται (*x2* `==` -5) από τη γνωστή ιδιότητα: *x1*`*`*x2* `==` *c*`/`*a*.

Αν όμως τα *a*, *b*, *c* έχουν τιμές 1, 11, 30, θα εκτελεσθεί η

```
         x1 = (-b - sqrt(delta))/(2*a);
```

όπου έχουμε τον υπολογισμό $-11 - \sqrt{11^2 - 4\cdot1\cdot30} = -11 - \sqrt{1} = -11 - 1 = -(11+1) = -12$ (*x1* `==` -6). Η άλλη λύση υπολογίζεται ως: *x2* `==` (30`/`1)/(-6) `==` -5.

Και στις δύο περιπτώσεις, οι δύο όροι του αριθμητή έχουν το ίδιο πρόσημο και έτσι «αποφεύγουμε την αφαίρεση».◀

Παράδειγμα 3

Ας έρθουμε στο παράδ. 2 της §5.4 (υπολογιστής τσέπης). Όπως λέγαμε, έχουμε πρόβλημα όταν πάρουμε σύμβολο πράξης που δεν είναι '`+`', '`-`', '`*`', '`/`'. Ακόμη, έχουμε πρόβλημα όταν η πράξη είναι '`/`' και ο διαιρέτης είναι "`0`" (μηδέν). Να λοιπόν οι προδιαγραφές μας:

Προϋπόθεση:

(*oper* `==` '`+`') `||` (*oper* `==` '`-`') `||` (*oper* `==` '`*`') `||`
((*oper* `==` '`/`') `&&` (*x*2 `!=` 0))

Απαίτηση:

((*oper* `==` '`+`') `&&` (*res* `==` *x*1 + *x*2)) `||` ((*oper* `==` '`-`') `&&` (*res* `==` *x*1 - *x*2)) `||`
((*oper* `==` '`*`') `&&` (*res* `==` *x*1 * *x*2)) `||` ((*oper* `==` '`/`') `&&` (*x*2 `!=` 0) `&&` (*res* `==` *x*1 / *x*2))

και να πώς ελέγχουμε την προϋπόθεση στο πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   double x1, x2;           // Τα ορίσματα της πράξης
   unsigned char oper;      // Το σύμβολο της πράξης
```

```
   double res;              // Το αποτέλεσμα της πράξης

   cin >> oper >> x1 >> x2;
   if (oper == '+' || oper == '-' ||
       oper == '*' || (oper == '/' && x2 != 0))
   {
      if (oper == '+')            res = x1 + x2;
      else if (oper == '-')       res = x1 - x2;
      else if (oper == '*')       res = x1 * x2;
      else /* if (oper == '/') */ res = x1 / x2;
      cout << res << endl;
   }
   else
      cout << " ΛΑΘΟΣ" << endl;
}
```

Φυσικά και εδώ, όπως και στο προηγούμενο παράδειγμα, η σύγκριση `x2 != 0` δεν λέει και πολλά πράγματα.

5.6 Σειρά Εκτέλεσης των Πράξεων

Στο προηγούμενο κεφάλαιο, §4.1, όταν εξετάζαμε τη σειρά εκτέλεσης των πράξεων, αφήσαμε για αργότερα ένα «λεπτό» σημείο· ας το δούμε. Η C++ επιταχύνει τον υπολογισμό λογικών παραστάσεων ως εξής

- Στη λογική πράξη "`&&`" αν το πρώτο όρισμα υπολογιστεί "`false`" τότε το αποτέλεσμα της πράξης υπολογίζεται "`false`" χωρίς να υπολογιστεί το δεύτερο όρισμα.
- Στη λογική πράξη "`||`" αν το πρώτο όρισμα υπολογιστεί "`true`" τότε το αποτέλεσμα της πράξης υπολογίζεται "`true`" χωρίς να υπολογιστεί το δεύτερο όρισμα.

Τι επιπτώσεις μπορεί να έχουν τα παραπάνω στον προγραμματισμό; Ας δούμε ένα

Παράδειγμα

Ας πούμε ότι σε κάποιο πρόγραμμα θέλουμε να εκτελεσθούν οι εντολές E με την προϋπόθεση ότι $\sqrt{x} > y + 2$ και, φυσικά, ότι $x \geq 0$.

Με αυτά που ξέραμε μέχρι τώρα θα γράφαμε:

```
   if ( x >= 0 )
      if ( sqrt(x) > y + 2 )  E
```

Έτσι έχουμε σιγουριά ότι δεν θα γίνει απόπειρα υπολογισμού της τετραγωνικής ρίζας αν δεν έχουμε εξασφαλίσει ότι η x έχει τιμή μη αρνητική.

Αν πάρουμε υπόψη τον γρήγορο υπολογισμό της "`&&`" γράφουμε:

```
   if ( x >= 0 && sqrt(x) > y + 2 )  E
```

Αν η x έχει τιμή αρνητική τότε η "`x >= 0`" υπολογίζεται σε "`false`" και την ίδια τιμή παίρνει και ολόκληρη η συνθήκη *χωρίς να γίνει απόπειρα υπολογισμού της* `sqrt(x)`, οπότε οι E δεν εκτελούνται.

Αυτό το χαρακτηριστικό της C++ ονομάζεται **υπολογισμός παράστασης bool με βραχυκύκλωμα** (short-circuit boolean evaluation)[4] και το αναφέρουμε για να καταλαβαίνεις προγράμματα γραμμένα από άλλους και όχι για να τη χρησιμοποιείς κατ' ανάγκη. Ο πρώτος τρόπος, με τις δύο **if**, είναι πιο σίγουρος και σου δίνει πρόγραμμα πιο ευανάγνωστο.[5]

[4] Η «συγγενείς» προς τη C++ γλώσσες Java και η C# δίνουν διαφορετικούς τελεστές για τις βραχυκυκλωμένες πράξεις από αυτούς που δίνουν για τις συνήθεις.

[5] Φυσικά ο δεύτερος είναι πιο γρήγορος, αλλά άφησέ για αργότερα την ανησυχία για την ταχύτητα.

5.7 Τα ";" και Άλλα Λάθη

Καινούριες εντολές, πιο πολλές δυνατότητες για προγραμματισμό, αλλά και πιο πολλές πιθανότητες για λάθη.

Πρώτα απ' όλα πρόσεξε «το λάθος που θα κάνεις πάντα» (§4.1.1):

♦ ***Η σύγκριση για ισότητα γίνεται με το "==" και όχι με το "=".***

Από το μεταγλωττιστή περνάει εύκολα (με μια προειδοποίηση) και μετά δεν είναι εύκολο να το ανακαλύψεις, αν μάλιστα το πρόγραμμα είναι μεγάλο.

Ας έρθουμε τώρα στο ';'. Μερικοί έχουν τη συνήθεια να βάζουν αυτόν το χαρακτήρα κάθε φορά που τελειώνουν μια γραμμή. Γράφει λοιπόν κάποιος:

```
if ( x > y );
   maxxy = x;
else;
   maxxy = y;
```

Ο μεταγλωττιστής της Borland C++ θα δώσει μια προειδοποίηση και ένα λάθος. Συγκεκριμένα:

- Για την `if (x > y)` θα δώσει προειδοποίηση: «code has no effect» (κώδικας που δεν κάνει κάτι). Όπως είναι γραμμένη η εντολή, ο μεταγλωττιστής καταλαβαίνει ότι πρόκειται για μια **if** (όχι **ifelse**) που ζητάει: αν έχουμε `x > y` να εκτελεσθεί η κενή εντολή! Επομένως, η **if** δεν έχει οποιοδήποτε αποτέλεσμα.
- Για το `else` μας δίνει λάθος «misplaced else» (else σε λάθος θέση). Αφού κατάλαβε ότι έχουμε **if** και όχι **ifelse**, μας λέει ότι το **else** κακώς τοποθετήθηκε εκεί.

Μέχρι εδώ τα πράγματα δεν είναι και τόσο άσχημα: αφού μας βγάζει λάθος το βρίσκουμε, το διορθώνουμε και το ξαναδίνουμε:

```
if ( x > y )
   maxxy = x;
else;
   maxxy = y;
```

Ο μεταγλωττιστής σου το βρήκε εντάξει. Το δοκιμάζεις κιόλας: δίνεις στο *x* την τιμή 4, στο *y* την τιμή 5, σου βγάζει μεγαλύτερο το 5, όλα πάνε καλά. Κάνεις άλλη μια δοκιμή: 5 στο *x*, 4 στο *y*, σου βγάζει μεγαλύτερο το 4. Μα τι γίνεται; Τη μια δουλεύει, την άλλη δεν δουλεύει! Για να δούμε: Την πρώτη φορά η συνθήκη "`x > y`" ή "4 > 5" είναι `false`, άρα θα πρέπει να εκτελεσθεί η περιοχή του **else** που είναι... ποιά; Η *κενή εντολή!* Μόνο αυτή χωράει ανάμεσα στο **else** και το ';' που το ακολουθεί. Στη συνέχεια εκτελείται η εντολή που ακολουθεί την **ifelse**, η "`maxxy = y`" και παίρνεις σωστό αποτέλεσμα, αλλά τελείως συμπτωματικά. Τη δεύτερη φορά, η "`x > y`" παίρνει τιμή `true` και εκτελείται η περιοχή του **if**, "`maxxy = x`". Η *maxxy* παίρνει τη σωστή τιμή (5) και στη συνέχεια εκτελείται η εντολή που ακολουθεί την **ifelse**, "`maxxy = y`", που αλλάζει την τιμή της *maxxy* σε 4.

Πάντως, αν ψάξεις τα μηνύματα που σου στέλνει ο μεταγλωττιστής, θα βρεις μια προειδοποίηση για την εντολή: "`maxxy = x`" (περιοχή του **if**): «'maxxy' is assigned a value that is never used» (στη *maxxy* εκχωρείται μια τιμή που δεν χρησιμοποιείται ποτέ).

Παρόμοιο πρόβλημα θα έχεις και στην περίπτωση που θα βάλεις ';' μετά τη συνθήκη κάποιας **if**:

```
maxxy = y;
if (x > maxxy);
   maxxy = x;
```

Εδώ, περιοχή του **if** είναι η κενή εντολή και η "`maxxy = x`" είναι η επόμενη εντολή.

Συνοψίζοντας μπορούμε να πούμε:

♦ ***Δεν βάζουμε ποτέ*** ';'
  - ***μετά τη συνθήκη***
  - ***μετά το else.***

Τώρα που έμαθες τις εντολές **if** μπορείς να μάθεις κάτι που κάνουν οι πεπειραμένοι προγραμματιστές για να εντοπίσουν λάθη στα προγράμματά τους. Ορίζουν μια σταθερά:

```
const bool debug( true );
```

Στη συνέχεια, αντί για απλές "`cout << ...` ", που τυπώνουν τις τιμές των συνθηκών επαλήθευσης, όπως είδαμε στην §3.2, βάζουν εντολές:

```
if ( debug )  cout << ...
```

Όταν ανακαλύψουν τα λάθη τους και τα διορθώσουν, αλλάζουν την αρχική δήλωση σε:

```
const bool debug( false );
```

Έτσι, βλέπουν την εκτέλεση του προγράμματος χωρίς τα ενδιάμεσα αποτελέσματα. Μόλις παρουσιαστούν τα νέα λάθη, που η πείρα τους λέει ότι πρέπει να τα περιμένουν[6], ξαναλλάζουν την τιμή της *debug* σε `true`.[7]

Αυτά βέβαια στην περίπτωση που δεν διαθέτουν **Βοηθητικά Προγράμματα για Ανεύρεση Λαθών** (debugging[8] utilities, debuggers), που σου δίνουν τη δυνατότητα να παίρνεις ενδιάμεσα αποτελέσματα με πιο απλούς τρόπους.

5.8 Τι (Πρέπει να) Έμαθες

Τώρα πρέπει να ξέρεις να γράφεις προγράμματα που να έχουν περισσότερους από έναν κλάδους εκτέλεσης.

Θα πρέπει να μπορείς να χρησιμοποιείς συνθήκες για να επιλέγεις ποιες εντολές θα εκτελεσθούν και ποιες όχι, με βάση τις τιμές που έχουν οι μεταβλητές του προγράμματός σου κατά τη διάρκεια της εκτέλεσης. Θα πρέπει να μπορείς να «φωλιάζεις» εντολές **if** ή/και **ifelse** τη μια μέσα στην άλλη.

Θα πρέπει να ξέρεις να ελέγχεις αν τα δεδομένα που διαβάζεις συμμορφώνονται με τις προδιαγραφές.

Τέλος, αν μελετάς και τις παραγράφους με τα αστεράκια, θα πρέπει να μπορείς να αποδείξεις την ορθότητα προγραμμάτων με εντολές **if** ή/και **ifelse**. Θα ξέρεις επίσης και πώς διαπλέκονται οι συνθήκες επαλήθευσης με αυτές που χρησιμοποιούμε στις εντολές επιλογής.

Ασκήσεις

Α Ομάδα

5-1 Αν πριν από κάθε μια εντολή $a = 0$, $b = 1$, $x = 2$, $y = 3$, τι τιμές θα έχουν οι μεταβλητές μετά την εκτέλεση κάθε μιας από τις παρακάτω εντολές;

α) `if (x > y)  x = y; else y = x;`
β) `if (x == y)  a = b; else b = a;`
γ) `if (x > 0)  a = b;`
δ) `if (x = y)  y = x;`
ε) `if (a = sqrt(b))  y = x;`
στ) `if (x > x)  if (y < y)  a = b;`

---

[6] There is always one more bug! (νόμος του Murphy).

[7] Επειδή αυτοί οι έλεγχοι κάνουν το πρόγραμμα κάπως πιο αργό, συνήθως βάζουμε (παρόμοιες) οδηγίες προς τον μεταγλωττιστή όπως θα μάθουμε αργότερα.

[8] Θα δεις στα ελληνικά τον όρο «εκσφαλμάτωση».

ζ) `if (a + b)  y = x;`

5-2 Ο αλγόριθμος υπολογισμών προβλέπει, ότι η μεταβλητή *y* πρέπει να πάρει τις παρακάτω τιμές, ανάλογα με τις τιμές των μεταβλητών *a* και *b*:

$y = a + 100$, αν $a \geq 0$ και $b = 0$
$y = b + 5$, αν $a \geq 0$ και $b \neq 0$
$y = b - 5$, αν $a < 0$ και $b = 0$
$y = a - 100$, αν $a < 0$ και $b \neq 0$

Δώσε τις αντίστοιχες εντολές στην C++.

5-3 α) Είπαμε παραπάνω ότι στην *P* { **if** (*S*) *Et* **else** *Ef* } *Q* η *Et* εκτελείται με προϋπόθεση *P* **&&** *S* και μας δίνει *Q*, ενώ η *Ef* με προϋπόθεση *P* **&&** (**!***S*) και μας δίνει *Q*.

Στη συνέχεια είπαμε ότι, πηγαίνοντας από το τέλος προς την αρχή, αν *Pt* και *Pf* είναι τέτοιες ώστε: *Pt* { *Et* } *Q* και *Pf* { *Ef* } *Q* τότε πριν από την **ifelse** θα πρέπει να ισχύει η (*Pt* **&&** *S*) **||** (*Pf* **&&** (**!***S*)).

Για να είναι τα παραπάνω συμβιβαστά θα πρέπει:

$$(((Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))) \;\&\&\; S) \Rightarrow Pt$$

και

$$(((Pt \;\&\&\; S) \;||\; (Pf \;\&\&\; (!S))) \;\&\&\; (!S)) \Rightarrow Pf$$

Να τις αποδείξεις!

β) Είπαμε ακόμη ότι στην *P* { **if** (*S*) *Et* } *Q* η *Et* εκτελείται με προϋπόθεση *P* **&&** *S* και μας δίνει *Q*, ενώ (*P* **&&** (**!***S*)) ⇒ *Q*.

Στη συνέχεια είπαμε ότι, πηγαίνοντας από το τέλος προς την αρχή, αν *Pt* είναι τέτοια ώστε: *Pt* { *Et* } *Q* τότε πριν από την **if** θα πρέπει να ισχύει η (*Pt* **&&** *S*) **||** (*Q* **&&** (**!***S*)).

Για να είναι τα παραπάνω συμβιβαστά θα πρέπει:

$$(((Pt \;\&\&\; S) \;||\; (Q \;\&\&\; (!S))) \;\&\&\; S) \Rightarrow Pt$$

και

$$(((Pt \;\&\&\; S) \;||\; (Q \;\&\&\; (!S))) \;\&\&\; (!S)) \Rightarrow Q$$

Να τις αποδείξεις και αυτές!

Β Ομάδα

5-4 Ποια μαθηματική συνάρτηση υλοποιούν οι παρακάτω εντολές (όπου οι μεταβλητές *a*, *b*, *c* και *d* είναι τύπου `int`):

```
if ( a > b )  y = a; else y = b;
if ( c > y )  y = c;
if ( d > y )  y = d;
```

5-5 Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 30 € για κάθε παιδί που έχει, αν έχει μέχρι τρία (3) παιδιά. Αν έχει περισσότερα από 3 παιδιά η προσαύξηση είναι 50 € για κάθε παιδί. Γράψε πρόγραμμα που θα διαβάζει τον βασικό μισθό και τον αριθμό παιδιών ενός υπαλλήλου και θα υπολογίζει το οικογενειακό επίδομα και τον συνολικό μισθό.

5-6 (Σύνθεση των Ασκ. 2-14 και 2-15) Γράψε πρόγραμμα που θα διαβάζει τις τιμές των R_1 και R_2 (θετικούς αριθμούς) και θα υπολογίζει και θα τυπώνει τη συνολική αντίσταση *R*. Πριν πάρει τις τιμές των αντιστάσεων, το πρόγραμμα θα ζητάει να διαβάσει, από το πληκτρολόγιο, τον τρόπο σύνδεσης των αντιστάσεων. Οι δεκτές απαντήσεις θα είναι:

- `'P'` για παράλληλη σύνδεση,
- `'S'` για σύνδεση εν σειρά.

Να διατυπωθεί η προϋπόθεση και το πρόγραμμα να την ελέγχει.

5-7 Ξαναγράψε το πρόγραμμα για το τριώνυμο της §5.6, έτσι ώστε:

1. Στην περίπτωση που $a = 0$ να υπολογίζει την απλή λύση $x1 = -c/b$, αν $b \neq 0$.
2. Αν $a = b = 0$ να εξετάζει τις περιπτώσεις $c = 0$ και $c \neq 0$.

3. Στην περίπτωση που *delta* < 0 να υπολογίζει και να τυπώνει τα

$ReX1 = ReX2 = -b/(2a)$

$ImX1 = ImX2 = \sqrt{-delta}\ /\ (2a)$

Η εκτύπωση των μιγαδικών θα πρέπει να γίνεται στη μορφή:

"`(`", *πραγματικό μέρος*, "`,`", *φανταστικό μέρος*, "`)`"

αφού δοθεί μήνυμα ότι οι ρίζες είναι μιγαδικές.

Γ Ομάδα

5-8 Απόδειξε ότι (`int a, b, c, y`):

```
// true
   if ( a > b )  y = a; else y = b;
   if ( c > y )  y = c;
// (y == a || y == b || y == c) && (y ≥ a && y ≥ b && y ≥ c)
```

5-9 Θέλουμε να αντικαταστήσουμε, στην προϋπόθεση του προβλήματος της δευτεροβάθμιας εξίσωσης, τη σύγκριση $a \neq 0$ με κάτι πιο «ρεαλιστικό». Ας υποθέσουμε ότι δεν έχουμε υπερχείλιση κατά τον υπολογισμό των $-b + \sqrt{b^2 - 4ac}$ ή της $-b - \sqrt{b^2 - 4ac}$. Ποια συνθήκη θα πρέπει να πληρούται ώστε να μην έχουμε υπερχείλιση όταν υπολογίζονται οι ρίζες;

5-10 (Συμπλήρωση της Ασκ. 2-7). Γράψε πρόγραμμα που θα διαβάζει τις καρτεσιανές συντεταγμένες ενός σημείου στο επίπεδο και θα υπολογίζει και θα τυπώνει τις πολικές r και ϕ. Αλλά, προσοχή: η atan επιστρέφει τιμή στο $(-\pi/2, \pi/2)$ ενώ θα πρέπει να έχουμε: $-\pi < \phi \leq \pi$.

Αν το σημείο βρίσκεται στον άξονα y, δηλ.: $x = 0$, τότε η ϕ θα γίνεται:

- $\pi/2$ αν $y > 0$,
- $-\pi/2$ αν $y < 0$,

ενώ θα παραμένει αόριστη αν έχουμε και $y = 0$.

Σύγκρινε τη ϕ που παίρνεις με αυτό που σου δίνει η atan2(y, x).

5-11 Απόδειξε ότι:

```
// (ia ≤ na || (ia = na+1) && ib ≤ nb)
   if ( ib > nb || ia ≤ na )  ia = ia + 1;
                      else  ib = ib + 1;
// ia ≤ na + 1
```

5-12 Απόδειξε ότι:

```
// true
   if ( x > y )  maxxy = x;
   if ( x <= y )  maxxy = y;
// ((maxxy ≥ x) && (maxxy ≥ y)) && ((maxxy == x) || (maxxy == y))
```

5-13 Ένας άλλος τρόπος για να διατυπώσουμε την απαίτηση στο πρόγραμμα για το μέγιστο είναι ο εξής:

$$((x \geq y)\ \&\&\ (maxxy == x))\ ||\ ((y \geq x)\ \&\&\ (maxxy == y))$$

Απόδειξε ότι:

```
// true
   if (x > y)  maxxy = x;
         else  maxxy = y;
// ((x ≥ y) && (maxxy == x)) || ((y ≥ x) && (maxxy == y))
```

5-14 Ψάξε τα εγχειρίδια του ΗΥ σου να δεις ποιό είναι το σύνολο χαρακτήρων που χρησιμοποιεί.

Γράψε ένα πρόγραμμα που θα διαβάζει ένα χαρακτήρα από το πληκτρολόγιο και θα τυπώνει το μήνυμα:

- `"ψηφίο"` αν ο χαρακτήρας ήταν ψηφίο,
- `"γράμμα"` αν ο χαρακτήρας ήταν γράμμα,
- `"κάτι άλλο"` αν δεν είναι ούτε γράμμα ούτε ψηφίο.

Αν έχεις κεφαλαία και μικρά γράμματα θα πρέπει να διαχωρίζονται επίσης.

Δώσε δύο λύσεις:

- Χρησιμοποιώντας τις συναρτήσεις που είδαμε στο προηγούμενο κεφάλαιο.
- Χωρίς να χρησιμοποιήσεις τις συναρτήσεις. Στην περίπτωση αυτή, αν έχεις και Ελληνικά και είναι εύκολος ο διαχωρισμός, να διαχωρίζονται επίσης.

κεφάλαιο

6

Επαναλήψεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κάνεις το δεύτερο βήμα στη χρήση συνθηκών για τον έλεγχο εκτέλεσης ενός προγράμματος: Να προγραμματίζεις επαναλαμβανόμενη εκτέλεση (ομάδων) εντολών.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις προγράμματά που θα έχουν *επαναληπτικούς* υπολογισμούς. Με αυτά αρχίζεις να εκμεταλλεύεσαι τη μεγάλη δύναμη του υπολογιστή: Πολύ μεγάλη ταχύτητα στην εκτέλεση επαναληπτικών αλγορίθμων.

Έννοιες κλειδιά:

- *εντολές επανάληψης - βρόχοι*
- *εντολή* `while`
- *εντολή* `for`
- *αναλλοίωτη επανάληψης*
- *τιμή-φρουρός*
- *τερματισμός επανάληψης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Στον προγραμματισμό παρουσιάζεται συχνά η ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Φυσικά, αυτό που μας ενδιαφέρει είναι να γράψουμε τις εντολές μια φορά μόνον. Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει αυτήν τη δυνατότητα.

6.1 Επαναλήψεις

Ας ξεκινήσουμε με ένα παράδειγμα.

Έστω ότι θέλουμε να υπολογίσουμε και εκτυπώσουμε το άθροισμα και τη μέση αριθμητική τιμή *n* (π.χ.10) πραγματικών αριθμών –ας τους πούμε t_1, t_2,... t_n– που δίνονται στην είσοδο του υπολογιστή.

Με όσα ξέρουμε μέχρι τώρα, θα έπρεπε να δηλώσουμε *n* μεταβλητές *x*1, *x*2, ... *x*10 και να δώσουμε τις εντολές:

```
cin >> x1;  cin >> x2; ... cin >> x10;
sum = x1 + x2 + ... + x10;
```

Βέβαια, ένα τέτοιο πρόγραμμα θα ήταν τρομακτικό, αν μάλιστα το *n* δεν είναι 10, αλλά είναι 100 ή 1000 ή 10000.

Ας προσπαθήσουμε να το γράψουμε αλλιώς. Κοίταξε την παρακάτω εναλλακτική λύση:

```
sum = 0;                    // sum == 0
cin >> x;  sum = sum + x;   // (x == t1) && (sum == Σ(j:1..1)tj)
cin >> x;  sum = sum + x;   // (x == t2) && (sum == Σ(j:1..2)tj)
  . . .
cin >> x;  sum = sum + x;   // (x == tN) && (sum == Σ(j:1..n)tj)
```

Τι κάνει το (κάθε) *m*-οστό ζευγάρι εντολών

```
cin >> x;  sum = sum + x;
```

Διαβάζει από το πληκτρολόγιο τη *m*-οστή τιμή (t_m), την αποθηκεύει στη θέση της μνήμης *x* και την προσθέτει στη μεταβλητή *sum*. Όταν εκτελεσθεί το επόμενο ζευγάρι *η προηγούμενη τιμή που υπάρχει στη x θα σβηστεί*. Αυτό δεν μας δημιουργεί πρόβλημα, αφού από τις προδιαγραφές του προγράμματος φαίνεται ότι ο μόνος προορισμός κάθε τιμής που διαβάζεται είναι να προστεθεί στο άθροισμα.

Με τη *sum* τι γίνεται; Αρχικώς της δίνουμε την τιμή 0. Μετά το πρώτο ζευγάρι εντολών η τιμή της είναι η πρώτη τιμή που διαβάστηκε. Μετά το δεύτερο ζευγάρι η τιμή της *sum* είναι το άθροισμα των δύο πρώτων τιμών, μετά το *m*-οστό ζευγάρι η *sum* έχει ως τιμή το μερικό άθροισμα των *m* πρώτων τιμών. Μετά το *n*-οστό ζευγάρι η *sum* έχει ως τιμή το άθροισμα των *n* τιμών που πληκτρολογήθηκαν.

Όλα αυτά φαίνονται και στις συνθήκες που βάζουμε μετά το κάθε ζευγάρι εντολών. Με το `Σ(j: 1..m)tj` εννοούμε το: $\sum_{j=1}^{m} t_j$.

Ωραία λοιπόν! Φαίνεται ότι και αυτές οι εντολές λύνουν το πρόβλημα. Τι κερδίσαμε που τις γράψαμε; Η λύση του προβλήματός μας διατυπώθηκε ως επανάληψη μιας ακολουθίας εντολών: οι εντολές "`cin >> x;  sum = sum + x;`" εκτελούνται *n* φορές. Μπορούμε να πούμε στον υπολογιστή να τις εκτελέσει *n* φορές με αρκετά συνοπτικό τρόπο:

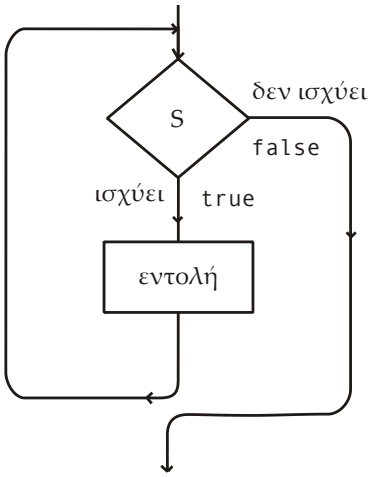
```
sum = 0;  m = 1;
όσο (m <= n) να εκτελείς την εντολή:
{
    cin >> x;  sum = sum + x;
 // (x == tm) && (sum == Σ(j:1..m)tj)
    m = m + 1;
}
```

Αυτό που εννοούμε εδώ είναι το εξής: όσο βρίσκει το $m \le n$ να εκτελεί τις τρεις εντολές που «πακετάραμε» σε μια σύνθετη εντολή. Η *m* είναι ένας **μετρητής** (counter) –συνήθως μεταβλητή τύπου (`unsigned`) `int`. Η τελευταία επαναλαμβανόμενη εντολή αυξάνει κάθε φορά την τιμή της *m* κατά 1· έτσι, αφού ξεκινάει από 1, όταν θα πάψει να ισχύει η συνθήκη "`m <= n`" η σύνθετη εντολή θα έχει εκτελεσθεί *n* φορές. Ακόμη, πρόσεξε ότι, αφού η *m* ξεκινάει από 1 και αυξάνεται κατά 1, είναι σίγουρο ότι κάποτε το *m* θα γίνει μεγαλύτερο από το *n*.

Το ίδιο πράγμα γράφεται στην C++:

```
sum = 0;  m = 1;
while ( m <= n )  {  cin >> x;  sum = sum + x;
                  // (x == tm) && (sum == Σ(j:1..m)tj)
                     m = m + 1;
                  }
```

Το **while** (που σημαίνει: όσο, για όσο, εφ' όσον) είναι λεξικό σύμβολο της C++. Η εντολή **while** είναι μια **επαναληπτική εντολή** (repetitive statement) και στο Πλ. 6.1 μπορείς να δεις την περιγραφή της. Η λειτουργία της εντολής **while** δίνεται επίσης από το λογικό διάγραμμα του Σχ. 6-1. Βλέποντάς το, μπορείς να κατάλάβεις γιατί οι επαναληπτικές εντολές λέγονται και **εντολές ανακύκλωσης** ή **βρόχοι** (loops).



Σχ. 6-1 Λογικό διάγραμμα της **while**.

Η συνθήκη $(x = t_m)$ **&&** $(sum = \sum_{j=1}^{m} t_j)$ δεν παύει να ισχύει, στο σημείο του προγράμματος που τη γράψαμε, όσο εκτελούνται ξανά και ξανά οι επαναλαμβανόμενες εντολές· λέμε λοιπόν ότι είναι **αναλλοίωτη** (invariant) της επανάληψης. Για αναλλοίωτες θα συζητήσουμε πιο εκτεταμένα παρακάτω.

Πρόσεξε το εξής: η **while**, όπως και η **if**, περιμένει μετά τη συνθήκη *μια* εντολή· αν θέλουμε να βάλουμε περισσότερες, όπως είδες ήδη στο παράδειγμά μας, τις «συσκευάζουμε» με ένα άγκιστρο (**{**) στην αρχή και ένα άγκιστρο (**}**) στο τέλος σε μια *σύνθετη εντολή*.

Άλλα παραδείγματα εντολών **while**:

```
while ( a > b )  a = a - b;
```

```
while ( n > 0 )
{
   a = a + n*n;
   n = n - 1;
}
```

Ας επιστρέψουμε τώρα στο πρόβλημά μας που διατυπώθηκε στο παράδειγμα. Το παρακάτω πρόγραμμα –Μέση Τιμή 1– είναι μια λύση με χρήση της εντολής **while**:

```
// πρόγραμμα: Μέση Τιμή 1
#include <iostream>
using namespace std;
int main()
{
   const int n( 10 );

   int m;       // Μετρητής των επαναλήψεων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  m = 1;
   while ( m <= n )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;          // x = tm
      sum = sum + x;           // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;           // Ετοιμαζόμαστε για τον επόμενο αριθμό
   } // while
   avrg = sum / n;
   cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
}
```

Πλαίσιο 6.1

Η Εντολή while

"`while`", "`(`", *συνθήκη*, "`)`", *εντολή*

Συστατικά:

- το `while` είναι λεξη-κλειδί,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή* είναι μια οποιαδήποτε εντολή της C++ και λέγεται *περιοχή της επανάληψης*.

Εκτέλεση της εντολής **while**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο `while`.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη.

Υπολογίζεται ξανά η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο **while**.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη κ.ο.κ.

Αν η τιμή της συνθήκης βρεθεί `false` τότε η εντολή που ακολουθεί τη συνθήκη δεν εκτελείται και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί τη **while**.

Αν η τιμή της συνθήκης είναι `false` την πρώτη φορά που θα ελεγχθεί, η εντολή που ακολουθεί τη συνθήκη δεν θα εκτελεστεί ποτέ!

Οι εντολές "`sum = 0; m = 1`", που δίνονται πριν από την εντολή **while**, εκτελούνται βέβαια μόνο μια φορά και αποτελούν το *μέρος προετοιμασίας* της επανάληψης, δηλ. το μέρος όπου καθορίζονται οι αρχικές τιμές μεταβλητών, που χρησιμοποιούνται μέσα στην περιοχή της **while**.

Στο πρόγραμμά μας το *n*, που καθορίζει το πλήθος των επαναλήψεων της **while**, είναι σταθερά (εδώ *n*=10), και επομένως η επαναληπτική διαδικασία δεν μπορεί να σταματήσει αν δεν διαβαστούν *n* αριθμοί. Μια τέτοια επιλογή όμως, που γίνεται *όταν γράφουμε το πρόγραμμα*, οδηγεί σε ένα πρόγραμμα εξαιρετικώς ανελαστικό. Πιο βολικές είναι οι λύσεις που μας επιτρέπουν να καθορίζουμε το πλήθος των αρχικών δεδομένων (στο παράδειγμά μας τον αριθμό *n*), *όταν αρχίζει η εκτέλεση του προγράμματος*. Έτσι δεν χρειάζεται αλλαγή (εδώ του ορισμού *n* = 100) και νέα μεταγλώττιση του προγράμματος κάθε φορά που αλλάζει το πλήθος των στοιχείων εισόδου. Σε ένα τέτοιο πρόγραμμα –ας το πούμε Μέση Τιμή 2– το *n* είναι *μεταβλητή* και η τιμή της πληκτρολογείται από το χρήστη πριν αρχίσει να πληκτρολογεί τους άλλους αριθμούς:

```
int n;       // Τώρα το πλήθος είναι μεταβλητή
   . . .
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
   . . .
```

Τώρα όμως προσοχή: Θα πρέπει να έχουμε ως προϋπόθεση *n* > 0 και να την ελέγχουμε μόλις διαβάσουμε την τιμή της *n*:

```
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
if ( n > 0 )
{
   sum = 0;  m = 1;
     . . .
}
```

Πλαίσιο 6.2α
Υπολογισμός Αθροίσματος

```
sum = 0;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   sum = sum + x;
      . . .
} // while
// sum == Σx
```

Πλαίσιο 6.2β
Υπολογισμός Γινομένου

```
prod = 1;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   prod = prod * x;
      . . .
} // while
// prod == Πx
```

```
else
   cout << " Το πλήθος πρέπει να είναι θετικό" << endl;
```

Αφήνουμε σε σένα, για άσκηση, να γράψεις ολόκληρο το Μέση Τιμή 2 (Άσκ. 6-1).

Όπως θα δεις και στη συνέχεια, έτσι υπολογίζουμε τα αθροίσματα ακόμη και αν ο έλεγχος της επανάληψης είναι διαφορετικός ή αν οι τιμές δημιουργούνται από το πρόγραμμα και δεν διαβάζονται από το πληκτρολόγιο. Σου το δίνουμε λοιπόν σαν συνταγή στο Πλ. 6.2α. Παρόμοιος είναι και ο υπολογισμός γινομένου (Πλ. 6.2β).

Αυτή η μορφή επανάληψης –που είδαμε στα παραπάνω παραδείγματα– είναι η πιο απλή και λέγεται **μετρούμενη** (counted) επανάληψη. Θα ασχοληθούμε με αυτήν ξανά στη συνέχεια.

6.1.1 Άγνωστο Πλήθος Στοιχείων – Τιμή-Φρουρός

Συχνά το πλήθος των στοιχείων που θέλουμε να δώσουμε στον υπολογιστή δεν είναι γνωστό. Σκέψου, για παράδειγμα, μερικά φύλλα με μετρήσεις από κάποιο πείραμα. Κάθε φορά που τα μετράς βρίσκεις και διαφορετικό πλήθος. Δεν είναι καλύτερο να τα μετρήσει το πρόγραμμα;

Έστω ότι θέλουμε να βρούμε και να τυπώσουμε το άθροισμα, τη Μέση Τιμή και το πλήθος πραγματικών αριθμών που θα πληκτρολογηθούν στην είσοδο του Υπολογιστή. *Ενώ το πλήθος τους δεν είναι γνωστό*, ξέρουμε ότι οι αριθμοί μας είναι *μη-μηδενικοί*.

Το πώς θα υπολογίσουμε το άθροισμα και τη μέση τιμή το ξέρουμε· δεν ξέρουμε όμως *πότε θα σταματήσει η επανάληψη*. Ή αλλιώς: *πώς θα πούμε στον Υπολογιστή ότι τελείωσαν τα στοιχεία εισόδου* και να μην περιμένει άλλα.

Είναι φανερό ότι την ώρα που εκτελείται το πρόγραμμα και δίνουμε τα στοιχεία εισόδου θα πρέπει να δώσουμε στον ΗΥ το σύνθημα «τέλος στοιχείων» με κάποιο τρόπο. Αλλά ποιά είναι η δυνατότητά μας να δώσουμε κάτι στον ΗΥ; Καθορίζεται με απόλυτη ακρίβεια από την εντολή: "`cin >> x`" που εκτελείται ξανά και ξανά. Μπορούμε λοιπόν να δώσουμε μια τιμή στη *x* που να είναι συνθηματική και να σημαίνει «τέλος». Ας δούμε τώρα, τι τιμές μπορεί να πάρει η *x*: τιμές τύπου **`double`**. Και τώρα γεννιούνται τα ερωτήματα: Πώς δεν θα γίνει μπέρδεμα; Γιατί ο ΗΥ δεν θα πάρει να επεξεργαστεί και τη συνθηματική τιμή όπως όλες τις άλλες; Και αν θέλουμε να δώσουμε αυτήν την τιμή για επεξεργασία;

Στο παράδειγμά μας, θα χρησιμοποιήσουμε την πληροφορία ότι *τα στοιχεία είναι μη μηδενικά*. Σχεδιάζουμε λοιπόν το πρόγραμμά μας με βάση την εξής σύμβαση: *Όταν τελειώσουν τα στοιχεία που θέλουμε να δώσουμε θα πληκτρολογήσουμε την τιμή "`0`"*. Πρόσεξε ότι:

- Το "`0`" είναι δεκτή απάντηση στην εντολή: `cin >> x`.
- Από τις προδιαγραφές του προβλήματος ξέρουμε ότι αποκλείεται να δώσουμε το 0 για επεξεργασία.

Το "0" λέγεται **τιμή-φρουρός** (sentinel value) και αυτή η τεχνική χρησιμοποιείται ευρύτατα από τους προγραμματιστές. Μπορούμε λοιπόν να πούμε ότι η **while** θα ξεκινάει ως εξής:

```
while ( x != sentinel )
```

Τώρα όμως κάναμε μια σοβαρή αλλαγή στη λογική του προγράμματος που είχαμε στα προηγούμενα παραδείγματα: Όταν έρχεται η στιγμή να εκτελεσθεί η **while** πρέπει να έχουμε την πρώτη τιμή της *x*. Η πρώτη ανάγνωση θα πρέπει να γίνεται πριν από την **while**:

```
cin >> x;
while ( x != sentinel )
```

Αλλαγών συνέχεια: Μέχρι τώρα η περιοχή της **while** είχε τη μορφή:

```
{
    cin >> x;
    Επεξεργάσου την τιμή που διάβασες στη x
}
```

Τι θα γίνει αν την αφήσουμε έτσι; Θα διαβάσουμε τη δεύτερη τιμή πριν επεξεργαστούμε την πρώτη. Θα πρέπει λοιπόν να αντιστρέψουμε τη σειρά των εντολών:

```
cin >> x;
while ( x != sentinel )
{
   Επεξεργάσου την τιμή που διάβασες στη x;
   cin >> x;      // Διάβασε την επόμενη τιμή
}
```

Και η τελευταία αλλαγή: Ο μετρητής μας, κάθε στιγμή, θα πρέπει να δείχνει πόσες τιμές διαβάστηκαν για επεξεργασία. Θα πρέπει να ξεκινάει από 0 και όχι από 1. Το παρακάτω πρόγραμμα, Μέση Τιμή 3, είναι μια λύση του προβλήματος, με αυτήν την τεχνική.

```
// πρόγραμμα: Μέση Τιμή 3
#include <iostream>
using namespace std;
int main()
{
   const double sentinel = 0;

   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  n = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;            // x == tn
      sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
}
```

Να παρατηρήσουμε ακόμη τα εξής:

1. Μπορεί να αναρωτιέσαι αν μπορούμε να γράψουμε το πρόγραμμα βάζοντας τις εντολές:

```
cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
```

Πλαίσιο 6.3

Γενικό Σχήμα Επανάληψης με Τιμή-Φρουρό

```
Ετοίμασε ή διάβασε την πρώτη τιμή v
while ( η v δεν είναι φρουρός )
{
   Επεξεργάσου τη v;
   Ετοίμασε ή διάβασε την επόμενη τιμή v
} // while
```

μόνο μια φορά. Ναι, μπορούμε! Δοκίμασε να το κάνεις.

2. Το πρόγραμμα μπορεί να γίνει πιο ευέλικτο αν ο *sentinel* δεν είναι σταθερά αλλά *μεταβλητή που η τιμή της θα δίνεται από το πληκτρολόγιο πριν από τα στοιχεία εισόδου*. Τροποποίησε το πρόγραμμα προς αυτήν την κατεύθυνση.

3. Μπορείς, αντί για μια τιμή-φρουρό, να έχεις μια ολόκληρη περιοχή τιμών. Π.χ. αν στο παράδειγμά μας ξέραμε ότι οι τιμές μας είναι *θετικές* θα μπορούσαμε να γράψουμε το πρόγραμμά μας ως εξής:

```
cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
while ( x > 0 )
{
   n = n + 1;            // x == tn
   sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
   cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
} //  while
```

4. Οι τιμές που ελέγχεις με φρουρό δεν είναι απαραίτητο να εισάγονται από το πληκτρολόγιο. Μπορεί να δημιουργούνται από το πρόγραμμα. Μπορεί, για παράδειγμα, να είναι όροι ακολουθίας που ξέρουμε το μαθηματικό της τύπο.

5. Δεν θα μπορούσαμε, αντί να ψάχνουμε για φρουρούς, να δηλώσουμε:

```
char resp;
```

να βάλουμε μια ερώτηση:

```
    cout << " Έχεις άλλες τιμές; (N/O): ");  cin >> resp;
```

και να ελέγχουμε την επανάληψη ως εξής:

```
while ( resp == 'N' )  { . . .
```

Και βέβαια θα μπορούσαμε[1]. (Πρόσεξε ότι στην περίπτωση αυτή δεν φτάνει ο έλεγχος για το `'N'`, αλλά πρέπει να ελέγχεις για ελληνικά, λατινικά, πεζά, κεφαλαία.)

Πάντως η τεχνική της τιμής-φρουρού (και η φιλοσοφία της) είναι πολύ χρήσιμη σε πολλές περιπτώσεις. Το γενικό σχήμα χρήσης το βλέπεις στο Πλ. 6.3.

6.1.2 Επιλεκτική Επεξεργασία

Το πρόβλημα που λύσαμε με τα τρία προγράμματα Μέση Τιμή μπορεί να το συναντήσεις αρκετά συχνά με δεδομένα διαφόρων ειδών, π.χ. μετρήσεις από κάποιο πείραμα, κάποια έρευνα αγοράς, κάποια σφυγμομέτρηση κλπ. Πολύ συχνά όμως, πέρα από τη γενική επεξεργασία που κάνουμε σε όλα τα στοιχεία, θέλουμε να κάνουμε ειδική επεξεργασία σε ένα υποσύνολο τιμών, που επιλέγονται με κάποια κριτήρια. Π.χ.

- Πόσες φορές μέτρησα στο πείραμά μου συχνότητα από 1000 *Hz* μέχρι 1500 *Hz*; Ποιά ήταν η μέση τιμή τους;
- Στη σφυγμομέτρησή μου, πόσοι από αυτούς που στις εκλογές ψήφισαν το κόμμα x, έχουν τελειώσει Πανεπιστήμιο;

[1] Μήπως στην περίπτωση αυτή κάθε τιμή ≠ `'N'` είναι φρουρός στην *resp*; Μήπως;...

- Στην έρευνα αγοράς που έκανα, πόσοι από αυτούς που έχουν πλυντήριο πιάτων κάνουν τα ψώνια τους σε μηνιαία βάση;

Στις περιπτώσεις αυτές το γενικό σχήμα επεξεργασίας είναι το εξής:

```
Κάνε τη γενική επεξεργασία στην τιμή x;
if (για τη x πληρούνται τα κριτήρια επιλογής)
   κάνε την ειδική επεξεργασία στην τιμή x
```

Ας κάνουμε κι εμείς κάτι παρόμοιο: θα τροποποιήσουμε το πρόγραμμα Μέση Τιμή 3 ώστε να μας δίνει επί πλέον:

- το πλήθος,
- το άθροισμα και
- τη μέση τιμή

όσων από τις τιμές που διαβάζει είναι θετικές και μέχρι (και) 10.

Η γενική επεξεργασία είναι αυτή που κάναμε και πιο πριν. Για την ειδική επεξεργασία χρειαζόμαστε: ένα μετρητή (*selN*) για το πλήθος των επιλεγόμενων τιμών, μια μεταβλητή (*selSum*) όπου θα «μαζεύουμε» το άθροισμα των επιλεγόμενων τιμών και μια μεταβλητή (*selAvrg*) για τη μέση τιμή τους. Η επιλογή και η ειδική επεξεργασία γίνεται με την εντολή:

```
      if ( 0 < x && x <= 10 )      // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;      // Αυτά γίνονται μόνο για
         selN = selN + 1;          // τους επιλεγόμενους αριθμούς
      } // if
```

Ολόκληρο το πρόγραμμα Μέση Τιμή 4:

```
// πρόγραμμα: Μέση Τιμή 4
#include <iostream>
using namespace std;
int main()
{
   const double sentinel( 0 );

   int n;          // Μετρητής όλων των στοιχείων
   int selN;       // Μετρητής επιλεγόμενων στοιχείων
   double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;     // Το άθροισμα όλων των στοιχείων
   double selSum;  // Άθροισμα επιλεγόμενων στοιχείων
   double avrg;    // Μέση Αριθμητική Τιμή όλων των στοιχείων
   double selAvrg; // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων

   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
   cout << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
   if ( selN > 0 )
   {
```

```
      selAvrg = selSum/selN;
      cout << " ΑΘΡΟΙΣΜΑ = " << selSum
           << "  <x> = " << selAvrg << endl;
   }
}
```

Εδώ βλέπεις μια **if** μέσα σε μια **while**.

6.2 * Αναλλοίωτες και Τερματισμός

Αντιγράφουμε από το παράδειγμα της §6.1:

```
   sum = 0;  m = 1;
   while ( m <= n )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;          // x = tm
      sum = sum + x;            // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;           // Ετοιμαζόμαστε για τον επόμενο αριθμό
   } // while
```

Είναι σωστό αυτό το πρόγραμμα; Δηλαδή: αν διαβαστούν n (> 0) τιμές t_1 ... t_n, θα ισχύει, μετά την εκτέλεση της **while**, η *sum* == $\sum_{j=1}^{n} t_j$;

Η τιμή του μετρητή m ξεκινάει από 1 και αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή επανάληψης. Άρα την πρώτη φορά που δεν θα ισχύει η `m <= n` –και θα σταματήσει η εκτέλεση της **while**– θα έχουμε m == n + 1. Αν αποδείξουμε ότι τότε θα ισχύει η *I*: *sum* == $\sum_{j=1}^{m-1} t_j$ θα έχουμε αποδείξει αυτό που θέλαμε.

Θα το αποδείξουμε με τη **μέθοδο της μαθηματικής επαγωγής**[2].

Βασικό βήμα: Αρχικώς, πριν αρχίσει η εκτέλεση της **while**, έχουμε, από τις εντολές προετοιμασίας: (*sum* == 0) && (m == 1). «Ταιριάζει» αυτή με την *sum* == $\sum_{j=1}^{m-1} t_j$; Στην *sum* έχουμε το άθροισμα των m – 1 πρώτων μελών του συνόλου { t_1 ... t_N } που έχουν ήδη πληκτρολογηθεί. Αφού αρχικώς δεν έχει πληκτρολογηθεί κάποια τιμή, η *sum* πρέπει να έχει τιμή 0. Αυτό είναι και το νόημα του $\sum_{j=1}^{1-1} t_j = \sum_{j=1}^{0} t_j$. Μπορούμε λοιπόν να πούμε ότι αρχικώς, πριν από τη **while**, ισχύει η *I*.

Επαγωγικό βήμα: Θα πρέπει να αποδείξουμε ότι αν η *I* ισχύει για κάποιο m θα ισχύει και για την επόμενη τιμή της m (δηλαδή: m+1). Στο πρόγραμμά μας όμως πηγαίνουμε στην επόμενη τιμή της m αφού διαβάσουμε, προσθέσουμε και μετρήσουμε την επόμενη τιμή. Θα πρέπει λοιπόν να αποδείξουμε ότι: *αν η I:* (*sum* == $\sum_{j=1}^{m-1} t_j$) *ισχύει όταν αρχίσουν να εκτελούνται οι επαναλαμβανόμενες εντολές θα ισχύει και μετά το τέλος της εκτέλεσής τους*, δηλαδή:

```
// sum == Σ(j:1..m-1)tj
   cin >> x;
   sum = sum + x;
   m = m + 1;
// sum == Σ(j:1..m-1)tj
```

[2] Δες για παράδειγμα Σ. Ανδρεαδάκη κ.ά. «ΑΛΓΕΒΡΑ Β′ ΛΥΚΕΙΟΥ», ΟΕΔΒ 1994.

Αυτό ξέρουμε να το αποδείξουμε: Για να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$ μετά τη `m = m + 1` θα πρέπει πριν από αυτήν να ισχύει η $sum == \sum_{j=1}^{(m+1)-1} t_j$ ή αλλιώς: $sum == \sum_{j=1}^{m} t_j$. Έχουμε δηλαδή:

```
   cin >> x;
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Για να ισχύει η $sum == \sum_{j=1}^{m} t_j$ μετά την `sum = sum + x` θα πρέπει πριν από αυτήν να ισχύει η: $sum + x == \sum_{j=1}^{m} t_j$:

```
   cin >> x;
// sum + x == Σ(j:1..m)tⱼ == Σ(j:1..m-1)tⱼ + tₘ }
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ)
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Αφού, όπως είπαμε, όταν η `cin >> x` εκτελείται για m-οστή φορά διαβάζεται η t_m, μπορούμε να πούμε ότι μετά την εκτέλεσή της θα έχουμε: $x == t_m$. Μαζί με το ότι $\sum_{j=1}^{m} t_j == \sum_{j=1}^{m-1} t_j + t_m$ καταλήγουμε στο ότι πριν από τη "`cin >> x`" θα πρέπει να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$.

Αυτή όμως ισχύει, αφού είναι η προϋπόθεσή μας.

Τι σημαίνουν τα παραπάνω με απλά λόγια;

- Η I ισχύει αρχικώς (βασικό βήμα), για $m == 1$.
- Αφού αποδείξαμε (επαγωγικό βήμα) ότι αν η I ισχύει αρχικώς[3] και εκτελεσθούν οι επαναλαμβανόμενες εντολές, που αυξάνουν την τιμή της m, η I ισχύει και πάλι, η I ισχύει και για $m == 2$.
- Αφού η I ισχύει για $m == 2$, λόγω του επαγωγικού βήματος, θα ισχύει και για $m == 3$

κ.ο.κ. Γενικώς, *η I ισχύει για οποιαδήποτε τιμή πάρει η m με τις επαναλαμβανόμενες εντολές*. Βλέπουμε λοιπόν ότι η εκτέλεση των επαναλαμβανόμενων εντολών αφήνει την I αναλλοίωτη· γι' αυτό και ονομάζεται **αναλλοίωτη της επανάληψης** (repetition invariant).

Παρατηρήσεις ▶

1. Για να αποδείξουμε την ορθότητα μιας **while** θα *πρέπει να έχουμε την αναλλοίωτη*. Πού θα τη βρούμε; Στα προγράμματα που γράφουμε εμείς δεν είναι δύσκολο να τη έχουμε: είναι, στην πραγματικότητα, η λογική περιγραφή της μεθόδου που χρησιμοποιούμε για να λύσουμε το πρόβλημά μας. Αν μας δώσουν μια **while** και μας ζητήσουν να βρούμε την αναλλοίωτη τα πράγματα είναι δύσκολα! Αν όμως υπάρχουν προδιαγραφές τα πράγματα είναι καλύτερα. Διάβασε όμως παρακάτω...
2. Όταν κάνουμε αποδείξεις από το τέλος προς την αρχή φτάνουμε σε κάτι σαν:

```
   while ( S )  E;
// Q
```

Εδώ τι κάνουμε; Αν μπορέσουμε να φέρουμε την Q στη μορφή (`!`S) `&&` Q_I, προσπαθούμε να αποδείξουμε ότι η Q_I είναι αναλλοίωτη της **while**. Φυσικά, η Q_I θα πρέπει να ισχύει και πριν

[3] Οι εντολές "`sum = 0; m = 1;`", που δίνονται πριν από την εντολή **while**, σκοπό έχουν να εξασφαλίσουν ότι ισχύει η I πριν από τη **while**.

από τη **while**. Αυτή η «συνταγή» μπορεί να μη φαίνεται και τόσο εύκολη για ορισμένες περιπτώσεις, όπως π.χ. η μετρούμενη επανάληψη· θα τα πούμε παρακάτω.
3. Πρόσεξε ακόμη το εξής: αν έχουμε **while** (*S*) *E*, με αναλλοίωτη *I*, οι *E* θα εκτελεσθούν μόνον αν ισχύει η *S*. Δηλαδή, αυτό που έχεις να αποδείξεις είναι:

```
// I && S
   E
// I  ◀
```

Τώρα, μπορούμε να διατυπώσουμε και συμβολικώς τον συμπερασματικό κανόνα της **while**:

$$\frac{I \,\&\&\, S\,\{E\}\,I}{I\,\{\textbf{while}(S)\;E\}\,(!S)\,\&\&\,I}$$

Και όταν κάνουμε αποδείξεις από το τέλος προς την αρχή, ποια συνθήκη θα πρέπει να ισχύει πριν από τη **while**; Ποια άλλη; Η *αναλλοίωτη*!

Όταν έχουμε επαναληπτικές εντολές, για να αποδείξουμε ότι το πρόγραμμά μας είναι ***ολικώς*** σωστό, *θα πρέπει να αποδείξουμε ότι η εκτέλεση όλων των επαναληπτικών εντολών θα τερματισθεί*. Πώς γίνεται αυτό; Ας έρθουμε στο παράδειγμά μας: Θεωρούμε την ακολουθία των διαδοχικών τιμών της διαφοράς $n - m$ που είναι ακέραιος αριθμός.

- Αφού η **while** εκτελείται μόνο όταν $m \le n$ έχουμε πάντοτε $n - m \ge 0$, δηλαδή έχουμε μια ακολουθία *φυσικών* αριθμών.
- Η ακολουθία αυτή είναι γνησίως φθίνουσα, αφού κάθε τιμή της *m* είναι κατά 1 μεγαλύτερη από την προηγούμενη.

Τα μαθηματικά μας λένε ότι *δεν είναι δυνατόν να έχουμε γνησίως φθίνουσα ακολουθία*[4] *φυσικών αριθμών με άπειρο πλήθος όρων*. Επειδή δεν είναι δυνατόν η ακολουθία που δημιουργεί η **while** να παραβιάσει αυτό το θεώρημα, η εκτέλεσή της θα σταματήσει κάποτε.

Αυτός είναι ο τρόπος που συνήθως χρησιμοποιούμε για να αποδείξουμε τον τερματισμό. Μερικές φορές δεν είναι και τόσο εύκολο να βρούμε τη γνησίως φθίνουσα ακολουθία φυσικών αριθμών.

6.2.1 * Παραδείγματα

Παράδειγμα 1 ✍

Ας υποθέσουμε ότι η C++ δεν μας δίνει την ακέραιη διαίρεση και την "**%**". Θέλουμε ένα πρόγραμμα που θα διαβάζει δύο ακέραιους *d1*, *d2* –ο πρώτος (διαιρετέος) μη αρνητικός, ο δεύτερος (διαιρέτης) θετικός– και θα υπολογίζει και θα τυπώνει το πηλίκο και το υπόλοιπο της ακέραιης διαίρεσης *d1*:*d2*.

Πρώτα οι προδιαγραφές: Για το υπόλοιπο *y*, σίγουρα θυμάσαι ότι, θα πρέπει να έχουμε $0 \le y < d2$. Για το πηλίκο *p* τι θα έχουμε; Τι άλλο από αυτό που σου μάθανε ως δοκιμή της διαίρεσης: $d1 == p \cdot d2 + y$:
Προϋπόθεση: $(d1 \ge 0)$ **&&** $(d2 > 0)$
Απαίτηση: $(0 \le y < d2)$ **&&** $(d1 == p \cdot d2 + y)$

Ο αλγόριθμος θα πρέπει να σου είναι γνωστός από το Δημοτικό Σχολείο (από τότε που σου λέγανε ότι η διαίρεση είναι πολλές αφαιρέσεις): όσο ο *d2* είναι μικρότερος από ή ίσος με *d1* (ο *d2* χωράει στον *d1*) αφαιρούμε από τον *d1* τον *d2* και μετρούμε την αφαίρεση. Τελικώς το πλήθος των αφαιρέσεων θα είναι το πηλίκο ενώ ότι έχει μείνει από τις διαδοχικές αφαιρέσεις είναι το υπόλοιπο. Δηλαδή:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
```

[4] Όπου λέμε «ακολουθία» εννοούμε ακολουθία με *πεπερασμένο* πλήθος όρων.

```
   while ( d2 <= y )
   {
      y = y - d2;
      p = p + 1;
   } // while
// (0 ≤ y < d2) && (d1 == p*d2 + y)
```

Ας αποδείξουμε τώρα, από το τέλος προς την αρχή, την ορθότητα του προγράμματός μας.

Όπως είπαμε, όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!`*S* `&&` *I*. Στην περίπτωσή μας

- `!`*S* είναι η `!(d2 <= y)`, ή αλλιώς: $d2 > y$.
- *I* θα πρέπει να είναι αναλλοίωτη της επανάληψης.

Πώς θα βρούμε την αναλλοίωτη; Παρατηρούμε το εξής: Η απαίτηση που έχουμε για μετά τη **while** είναι: $(0 \le y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$ ή αλλιώς: $(0 \le y)$ `&&` $(y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αν από αυτήν ξεχωρίσουμε την $y < d2$, που είναι η `!`*S*, το υπόλοιπο θα πρέπει να είναι η αναλλοίωτη. Θα πρέπει λοιπόν να αποδείξουμε ότι η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ είναι η αναλλοίωτη της **while**.

1. *Βασικό βήμα*: Η αναλλοίωτη θα πρέπει να ισχύει πριν από τη **while**. Θα πρέπει δηλαδή να έχουμε:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ μετά την `p = 0` θα πρέπει να έχουμε πριν από αυτήν $(0 \le y)$ `&&` $(d1 == 0{\cdot}d2 + y)$ ή αλλιώς:

$$(0 \le y) \text{ \&\& } (d1 == y)$$

```
   y = d1;
// (0 ≤ y) && (d1 == y) }
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == y)$ μετά την `y = d1` θα πρέπει να ισχύει πριν από αυτήν η: $(0 \le d1)$ `&&` $(d1 == d1)$ ή απλούστερα: $d1 \ge 0$ που συνάγεται από την προϋπόθεση.

2. *Επαγωγικό βήμα*: Θα αποδείξουμε ότι:

```
   // (d2 ≤ y) && (0 ≤ y) && (d1 == p*d2 + y)
      y = y - d2;
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να έχουμε μετά την `p = p + 1` την $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$:

```
      y = y - d2;
   // (0 ≤ y) && (d1 == (p+1)*d2 + y)
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$ μετά την `y = y - d2` θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y - d2)$ `&&` $(d1 == (p+1){\cdot}d2 + y - d2)$ ή αλλιώς: $(d2 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αυτή όμως συνάγεται από την $(d2 \le y)$ `&&` $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$.

Άρα το πρόγραμμά μας είναι *μερικώς σωστό*.

Να δούμε τώρα αν είναι και *ολικώς σωστό*, αν δηλαδή τερματίζεται η εκτέλεσή του. Ας θεωρήσουμε την ακολουθία των διαδοχικών τιμών της *y*. Αυτές είναι:

- ακέραιες,
- μη αρνητικές, αφού η `y = y - d2` εκτελείται μόνον αν $d2 \le y$ και
- κάθε μια είναι *μικρότερη από την προηγούμενή της* αφού έχουμε από την προϋπόθεσή μας ότι $d2 > 0$.

Έχουμε δηλαδή μια γνησίως φθίνουσα ακολουθία φυσικών αριθμών. Αυτή αποκλείεται να έχει άπειρο πλήθος όρων· άρα η εκτέλεση της **while** θα τερματισθεί κάποτε. Δηλαδή το πρόγραμμά μας *είναι ολικώς σωστό*. Καμάρωσέ το:

```
#include <iostream>
using namespace std;
int main()
{
   int d1, d2, p, y;

   cin >> d1 >> d2;
   if ( d1 >= 0 && d2 > 0 )
   { // (d1 ≥ 0) && (d2 > 0)
      y = d1;
      p = 0;
      while ( d2 <= y )  // I: (0 ≤ y) && (d1 == p*d2 + y)
      {
         y = y - d2;
         p = p + 1;
      } // while
   // (0 ≤ y < d2) && (d1 == p*d2 + y)
      cout << " Πηλίκο: " << p << "    Υπόλοιπο: " << y << endl;
      cout << " Η C++ δίνει: " << endl;
      cout << " Πηλίκο: " << (d1 / d2)
           << "    Υπόλοιπο: " << (d1 % d2) << endl;
   }
   else
   // false
      cout << " Λάθος! " << endl;
}
```

Παρατήρηση ▶

Τι θα πει διαίρεση δια 0 (μηδέν); Θα πει ότι η ακολουθία μας *δεν είναι γνησίως φθίνουσα* οπότε η απόδειξη του τερματισμού δεν γίνεται. Ούτε και τερματισμός της εκτέλεσης της **while** γίνεται! Να λοιπόν που μια από τις «κακοτοπιές» που προσέχουμε (διαίρεση δια 0) έχει σχέση με μια **while** που δεν τελειώνει ποτέ[5]! Έτσι, από περιέργεια, αξίζει τον κόπο να βγάλεις την **if**, που ελέγχει την προϋπόθεση, και να δώσεις $d2 = 0$. ◀

Παράδειγμα 2

Ας πούμε τώρα ότι η C++ *δεν* μας δίνει συνάρτηση για την τετραγωνική ρίζα θετικού αριθμού. Ας γράψουμε ένα πρόγραμμα που να την υπολογίζει. Πώς; Τα μαθηματικά[6] μας λένε ότι: αν $x_0 > 0$ και $x_0^2 > a$ τότε η ακολουθία:

$$\{ n: \mathbb{N}^+, x_n: \mathbb{R} \bullet x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}} \right) \}$$

συγκλίνει προς τη $\sqrt{a}$.[7]

Τι προδιαγραφές θα έχουμε; Ή αλλιώς: πόσο καλή προσέγγιση θα έχουμε; Ας πούμε ότι θέλουμε το σχετικό σφάλμα να είναι μικρότερο από κάποιο $\varepsilon > 0$:

$$\frac{\left| x - \sqrt{a} \right|}{\sqrt{a}} < \varepsilon \qquad (1)$$

---

[5] Φυσικά, ο υπολογιστής σου προσέχει ότι ο διαιρέτης είναι 0 πριν κάνει τη διαίρεση και διακόπτει την εκτέλεση του προγράμματος με το σχετικό μήνυμα λάθους.

[6] Η ακολουθία προκύπτει όταν προσπαθούμε να βρούμε μια προσέγγιση της λύσης της εξίσωσης $x^2 - A = 0$ με τη μέθοδο Newton - Raphson.

[7] Δες την απόδειξη στο `http://en.wikipedia.org/wiki/ Methods_of_computing_square_roots` ή στο (Κάππος 1962), ας πούμε.

Μπορούμε εύκολα να γράψουμε μια **while** που να υπολογίζει τους όρους αυτής της ακολουθίας:

```
x = αρχική τιμή;
while ( S )
   x = 0.5*(x + a/x);
```

Τι θα βάλουμε ως αρχική τιμή *x*; Κάποια τιμή που να μας δίνει: $(x > 0)$ **&&** $(x^2 > a)$. Να μια σκέψη:

```
if (a <= 1)  x = a + 1;  else  x = a;
```

αλλά θα πρέπει να αποδείξουμε ότι είναι σωστή:

```
// a > 0
   if (a <= 1)  x = a + 1;  else  x = a;
// (x > 0) && (x² > a) }
```

Για να ισχύει αυτό, αρκεί να ισχύουν οι:

```
// (a > 0) && (a <= 1)      // (a > 0) && (a > 1)
   x = a + 1;          και      x = a;
// (x > 0) && (x² > a)      // (x > 0) && (x² > a)
```

Η απόδειξη των παραπάνω είναι τετριμμένη.

Η *S* ποια θα είναι; Προφανώς η άρνηση της (1), δηλαδή η:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} \geq \varepsilon$$

ή:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} = \frac{\left|x-\sqrt{a}\right|\left|x+\sqrt{a}\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} = \frac{\left|x^2-a\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} \approx \frac{\left|x^2-a\right|}{2a} \geq \varepsilon$$

(στο τελευταίο βήμα χρησιμοποιήσαμε την προσέγγιση: $x \approx \sqrt{a}$).

Τι θα πάρουμε ως ε; Για να σκεφτούμε λίγο παραπάνω την τελευταία σχέση που γράφεται:

$$\left|\frac{x^2}{a}-1\right| \geq 2\varepsilon$$

Το ιδανικό θα ήταν να μηδενίσουμε το αριστερό μέρος. Αλλά ξέρουμε ότι το αριστερό μέρος θα είναι μηδέν αν η διαφορά του x^2/a από το 1 είναι < ε`double` (**DBL_EPSILON**). Επομένως, το καλύτερο που μπορούμε να πάρουμε είναι όταν ε = ½**DBL_EPSILON**. Μπορούμε λοιπόν να γράψουμε:

```
if ( a <= 1 )  x = a + 1;  else  x = a;
while ( fabs(x*x - a ) >= DBL_EPSILON*a)
   x = 0.5*(x + a/x);
```

Δεν θα ψάξουμε να βρούμε την αναλλοίωτη; *Δεν μας χρειάζεται!* Όλοι οι υπολογισμοί αποβλέπουν στο να ανατρέψουμε τη συνθήκη της **while**.

Είναι σίγουρος ο τερματισμός αυτής της **while**; Αφού τα μαθηματικά μας εγγυώνται ότι η ακολουθία των διαδοχικών τιμών της *x* συγκλίνει στην $\sqrt{a}$ έχουμε ότι για κάθε $\delta > 0$ υπάρχει *N* τέτοιο ώστε για κάθε $n > N$ να έχουμε $|x_n - \sqrt{a}| < \delta$. Αν λοιπόν πάρουμε $\delta = \varepsilon\sqrt{a}$, τοτε από κάποιο *n* και μετά θα έχουμε: $|x_n - \sqrt{a}| < \varepsilon\sqrt{a}$, που όπως είδαμε παραπάνω ισοδυναμεί με: $|x^2 - a| < 2\varepsilon a$. Αυτή όμως είναι η **!**$(|x^2 - a| \geq 2\varepsilon a)$, δηλαδή η άρνηση της συνθήκης της **while**.

Αλλά, προσοχή: Η Ανάλυση μας εγγυάται τη σύγκλιση με την προϋπόθεση ότι $a > 0$. Αν $a < 0$ –μπορείς να το δεις με μερικές δοκιμές– η **while** (και το πρόγραμμα) δεν τελειώνει ποτέ! Να λοιπόν άλλη μια περίπτωση (μετά τη διαίρεση δια μηδέν) όπου αυτό που λέγαμε «κακοτοπιά» έχει να κάνει με τον τερματισμό μιας **while**. Η C++ προσεγγίζει την τετραγωνική ρίζα (*sqrt*) με τον τρόπο που είδαμε παραπάνω, αλλά αν δώσεις αρνητικό όρισμα

στην *sqrt*, αντί να κολλήσει σε μια αέναη επανάληψη, προτιμάει να κόψει την εκτέλεση του προγράμματος.

Εδώ είδαμε παράδειγμα προγράμματος στο οποίο τα μαθηματικά μας εγγυώνται και τη μερική και την ολική ορθότητα. Και όλα αυτά που γράφουμε τι είναι; *Προσεκτική υλοποίηση αυτών που μας λένε τα μαθηματικά!*

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
#include <cmath>
using namespace std;
int main()
{
   double a, x;

   cin >> a;
   if ( a > 0 )
   {// a > 0
      if ( a <= 1 )  x = a + 1;  else  x = a;
      while ( fabs(x*x - a) >= DBL_EPSILON*a )
         x = 0.5*(x + a/x);
   // |x² - a| < εa
      cout.precision(16);
      cout << x << "  " << sqrt(a) << endl;
   }
   else
      cout << " μόνον θετικούς παρακαλώ" << endl;
}
```

Παράδειγμα 3

Πρόβλημα: Θέλουμε ένα πρόγραμμα που θα διαβάσει από το πληκτρολόγιο n (> 0) τιμές t_k, k = 1, 2,...n –ας πούμε τύπου `int`– και στο τέλος θα μας δώσει τη μεγαλύτερη από αυτές και τη σειρά (*kmax*) με την οποία πληκτρολογήθηκε. Πριν από τους αριθμούς θα διαβάζει την τιμή της n.

Σε μια θέση της μνήμης, *tmax*, κρατούμε τη μέγιστη τιμή από όσες έχουμε διαβάσει και σε μια άλλη, *kmax*, τη σειρά της. Να οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$

Απαίτηση: $1 \le kmax \le n$ && $tmax == t_{kmax}$ && $\forall j: 1..n \bullet t_j \le tmax$

Ας δούμε πώς μπορούμε να λύσουμε το πρόβλημά μας.

Διαβάζουμε την πρώτη τιμή, που προφανώς είναι και η μέγιστη μέχρι στιγμής:

```
   cin >> t;  // t == t1 }
   k = 1;  tmax = t;  kmax = k;
```

Τώρα διαβάζουμε τη δεύτερη τιμή. Αν είναι μεγαλύτερη από την πρώτη, που τη θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη δεύτερη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη την πρώτη:

```
   cin >> t;   // t == t2
   k = k + 1;  // == 2
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Στη συνέχεια διαβάζουμε την τρίτη τιμή. Αν είναι μεγαλύτερη από αυτήν που θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη τρίτη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη αυτήν που είχαμε μέχρι τώρα:

```
   cin >> t;   // t == t3
   k = k + 1;  // == 3
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Όπως βλέπεις, διαχειριζόμαστε τη 2η και την 3η τιμή με τον ίδιο ακριβώς τρόπο και με τον ίδιο τρόπο θα διαχειριστούμε και τις υπόλοιπες. Στην 1η έχουμε διαφορά. Μπορούμε

λοιπόν να ξεκινήσουμε, όπως είδαμε, με την πρώτη τιμή και να βάλουμε τα υπόλοιπα σε μια **while**:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if (t > tmax)  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

Όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!(k <= n)`, δηλαδή η $k > n$. Αφού όμως η k αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή της επανάληψης και αφού $n > 0$ θα έχουμε ακριβέστερα: $k == n + 1$. Από αυτό και από την απαίτηση προσπαθούμε να μαντέψουμε την αναλλοίωτη της επανάληψης: βάζουμε στην απαίτηση όπου n το k–1 και παίρνουμε:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j\text{: } 1..\ k\text{–}1 \bullet t_j \le tmax$$

Αν αποδείξουμε ότι αυτή είναι η αναλλοίωτη *I*, τότε σίγουρα μετά τη **while** θα έχουμε την απαίτηση. Η απόδειξη δεν είναι δύσκολη αλλά είναι αρκετά μακροσκελής και την αφήνουμε για άσκηση (6-16).

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  t, tmax;
   int  n, k, kmax;

   cin >> n;
   if ( n > 0 )
   {// n > 0
      cin >> t;     // t == t1
      k = 1;
      tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
   // 1 ≤ kmax ≤ n && tmax == tkmax && ∀j:1..n • tj ≤ tmax
      cout << " Μέγιστη τιμή: " << tmax
           << "  (" << kmax << "η) " << endl;
   }
   else
      cout << " το πλήθος πρέπει να είναι θετικό" << endl;
}
```

Στο Πλ. 6.4 βλέπεις το γενικό σχήμα προγράμματος για τον υπολογισμό μεγίστου. Για να υπολογίσεις το ελάχιστο άλλαξε τη συνθήκη στην **if**:

```
if ( t < tmin ) ...
```

Μερικές φορές, όπως θα δεις στη συνέχεια, δεν είναι εύκολο να επεξεργαστείς χωριστά την πρώτη τιμή. Γράφουμε λοιπόν τη **while** έτσι ώστε να επεξεργάζεται όλες τις τιμές (k από 1 μέχρι n). Τώρα όμως υπάρχει το εξής πρόβλημα: την πρώτη φορά που θα εκτελεσθεί η **if** ($t > tmax$) τι τιμή θα έχει η *tmax*;

Η *tmax* θα πρέπει να έχει τέτοια τιμή ώστε:

- να αλλάξει οπωσδήποτε –όποια και αν είναι η πρώτη τιμή της t– και έτσι να πάρει μια ρεαλιστική τιμή,
- να μην υπάρχει πιθανότητα να θεωρηθεί ως μια από τις τιμές που επεξεργαζόμαστε.

Δηλαδή, πρέπει να βάλουμε ως αρχική τιμή της *tmax* μια *απίθανα μικρή τιμή*, π.χ. το $-\infty$!

«Πλην άπειρο και πράσινα άλογα!» θα σκεφτείς, «Γίνονται τέτοια πράγματα;» Δεν γίνονται βέβαια, αλλά για κάθε πρόβλημα που έχεις να λύσεις, είναι πολύ πιθανό να μπορείς να βρεις μια απίθανα μικρή τιμή. Π.χ.

- αν οι τιμές που μελετάς είναι θετικές, μια απίθανα μικρή τιμή είναι το 0 ή το -1 (ή οποιοσδήποτε αρνητικός),
- αν ξέρεις ότι οι τιμές που επεξεργάζεσαι είναι μεταξύ 150 και 250, μια απίθανα μικρή τιμή μπορεί να είναι το 0 ή το 10 κλπ.

Στο παράδειγμα που δώσαμε παραπάνω, αν ξέραμε επιπλέον ότι *όλες οι τιμές που διαβάζουμε είναι θετικές*, θα μπορούσαμε να γράψουμε:

```
tmax = -1;
k = 1;
while ( k <= n )
{
   cin >> t;  // t = tk
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Φυσικά, αν ψάχνεις για ελάχιστη τιμή θα ξεκινήσεις από μια *απίθανα μεγάλη τιμή* ($+\infty$).

Πλαίσιο 6.4

Εύρεση Μέγιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmax = t;  kmax = k;
k = k + 1;  // == 2
while ( συνθήκη )
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Εύρεση Ελάχιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmin = t;  kmin = k;
k = k + 1;  // == 2
while (συνθήκη)
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t < tmin )  {  tmin = t;  kmin = k;  }
   k = k + 1;
} // while
```

6.3 Η Μετρούμενη Επανάληψη

Η **μετρούμενη επανάληψη** είναι πολύ συνηθισμένη και απλή. Στην §6.1 είδαμε την απλούστερη μορφή της:

```
m = 1;
while ( m <= n )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m + 1;  }
```

Από όσα μάθαμε μέχρι τώρα, αν οι «άλλες επαναλαμβανόμενες εντολές» δεν αλλάζουν την τιμή της *m*, θα εκτελεσθούν:

- 0 φορές αν $n < 1$. Η τιμή της *m* δεν θα αλλάξει.
- *n* φορές αν $n \geq 1$. Στο τέλος η τιμή της *m* θα είναι $n + 1$.

Όταν οι «άλλες επαναλαμβανόμενες εντολές» εκτελούνται για *k*-οστή φορά έχουμε $m == k$.

Η *m* είναι η **μεταβλητή ελέγχου** (control variable) της επανάληψης.

Πολλοί προτιμούν μια άλλη μορφή για την ίδια δουλειά:

```
m = n;
while ( m >= 1 )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m - 1;    }
```

ή

```
m = n;
while (m > 0)  {  άλλες επαναλαμβανόμενες εντολές
                  m = m - 1;    }
```

διότι η ακολουθία τιμών της *m* είναι ακριβώς αυτή που χρησιμοποιήσαμε για να αποδείξουμε τον τερματισμό.

Ας δούμε όμως την εξής γενίκευση: αν η(οι) εντολή (-ές) *E* δεν μεταβάλλουν τις τιμές των *m*, *Ma*, *Mt*, *b* (οποιουδήποτε αριθμητικού τύπου) και αν $b > 0$, τότε κατά την εκτέλεση των

```
m = Ma;
while ( m <= Mt )
{
   E;
   m = m + b;
}
```

η(οι) εντολή(-ές) *E* θα εκτελεσθεί(-ούν) *n* φορές όπου:

- $n = 0$ αν $Ma > Mt$,
- $n = Trunc[(Mt - Ma + b)/b]$ φορές αν $Ma \leq Mt$.

Κατά την *k*-οστή φορά που θα εκτελεσθεί(-ούν) η(οι) *E*, η *m* θα έχει τιμή $Ma + (k-1)\cdot b$. Το *b* λέγεται **βήμα** (step).

Ας δούμε ένα

Παράδειγμα

Το παρακάτω πρόγραμμα μας δίνει, για τα πρώτα 15 sec, την απόσταση που διανύει, ανά 0.5 *sec*, ένα κινητό που ξεκινάει από την ηρεμία και κινείται με σταθερή επιτάχυνση 10 m/sec^2.

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf(ios::fixed,ios::floatfield);
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
```

```
   step = 0.5; // sec
   t = 0.0;
   while ( t <= 15.0 )
   {
      x = 0.5*a*t*t;
      cout.width(6);  cout.precision(1);  cout << t << "     ";
      cout.width(7);  cout.precision(2);  cout << x << endl;
      t = t + step;
   } // while
}
```

Το πρόγραμμα αυτό θα μας δώσει τα αποτελέσματα σε μορφή πίνακα:

```
t          x
   sec        m
   0.0        0.00
   0.5        1.25
   1.0        5.00
  . . .      . . .
  14.5     1051.25
  15.0     1125.00
```

6.4 Η Εντολή for

Η C++ έχει μια εντολή, τη **for**, που παραδοσιακά χρησιμοποιείται κυρίως για να γράφουμε μετρούμενες επαναλήψεις. Π.χ. η:

```
   m = Ma;
   while ( m <= Mt )
   {
      E;
      m = m + b;      // b > 0
   }
```

που είδαμε στην προηγούμενη παράγραφο μπορεί να γραφεί:

```
   for ( m = Ma;  m <= Mt;  m = m + b )
   {
      E;
   }
```

ενώ η

```
   m = Ma;
   while ( m >= Mt )
   {
      E;
      m = m - b;      // b > 0
   }
```

μπορεί να γραφεί:

```
   for ( m = Ma;  m >= Mt;  m = m - b )
   {
      E;
   }
```

Η μεταβλητή ελέγχου *m* πρέπει να είναι αριθμητικού ή απαριθμητού τύπου. Συνηθέστερα χρησιμοποιούμε ακέραιο ή απαριθμητό τύπο.

Τα *Ma, Mt, b* μπορεί γενικώς να είναι παραστάσεις. Παρ' όλο που δεν απαγορεύεται, απόφευγε όταν χρησιμοποιείς τη **for** να γράφεις επαναλαμβανόμενες εντολές (*E*) που να τροποποιούν οποιαδήποτε από τις *m, Ma, Mt, b*.

- ***Είναι καλό οι τιμές των Ma, Mt, b να καθορίζονται όταν αρχίζει η εκτέλεση της for και να μη μεταβάλλονται οι τιμές τους στη διάρκεια της εκτέλεσης της for. Η τιμή της μεταβλητής ελέγχου θα πρέπει να μεταβάλλεται μόνο από την*** `m = m ± b`.

Και γιατί όλα αυτά; Με τους παραπάνω περιορισμούς, τα δύο σχήματα επανάληψης με **for** που γράψαμε πιο πάνω δεν χρειάζονται απόδειξη για τον τερματισμό. Χωρίς αυτούς τους περιορισμούς η απόδειξη ορθότητας μπορεί να γίνει αρκετά πολύπλοκη. Γράφουμε λοιπόν μια **while** (και όχι **for**) για να ξέρουμε τι μας περιμένει.

Όπως η **while** έτσι και η **for**, περιμένει *μια επαναλαμβανόμενη εντολή*· αν θέλουμε να βάλουμε περισσότερες, όπως κάναμε στην **if** και στη **while**, τις «συσκευάζουμε» με ένα άγκιστρο (`{`) στην αρχή και ένα άγκιστρο (`}`) στο τέλος σε μια *σύνθετη εντολή*.

Δες πώς θα μπορούσε να γραφεί η επανάληψη του Μέση Τιμή 1 (ή 2) με χρήση της **for**:

```
   sum = 0;
   for ( m = 1;  m <= n;  m = m+1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;       // x = tN
      sum = sum + x;     // (x == tm) && (sum = Σ(j:1..m)tj)
   } // for
   avrg = sum / n;
```

Ας δούμε μερικά παραδείγματα ακόμη:

Παράδειγμα 1

Η μεταβλητή ελέγχου της **for** μπορεί να είναι και τύπου `bool`. Ξαναγράφουμε το πρόγραμμα της §4.3 που δημιουργεί τους αληθοπίνακες:

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;
   int   iP, iQ;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q " << endl;
   for ( iP = 0;  iP <= 1;  iP = iP+1 )
   {
      P = static_cast<bool>(iP);
      for (iQ = 0;  iQ <= 1;  iQ = iQ+1)
      {
         Q = static_cast<bool>(iQ);
         cout << P << "   " << Q << "   " << !P << "   "
              << (P && Q) << "    " << (P || Q) << "    "
              << (P <= Q) << "    " << (P == Q) << "    "
              << (P != Q) << endl;
      } // for (iQ...
   } // for (iP
}
```

Παράδειγμα 2

Ο πιο συνηθισμένος τύπος για τη μεταβλητή ελέγχου είναι ο `int`. Ας δούμε το εξής πρόβλημα: Να γραφεί πρόγραμμα που θα διαβάζει την τιμή του n και θα υπολογίζει και θα τυπώνει το άθροισμα:

$$1^1 + 2^2 + \ldots + n^n$$

Εδώ βλέπουμε ότι έχουμε (α) μετρούμενη επανάληψη (1.. n) και (β) υπολογισμό αθροίσματος. Σύμφωνα με όσα μάθαμε (Πλ. 6.2α) γράφουμε:

```
   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      υπολόγισε τον m-οστό όρο mm;
      sum = sum + mm;
   }
```

Αυτό το πρόγραμμα θα μοιάζει με αυτό της Μέσης Τιμής, με τη διαφορά ότι ο όρος που θα προσθέτουμε στο *sum* δεν προέρχεται από ανάγνωση από το πληκτρολόγιο αλλά από υπολογισμό που γίνεται μέσα στο πρόγραμμα.

Για τον υπολογισμό του m^m θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση **pow**. Αλλά εδώ θα κάνουμε τον υπολογισμό με πολλούς πολλαπλασιασμούς:

$$m^m = \underbrace{m \cdot \ldots \cdot m}_{m\varphi o\rho\varepsilon\varsigma}$$

Το πώς υπολογίζουμε ένα γινόμενο το είδαμε στο Πλ. 6.2β. Υπολογίζουμε λοιπόν το m^m ως εξής:

```
product = 1;  k = 1;          product = 1;
while ( k <= m )          ή  for ( k=1; k <= m; k=k+1 )
{
   product = product*m;          product = product*m;
   k = k + 1;
} // while (k ...
```

Να λοιπόν το πρόγραμμά μας:

```
#include <iostream>
using namespace std;
int main()
{
   int  n, sum;
   int  m, k, product;

   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      product = 1;
      for ( k = 1;  k <= m;  k = k + 1 )
      {
         product = product*m;
      } // for (k...
      sum = sum + product;
   } // for (m...
   cout << " n = " << n << "  Άθροισμα Σειράς = "
        << sum << endl;
} // main
```

Παράδειγμα 3

Ας έρθουμε τώρα στο πρόγραμμα της §6.4: Εδώ η μεταβλητή ελέγχου είναι τύπου **double**. Μπορούμε να γράψουμε την επανάληψη με μια **for** ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 8 );
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
   step = 0.5; // sec
   for ( t = 0.0;  t <= 15.0;  t += step )
   {
```

```
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
   } // for
}
```

Πάντως, όπως θα μάθεις αργότερα, είναι καλύτερο να σκεφτείς ότι οι επαναλαμβανόμενες εντολές εκτελούνται 31 φορές και να γράψεις:

```
   int    k;
     . . .
   t = 0.0;
   for ( k = 1;  t <= 31;  k++ )
   {
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
      t = t + step;
   } // for
```

Η **for** της C++ έχει πολύ περισσότερες δυνατότητες. Προς το παρόν είδαμε –και θα χρησιμοποιούμε– μόνον αυτές που μας χρειάζονται.

6.5 Λαθάκια και Σοβαρά Λάθη

Τα λάθη στις επαναληπτικές εντολές μπορεί να είναι πολύ πιο «δραματικά» και αυτό έχει να κάνει με τον τερματισμό. Το δίδαγμα από εδώ είναι:

- ***Σε κάθε εντολή while πρέπει να φροντίζεις ώστε κάποτε, κατά τη διάρκεια της εκτέλεσης, η συνθήκη της εντολής να γίνεται*** `false`**.**

Τα ίδια ισχύουν και για τη **for**, αλλά οι περιορισμοί που έχουμε βάλει στη χρήση της είναι πολύ ισχυρότεροι.

Ένας πολύ απλός τρόπος να παραβείς το παραπάνω δίδαγμα είναι να βάλεις ένα `";"` μετά την παρένθεση της συνθήκης της **while**:

```
while ( S );
```

Στην περίπτωση αυτή ζητάς την εκτέλεση της *κενής εντολής* όσο ισχύει η συνθήκη της **while**. Αν λοιπόν η συνθήκη ισχύει αρχικώς και αρχίσει η εκτέλεση της **while**, η κενή εντολή δεν μπορεί να την ανατρέψει. Αν δεν το πιστεύεις δοκίμασέ το. Αλλά να θυμάσαι:

- ***Δεν βάζουμε ποτέ "*** `;` ***" μετά την παρένθεση της συνθήκης της while.***

Όπως βλέπεις, ένα «τόσο δα λαθάκι» μπορεί να έχει σοβαρά επακόλουθα.

6.6 Τι (Πρέπει να) Έμαθες

Θα πρέπει να μπορείς να γράψεις προγράμματα στα οποία υπάρχει ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Θα πρέπει να μπορείς να γράψεις τέτοια προγράμματα είτε ξέρεις το πλήθος των επαναλήψεων πριν αρχίσει η εκτέλεσή τους (μετρούμενη επανάληψη) είτε θα τις συνεχίσεις μέχρι να ισχύσει κάποια συνθήκη (π.χ. να διαβαστεί η τιμή-φρουρός).

Θα πρέπει να έμαθες ακόμη να κάνεις μερικές πολύ συνηθισμένες επαναληπτικές δουλειές:

- υπολογισμό αθροίσματος ή γινομένου,

- υπολογισμό μεγίστου ή ελαχίστου

για τιμές που δημιουργεί το πρόγραμμα ή για τιμές που διαβάζει,

Τέλος, θα πρέπει να μπορείς να αποδείξεις την ορθότητα (απλών τουλάχιστον) προγραμμάτων με επαναλήψεις.

Ασκήσεις

Α Ομάδα

6-1 Γράψε το πρόγραμμα Μέση Τιμή 2 που περιγράψαμε στην §6.1. Θα διαφέρει από το Μέση Τιμή 1 στο ότι το *n* είναι μεταβλητή που η τιμή της διαβάζεται πριν από τις τιμές που θα αθροίσουμε.

6-2 Με βάση το πρόγραμμα για το άθροισμα, γράψε πρόγραμμα που θα υπολογίζει και θα εκτυπώνει το γινόμενο *n* πραγματικών αριθμών που δίνονται από το πληκτρολόγιο. Πριν από τις τιμές θα δίνεται η τιμή του *n*. Απόδειξε ότι το πρόγραμμά σου είναι ολικώς σωστό.

6-3 Ξαναδιάβασε το πρόγραμμα για το λογαριασμό της ΔΕΗ στην §6.4. Τροποποίησέ το έτσι ώστε να βγάζει πολλούς λογαριασμούς, διαβάζοντας από το πληκτρολόγιο τις καταναλώσεις. Το πρόγραμμα θα σταματάει αν του δώσουμε *αρνητική* κατανάλωση (κάθε αρνητική τιμή είναι τιμή - φρουρός).

Β Ομάδα

6-4 Τροποποίησε το πρόγραμμα Μέση Τιμή 3 της ώστε να έχει μια μόνον φορά την `cin >> x`. Σου αρέσει όπως έγινε;

6-5 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει την τιμή του *n*-οστού όρου της ακολουθίας, που καθορίζεται από τον τύπο:

α) $a_0 = 0,\ a_{k+1} = a_k k + k,$ για $k \geq 1$

β) $b_0 = b_1 = 0,\ b_k = b_{k-1} + b_{k-2} + k,$ για $k \geq 2$

6-6 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο πραγματικούς αριθμούς – θετικούς ή αρνητικούς– και θα σταματάει μόλις διαβάσει 10 αρνητικούς. Στο τέλος θα μας λέει:

- πόσους αριθμούς διάβασε συνολικώς και
- το άθροισμα των θετικών αριθμών που διάβασε.

6-7 Μετά την ανακοίνωση των αποτελεσμάτων στις εξετάσεις δύο μαθημάτων, έγινε φανερό ότι στο δεύτερο από αυτά έγινε «σφαγή». Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 40 δυάδες πραγματικών αριθμών που αντιπροσωπεύουν τους βαθμούς 40 σπουδαστών στα δύο μαθήματα. Στο τέλος θα πρέπει να μας λέει:

- πόσοι σπουδαστές είχαν στο δεύτερο μάθημα μεγαλύτερο βαθμό από ότι στο πρώτο,
- τους μέσους όρους των βαθμών στα δύο μαθήματα.

6-8 Απόδειξε ότι:

```
// n > 0
   k = 1;  sum = 0;
   while ( k <= n )    // I:  (sum = (k - 1)*k/2)
   {  sum = sum + k;  k = k + 1;  }
// sum = (n + 1)*n/2
```

Γ Ομάδα

6-9 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 150 πραγματικούς αριθμούς. Το πρόγραμμα θα πρέπει να υπολογίσει και στο τέλος να γράψει:

- το άθροισμα των θετικών,
- πόσοι είχαν απόλυτη τιμή μεγαλύτερη από 10,
- ποιος ήταν ο μέγιστος *αρνητικός* από τους αριθμούς που δόθηκαν (δηλαδή, αρνητικός με τη ελάχιστη απόλυτη τιμή).

6-10 Ξαναδιάβασε το πρόγραμμα της ελεύθερης πτώσης, της §3.4 και κάνε τις εξής παραλλαγές:

1. Το πρόγραμμα να τυπώνει ανά 1 sec, από τότε που αφέθηκε και μέχρι να κτυπήσει στο έδαφος
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - το ύψος που βρίσκεται το σώμα,
   - την ταχύτητα που έχει το σώμα.
2. Το πρόγραμμα να τυπώνει ανά 10 *m* διανυθέντος διαστήματος, από τότε που αφέθηκε (διάστημα 0) και μέχρι να κτυπήσει στο έδαφος (διάστημα *h*)
   - το διάστημα που διανύθηκε από την αρχή της πτώσης,
   - το ύψος που βρίσκεται το σώμα,
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - την ταχύτητα που έχει το σώμα.
3. Το πρόγραμμα δεν θα διαβάζει το αρχικό ύψος αλλά θα υπολογίζει και θα τυπώνει τα *tP*, *vP* για *h* = 100 *m*, 200 *m*, ..., 1000 *m*.

6-11 *Μέθοδος των ελαχίστων τετραγώνων.* Αν μας δοθούν *n* σημεία του επιπέδου *xy*, (x_k, y_k), *k* = 1..*n*, η «καλύτερη ευθεία», $y = \alpha x + \beta$, που περνάει από αυτά, είναι αυτή που έχει:

$$\alpha = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - \left(\sum x\right)^2} \quad \text{και} \quad \beta = \frac{\sum y \sum x^2 - \sum x \sum xy}{n\sum x^2 - \left(\sum x\right)^2}$$

όπου:

$$\sum x = \sum_{k=1}^{n} x_k \,,\ \sum y = \sum_{k=1}^{n} y_k \,,\ \sum x = \sum_{k=1}^{n} n_k^2 \,,\ \sum xy = \sum_{k=1}^{n} x_k y_k$$

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του *n* και στη συνέχεια τα (x_k, y_k), *k* = 1..*n* και θα υπολογίζει τα α και β.

6-12 Απόδειξε ότι αν:

```
   int  x, z, u;
```

τότε:

```
// x ≥ 0
   z = 0;  u = 1;
   while ( u <= x )     // I: (z² ≤ x) && (u = (z+1)²)
   {
      z = z + 1;
      u = u + z + z + 1;
   }
// z² ≤ x < (z+1)²
```

Απόδειξε πρώτα ότι η $(z^2 \le x)$ **&&** $(u = (z+1)^2)$ είναι αναλλοίωτη της **while**.

Όπως καταλαβαίνεις, το *z* είναι ακέραια προσέγγιση στην $\sqrt{x}$. Συμπλήρωσε το παραπάνω πρόγραμμα έτσι ώστε να διαβάζει από το πληκτρολόγιο την τιμή της *x* και εφόσον είναι σύμφωνη με την προϋπόθεση να υπολογίζει και να τυπώνει την «ακέραια τετραγωνική ρίζα» της *x* με τον παραπάνω τρόπο και με την `static_cast <int>(sqrt(x))`.

6-13 Απόδειξε ότι αν:

```
   int  x, c;
```

τότε:

```
// x ≥ 0
   c = x;
   while ( c*c*c > x )   // a: x < (c + 1)³
      c = c - 1;
// c³ ≤ x < (c + 1)³
```

6-14 Έστω ότι έχουμε την ακολουθία: $a_k = k!/p^k \mid k = 1,2,...$ όπου p φυσικός > 1. Όποια και να είναι η τιμή της p, η ακολουθία είναι τελικώς αύξουσα, αλλά στη αρχή –στους πρώτους όρους– βλέπεις οτι οι τιμές είναι φθίνουσες. Γράψε πρόγραμμα που

- θα διαβάζει την τιμή του p από το πληκτρολόγιο,
- θα υπολογίζει και θα τυπώνει τους $2p$ πρώτους όρους της ακολουθίας,
- θα μας δίνει την τιμή και την τάξη του ελάχιστου από αυτούς.

6-15 Θέλουμε πρόγραμμα που θα παρακολουθεί έναν παίκτη του μπάσκετ κατά τη διάρκεια ενός παιγνιδιού. Το πρόγραμμα θα παίρνει τα εξής στοιχεία:
`'1'` κάθε φορά που ο παίκτης επιτυγχάνει καλάθι με ελεύθερη βολή,
`'2'` κάθε φορά που ο παίκτης επιτυγχάνει δίποντο,
`'3'` κάθε φορά που ο παίκτης επιτυγχάνει τρίποντο και
`'4'` κάθε φορά που ο παίκτης κάνει φάουλ.
Πληκτρολογώντας `'0'` δείχνουμε στο πρόγραμμα ότι τελείωσε το παιχνίδι. Όταν ο παίκτης συμπληρώσει πέντε φάουλ, θα πρέπει να βγαίνει το μήνυμα: `"ΒΓΑΙΝΕΙ ΕΞΩ ΜΕ 5 ΦΑΟΥΛ"`.

Όταν τελειώσει η συμμετοχή του παίκτη –είτε λόγω αποβολής είτε λόγω τέλους του παιχνιδιού– θα γράφεται στην οθόνη η στατιστική του.

6-16 Απόδειξε ότι η:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j: 1..\, k\text{–}1 \bullet t_j \le tmax$$

είναι αναλλοίωτη της **while** στο:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

6-17 α) Γενίκευσε το πρόγραμμα ανάγνωσης ακεραίου της §4.5 ώστε να διαβάζει οποιαδήποτε τιμή τύπου `unsigned long`.

β) Βελτίωσε το πρόγραμμά σου ώστε να διαβάζει και προσημασμένες τιμές.

6-18 Βελτίωσε το πρόγραμμα που έγραψες στην άσκ. 6.17 ώστε να διαβάζει πραγματικούς της μορφής πρόσημο, ακέραιο μέρος, κλασματικό μέρος.

6-19 *Ο Χρόνος στη C++.* Η C++ μας δίνει τη συνάρτηση *time* που, με την κλήση `time(0)`, μας επιστρέφει τα δευτερόλεπτα που πέρασαν από τη στιγμή 00:00:00 *GMT, της* 1ης Ιανουαρίου 1970. Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <ctime>`".

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο ακέραιους αριθμούς και θα σταματάει όταν διαβάσει την τιμή 0. Αρχίζοντας από τον δεύτερο αριθμό, μετά την ανάγνωση θα μας δίνει τον χρόνο που πέρασε από την προηγούμενη πληκτρολόγηση. Στο τέλος θα μας δίνει:

- το πλήθος των πληκτρολογήσεων,

- το μέγιστο χρονικό διάστημα μεταξύ δύο πληκτρολήσεων,
- τον μέσο χρόνο μεταξύ δύο πληκτρολογήσεων.

κεφάλαιο

7

Συναρτήσεις Ι

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μπορείς να γράφεις δικές σου συναρτήσεις και να τις χρησιμοποιείς στα προγράμματά σου.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς στα προγράμματά σου συναρτήσεις που δεν υπάρχουν στις βιβλιοθήκες της γλώσσας και θα τις γράφεις εσύ. Πέρα από αυτό θα μπορείς να γράφεις συναρτήσεις για να επιμερίσεις την πολυπλοκότητα του προγράμματός σου και να μπορείς να χειριστείς πιο εύκολα προβλήματα ελέγχου και επαλήθευσης.

Έννοιες κλειδιά:

- *(ολική) συνάρτηση*
- *μερική συνάρτηση*
- *σύνολο αφετηρίας, σύνολο τιμών*
- *πεδίο ορισμού*
- *τυπική παράμετρος*
- *πραγματική παράμετρος (όρισμα)*
- *εντολή* `return`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Όταν έχουμε να γράψουμε ένα πρόγραμμα για να λύσουμε κάποιο σύνθετο πρόβλημα, χωρίζουμε συνήθως το όλο πρόβλημα σε επιμέρους προβλήματα και μετά γράφουμε ανεξάρτητα τμήματα προγραμμάτων για το κάθε ένα από αυτά. Τα ανεξάρτητα αυτά τμήματα προγραμμάτων λέγονται **υποπρογράμματα** (subprograms) ή, όπως τις λέει η C++, **συναρτήσεις** (functions) και μπορούμε να τα θεωρήσουμε σαν εντολές ή συναρτήσεις μιας γλώσσας προγραμματισμού που δημιουργούμε για να λύσουμε το πρόβλημά μας.

Στη C++ υπάρχουν δύο διαφορετικές κατηγορίες συναρτήσεων: *με τύπο* –που θα δούμε τώρα– και *χωρίς τύπο* (**void**), που θα δούμε αργότερα.

7.1. Συναρτήσεις με Τύπο – Εισαγωγή

Στα προηγούμενα κεφάλαια γνωρίσαμε μερικές *συναρτήσεις*, από τις βιβλιοθήκες της C++, όπως π.χ. οι συναρτήσεις *sqrt()*, *exp()*, *pow()*, *log()*, *cos()*, *sin()* κλπ., που χρησιμοποιούνται συχνά στα προγράμματά μας. Συχνά όμως, χρειαζόμαστε και άλλες συναρτήσεις, που φυσικά δεν μπορούσαν να προβλέψουν αυτοί που σχεδίασαν τη C++. Γι' αυτό η γλώσσα μας δίνει τη δυνατότητα υλοποίησης οποιασδήποτε (κατ' αρχήν) συνάρτησης μέσα στο πρόγραμμα και φυσικά τη δυνατότητα χρήσης αυτής της συνάρτησης.

Ας ξεκινήσουμε με τα μαθηματικά: αν έχουμε δύο σύνολα A και B, μια **μερική συνάρτηση** (partial function) f από το A στο B είναι ένα σύνολο ζευγών (x, y), όπου $x \in A$ και $y \in B$, που δεν περιέχει ζεύγη που να έχουν το ίδιο x και διαφορετικά y. Το A λέγεται **σύνολο αφετηρίας** (domain) και το B **σύνολο** (ή **πεδίο**) **τιμών** (range) της f. Γράφουμε:

$f: A \nrightarrow B$

Το υποσύνολο του A για κάθε μέλος x του οποίου υπάρχει ζεύγος (x, y) στην f, λέγεται **πεδίο ορισμού** (domain of definition) της f. Αν το πεδίο ορισμού είναι ίσο με ολόκληρο το σύνολο αφετηρίας η f λέγεται **ολική συνάρτηση** (total function) ή απλώς **συνάρτηση** (function). Γράφουμε:

$f: A \to B$

Σε κάθε ζεύγος (x, y), το x λέγεται **πρότυπο** και το y **εικόνα** (image). Γράφουμε $y = f(x)$ ή $x \mapsto y$.

Παραδείγματα

Αν συμβολίσουμε με το $\sqrt{}$ το σύνολο των ζευγών $(x, \sqrt{x})$ τότε έχουμε:

$\sqrt{} : \mathbb{R} \nrightarrow \mathbb{R}$

δηλαδή η τετραγωνική ρίζα είναι μερική συνάρτηση από το $\mathbb{R}$ στο $\mathbb{R}$, διότι στο σύνολο $\sqrt{}$ δεν έχουμε ζεύγη για $x < 0$. Αλλά:

$\sqrt{} : \mathbb{R}_{0+} \to \mathbb{R}$

Παρομοίως, αν πάρουμε το σύνολο f των ζευγών $(x, 1/x)$, έχουμε:

$f: \mathbb{R} \nrightarrow \mathbb{R}$

διότι: ενώ $0 \in \mathbb{R}$, δεν υπάρχει στο f ζεύγος με πρώτο μέλος "`0`". Και στην περίπτωση αυτή όμως μπορούμε να έχουμε μια ολική συνάρτηση:

$f: \mathbb{R}^{*} \to \mathbb{R}$

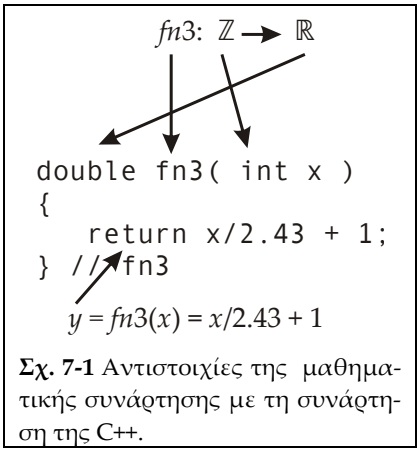
Όπως είδες στα παραδείγματα, στα μαθηματικά μπορούμε να περιορίσουμε το σύνολο αφετηρίας ώστε να γίνει ίσο με το πεδίο ορισμού. Στον προγραμματισμό αυτό δεν είναι πάντοτε εφικτό.

Συνήθως υπάρχει κάποιος *μηχανισμός*, δηλαδή τρόπος υπολογισμού, που μας επιτρέπει από τα πρότυπα x να βρίσκουμε τις εικόνες y ώστε το $(x, y) \in f$. Π.χ.

$$y = f(x) = 2x^2 + \frac{5}{x-1} \quad \text{ή}$$

$$y = f(x) = \begin{cases} -5, & -5.5 \le x \le -4.5 \\ 0, & -4.5 < x < 4.5 \\ 5, & 4.5 \le x \le 5.5 \end{cases}$$

Μια συνάρτηση με τύπο της C++ είναι ένα κομμάτι προγράμματος που υλοποιεί έναν τέτοιο μηχανισμό. Ας πούμε ότι έχουμε τη συνάρτηση *fn3*: $\mathbb{Z} \to \mathbb{R}$ με μηχανισμό υπολογισμού: $y = fn3(x) = \frac{x}{2.43} + 1$. Στο Σχ. 7-1 βλέπεις τι θα γράψουμε στη C++ και τις αντιστοιχίες μεταξύ αυτών που είπαμε πιο πάνω και του ορισμού της συνάρτησης στη C++.



Σχ. 7-1 Αντιστοιχίες της μαθηματικής συνάρτησης με τη συνάρτηση της C++.

Όπως βλέπεις, ο ορισμός μιας συνάρτησης ξεκινάει με μια **επικεφαλίδα**, που αποτελείται:

- από το όνομα τύπου που αντιστοιχεί στο πεδίο τιμών της συνάρτησης –ℝ που μεταφράζεται στον `double`–
- το όνομα της συνάρτησης –στην περίπτωσή μας `fn3`– και
- τον τύπο που αντιστοιχεί στο πεδίο ορισμού –ℤ που μεταφράζεται στον τύπο `int`.

Το *x* είναι μια **παράμετρος** της συνάρτησης· οι παράμετροι αν υπάρχουν γράφονται μέσα σε παρενθέσεις, μετά το όνομα. Στο **σώμα** (body) του υποπρογράμματος γράφεται κομμάτι προγράμματος που υλοποιεί το μηχανισμό υπολογισμού της εικόνας από το πρότυπο.

Μέσα στο σώμα της συνάρτησης θα πρέπει να υπάρχει οπωσδήποτε μια εντολή

`return` *Π*`;`

όπου *Π* μια παράσταση. Η τιμή της *Π*, αφού μετατραπεί στον τύπο *T* της συνάρτησης (`static_cast<`*T*`>`), είναι η τιμή που *επιστρέφει* η συνάρτηση.

Πού γράφουμε τη συνάρτησή μας; *Ο ορισμός μιας συνάρτησης πρέπει να βρίσκεται, κατ' αρχήν, πριν από τη χρήση (κλήση) της*. Το τι σημαίνει αυτό θα γίνει φανερό στη συνέχεια.

Πώς χρησιμοποιούμε μια δική μας συνάρτηση; Όπως ακριβώς χρησιμοποιούμε και τις προδηλωμένες συναρτήσεις της C++, μέσα σε παραστάσεις. Π.χ. μπορούμε να γράψουμε:

```
synist = f1*fn3(h/3) + f2 - f3*cos(phi/3+pi/2);
```

Συνοψίζοντας μπορούμε να πούμε ότι: **συνάρτηση** (function) *με τύπο* στη C++ *είναι ανεξάρτητο υποπρόγραμμα, που υπολογίζει και μεταβιβάζει ένα μοναδικό αποτέλεσμα*. Το αποτέλεσμα αυτό έχει τον δικό του τύπο, ο οποίος μπορεί να διαφέρει από τον τύπο των τυπικών παραμέτρων της συνάρτησης. *Το μοναδικό αποτέλεσμα της συνάρτησης διαβιβάζεται μέσω του ονόματός* της.

7.2. Η Εντολή "`return`"

Είπαμε παραπάνω ότι: με τη "`return` *Π*" καθορίζουμε ότι η συνάρτησή μας θα επιστρέψει την τιμή της Π· ακριβέστερα: *επιστρέφει την τιμή της Π αφού τη μετατρέψει στον τύπο της συνάρτησης*. Αλλά η **return** κάνει και κάτι άλλο: *τελειώνει την εκτέλεση της συνάρτησης στην οποία υπάρχει* και η εκτέλεση συνεχίζεται στο σημείο που κλήθηκε· όσες εντολές ακολουθούν τη **return** δεν θα εκτελεσθούν. Ας δούμε ένα

Παράδειγμα

Για να υπολογίσουμε τη μέγιστη από δύο ακέραιες τιμές γράφουμε την παρακάτω συνάρτηση:

```
int max( int x, int y )
{
```

```
   int fvx;

   if ( x > y )  fvx = x;
          else  fvx = y;
   return fvx;
} // max
```

Ένας άλλος τρόπος να τη γράψουμε είναι ο εξής:

```
int max( int x, int y )
{
   if ( x > y )  return x;
          else  return y;
} // max
```

που είναι οικονομικότερος χωρίς να γίνεται λιγότερο καθαρή η λογική της συνάρτησης. Για δες όμως άλλον έναν τρόπο:

```
int max( int x, int y )
{
   if ( x > y )  return x;
   return y;
} // max
```

Εδώ τι γίνεται; Αν $x > y$ τότε θα εκτελεσθεί η "`return x`" και έτσι η "`return y`" δεν θα εκτελεσθεί ποτέ· αν δεν ισχύει η $x > y$ τότε η "`return x`" θα αγνοηθεί και θα εκτελεσθεί η "`return y`". Άρα, όλα πάνε μια χαρά.

Και οι τρεις μορφές είναι σωστές, αλλά ποια είναι προτιμότερη; Θα προτιμήσουμε την πρώτη διότι συμμορφώνεται με τον κανόνα:

- ***Κάθε συνάρτηση θα πρέπει να έχει μια είσοδο και μια έξοδο (return).***

Αυτό θα μας διευκολύνει σημαντικά στην απόδειξη ορθότητας.

7.3. Μια Ιστορία με Συναρτήσεις

Μας δίνεται το εξής πρόβλημα:

> *Να γραφεί ένα πρόγραμμα που θα διαβάζει τρεις ακέραιους, x1, x2, x3 και θα τους τυπώνει κατ' αύξουσα τάξη.*

Η πιο απλή σκέψη είναι να εξετάσουμε όλες τις δυνατές (6) περιπτώσεις. Ένας άλλος τρόπος είναι:

- βρες τον ελάχιστο και τύπωσέ τον πρώτο,
- βρες το μέγιστο και τύπωσέ τον τελευταίο.

Και ο ενδιάμεσος; Αυτός είναι ο:

$$x1 + x2 + x3 - \text{Μέγιστος} - \text{Ελάχιστος}$$

Ας δούμε τώρα πώς θα υπολογίσουμε το μέγιστο. Θα μπορούσαμε να γράψουμε μια συνάρτηση:

$max3: \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Αυτή θα μεταφραστεί σε μια συνάρτηση C++ με τρεις παραμέτρους και αν προσπαθήσεις να τη γράψεις θα δεις ότι πρέπει να κάνεις αρκετές συγκρίσεις. Ένας άλλος τρόπος είναι να χρησιμοποιήσουμε τη:

$max: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

–που μεταφράσαμε σε μια συνάρτηση της C++ στην προηγούμενη παράγραφο– και να κάνουμε συγκρίσεις ανά δύο: αν $max(x1, x2)$ είναι ο μέγιστος από τους $x1$, $x2$, τότε ο μέγιστος από τους $x1$, $x2$, $x3$ είναι ο $max(max(x1, x2), x3)$.

Με τον ίδιο τρόπο μπορούμε να γράψουμε και να χρησιμοποιήσουμε μια

$min: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

για το ελάχιστο:

```
   int min( int t, int u )
   {
      int fvn;

      if ( t < u )  fvn = t;
              else  fvn = u;
      return fvn;
   } // min
```

Δες ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;

   // max -- Επιστρέφει την τιμή της μέγιστης των παραμέτρων
   int max( int x, int y )
   {
      int fvx;

      if ( x > y )  fvx = x;
              else  fvx = y;
      return fvx;
   } // max

   // min -- Επιστρέφει την τιμή της ελάχιστης των παραμέτρων
   int min( int t, int u )
   {
      int fvn;

      if ( t < u )  fvn = t;
              else  fvn = u;
      return fvn;
   } // min

int main()
{
   int x1, x2, x3;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   cout << min( min(x1, x2), x3 ) << "  "
        << ( x1+x2+x3 - min(min(x1,x2),x3) - max(max(x1,x2),x3) )
        << "  " << max( max(x1, x2), x3 ) << endl;
} // main
```

Ας δούμε τώρα πώς δουλεύει αυτό το πρόγραμμα. Οι συναρτήσεις γράφτηκαν *πριν από τις κλήσεις τους*, που υπάρχουν στη `main`. Αλλά η εκτέλεση αρχίζει από την πρώτη εντολή της `main` και βλέπουμε στην οθόνη μας:

```
 Δώσε τρεις ακέραιους: 12 43 5<enter>
```

Σύμφωνα με αυτά που ξέρουμε: το "`12`" θα γίνει τιμή της *x1*, το "`43`" της *x2* και το "`5`" της *x3*.

Ας πούμε ότι τα ορίσματα της "`cout << ...`" υπολογίζονται με τη σειρά που γράφονται[1]. Το πρώτο που θα γίνει είναι να υπολογιστεί η κλήση:

```
   min( min(x1, x2), x3 )
```

Οι `min(x1, x2)` και `x3` είναι οι **πραγματικές παράμετροι** (actual parameters) ή **ορίσματα** (arguments) που αντιστοιχούν στις **τυπικές παραμέτρους** (formal parameters) της συνάρτησης *min*: Η `min(x1, x2)` αντιστοιχεί στην `t` και η `x3` στη `u`.

- ***Ο υπολογισμός κλήσης μιας συνάρτησης ξεκινάει από τον υπολογισμό των πραγματικών παραμέτρων.***

Για τον υπολογισμό της δεύτερης παραμέτρου δεν χρειάζεται κάτι ιδιαίτερο: απλώς παίρνουμε από τη μνήμη την τιμή της *x3* (5). Η πρώτη παράμετρος `min(x1, x2)` είναι μια

[1] Και να μην ισχύει αυτό, τα πράγματα δεν αλλάζουν.

κλήση συνάρτησης (και πάλι της *min()*)· εδώ έχουμε τις αντιστοιχίες: `x1` (με τιμή 12) στην `t` και `x2` (με τιμή 43) στη `u`. Ας δούμε πώς θα γίνει ο υπολογισμός της.

Κατ' αρχάς παραχωρείται στη συνάρτηση *min()* μνήμη για όλα τα τοπικά αντικείμενα:

- μια θέση μνήμης για τιμές τύπου `int`: τυπική παράμετρος `t`,
- μια θέση μνήμης για τιμές τύπου `int`: τυπική παράμετρος `u`,
- μια θέση μνήμης για τιμές τύπου `int`: μεταβλητή `fvn`.

Στη συνέχεια αντιγράφονται οι τιμές των πραγματικών παραμέτρων στις αντίστοιχες τυπικές. Έτσι, η `t` παίρνει τιμή "`12`" (από τη `x1`) και η `u` τιμή "`43`" (από τη `x2`).

Μετά από αυτό εκτελούνται οι εντολές που υπάρχουν στο σώμα της συνάρτησης μέχρι να βρεθεί εντολή **return**. Στην περίπτωσή μας εκτελείται η **ifelse** από την οποία η `fvn` παίρνει τιμή "`12`".

Με την εκτέλεση της **return** διακόπτεται η εκτέλεση της συνάρτησης. Η συνάρτηση επιστρέφει τη μνήμη που της παραχωρήθηκε (*t*, *u*, *fvn*). Η τιμή της παράστασης που υπάρχει στη **return** (στην περίπτωσή μας της "`fvn`") αντικαθιστά την κλήση της συνάρτησης.

Έτσι η "`min(min(x1, x2), x3)`" γίνεται: "`min(12, 5)`". Για τον υπολογισμό της ξαναγίνονται τα ίδια: της παραχωρείται μνήμη (για τις *t*, *u*, *fvn*), αντιγράφεται το "`12`" στην *t* και το "`5`" στη *u*, εκτελείται η **ifelse** και η *fvn* παίρνει τιμή "`5`". Με την εκτέλεση της "`return fvn`" τελειώνει η εκτέλεση της *min()* και η κλήση "`min(min(x1, x2), x3)`" αντικαθίσταται από το "`5`" (που είναι και η πρώτη τιμή που θα τυπωθεί).

Ολόκληρη αυτή η ιστορία θα επαναληφθεί και όταν θα έρθει η ώρα για τον υπολογισμό του τρίτου ορίσματος της "`cout << ...`", όπου έχουμε:

```
. . . x3 - min( min(x1,x2),x3 ) - max(. . .
```

Παρόμοια θα γίνουν και με τη *max()*.

Ας δούμε τώρα δύο ακόμη σημεία σχετικά με το πρόγραμμά μας:

1. Σε σχέση με το παράδειγμα, θα πρέπει να σημειώσουμε και το εξής: η κλήση και η εκτέλεση μιας συνάρτησης καταναλώνει συνήθως κάποιο υπολογίσιμο χρόνο. Συνεπώς πρέπει να αποφεύγουμε τις αναφορές σε συναρτήσεις που δεν είναι απαραίτητες, π.χ. στις περιπτώσεις που οι αναφορές στη συνάρτηση παίρνουν τις ίδιες πραγματικές παραμέτρους. Έτσι είναι προτιμότερο να γράψουμε τη `main()` ως εξής:

```
int main()
{
   int x1, x2, x3;
   int xmin, xmax;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   xmin = min( min(x1, x2), x3 );
   xmax = max( max(x1, x2), x3 );
   cout << xmin << "  " << x1 + x2 + x3 - xmin - xmax << "  "
        << xmax << endl;
} // main
```

Δηλαδή, να δηλώσουμε δυο μεταβλητές:

```
   int xmin, xmax;
```

στις οποίες να αποθηκεύσουμε αντίστοιχα τις "`min(min(x1, x2), x3)`" και "`max(max(x1, x2), x3)`". Στη συνέχεια, στην εντολή εξόδου χρησιμοποιούμε από δύο φορές τη *xmin* και την *xmax* αντί να ξανακαλούμε τις *min()* και *max()*. Έτσι, έχουμε δυο μόνον αναφορές στη *min* και δυο στη *max*, αντί για τέσσερις που είχαμε αρχικά.

Πλαίσιο 7.1

Ορισμός Συνάρτησης

- ξεκινάει με το όνομα τύπου του αποτελέσματος,
- στη συνέχεια ακολουθεί το όνομα της συνάρτησης, που είναι σαν όλα τα ονόματα της C++,
- μετά, μέσα σε παρενθέσεις, παρατίθενται τα τυπικά ορίσματα, αν υπάρχουν, και
- τέλος υπάρχει η μια σύνθετη εντολή.

Ο ορισμός μιας συνάρτησης είναι ένα κομμάτι προγράμματος που περιγράφει τον τρόπο που θα υπολογισθεί η τιμή της συνάρτησης όταν κληθεί. Για τον υπολογισμό της τιμής η συνάρτηση μπορεί να εξοπλισθεί με τοπικά αντικείμενα (σταθερές, μεταβλητές κλπ). Επικοινωνεί με το πρόγραμμα ή τη συνάρτηση, που την καλεί, με παραμέτρους ή καθολικά αντικείμενα.

Μέσα στο σώμα της συνάρτησης πρέπει να υπάρχει μια τουλάχιστον εντολή `return` *Π*, όπου *Π* μια παράσταση που να ορίζει την τιμή που επιστρέφει.

2. Οι προγραμματιστές που έχουν πείρα σε C δεν θα έγραφαν το πρόγραμμα όπως το γράψαμε, αλλά ως εξής:

```
#include <iostream>
using namespace std;

   int max( int x, int y );
   int min( int t, int u );

int main()
{
   int x1, x2, x3;

   cout << " Δώσε τρεις ακέραιους: ";  cin >> x1 >> x2 >> x3;
   cout << min( min(x1, x2), x3 ) << "  "
        << ( x1+x2+x3 - min(min(x1,x2),x3) - max(max(x1,x2),x3) )
        << "  " << max( max(x1, x2), x3 ) << endl;
} // main
   // max -- Επιστρέφει την τιμή της μέγιστης των παραμέτρων
   int max( int x, int y )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

   // min -- Επιστρέφει την τιμή της ελάχιστης των παραμέτρων
   int min( int t, int u )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Και αυτό που είπαμε, ότι μια συνάρτηση πρέπει να ορίζεται πριν από την κλήση της, δεν ισχύει; Θα το διατυπώσουμε κάπως πιο ελαστικά:

♦ ***Πριν από οποιαδήποτε κλήση μιας συνάρτησης θα πρέπει να υπάρχει ο ορισμός ή, απλώς, η δήλωσή της.***

Οι δηλώσεις των δύο συναρτήσεων θα μπορούσαν να γίνουν και ως εξής:

```
   int max( int, int );
   int min( int, int );
```

ή ακόμη:

```
   int max( int, int ), min( int, int );
```

7.4. Η Συνάρτηση στο Πρόγραμμα

Στην προηγούμενη παράγραφο είδαμε τι περίπου γίνεται με τις συναρτήσεις. Τώρα θα τα ξαναπούμε αλλά σε μεγαλύτερη έκταση. Θα πρέπει όμως να σου δώσουμε μια συμβουλή: *Αν δεν καταλάβεις τα πάντα από αυτήν την παράγραφο, μην προχωρήσεις παρακάτω*. Πέρασε τα προγράμματα στον ΗΥ και σιγουρέψου ότι παίρνεις ακριβώς τα ίδια αποτελέσματα με αυτά που σου δίνουμε. Αν έχεις αμφιβολία για οτιδήποτε, κάνε μερικά πειράματα μέχρι να τη λύσεις.

Ένα πρόγραμμα της C++ είναι ένα *σύνολο από συναρτήσεις*. Μια από αυτές τις συναρτήσεις έχει το όνομα "`main`" και από αυτήν αρχίζει η εκτέλεση του προγράμματος (δες την §7.6).

7.4.1 Εμβέλεια και Χρόνος Ζωής

Κάθε συνάρτηση είναι μια *σύνθετη εντολή* (σώμα της συνάρτησης) με μια επικεφαλίδα. Στην επικεφαλίδα, δηλώνεται και ένα όνομα που χαρακτηρίζει τη συνάρτηση. Η σύνθετη εντολή αποτελείται από τις δηλώσεις των σταθερών, των τύπων και των μεταβλητών και τις άλλες εντολές. Στις εντολές αυτές μπορεί να περιλαμβάνονται *άλλες σύνθετες εντολές με το περιεχόμενο που περιγράφουμε* (δηλ. δηλώσεις κλπ).

Κάθε οντότητα (σταθερά, μεταβλητή κλπ) που δηλώνεται μέσα σε μια συνάρτηση είναι **τοπική** (local) στη συνάρτηση. *Τοπικά είναι και τα ονόματα των παραμέτρων μιας συνάρτησης*: τα χειριζόμαστε σαν να δηλώνονται στη σύνθετη εντολή που ακολουθεί την επικεφαλίδα. Το όνομα μιας τοπικής οντότητας είναι άγνωστο έξω από τη συνάρτηση που δηλώνεται (εκεί μπορεί να χρησιμοποιείται για κάποια άλλη οντότητα).

Για παράδειγμα, ας ξαναγυρίσουμε στο πρόγραμμα της §7.4. Εδώ:

- Οι μεταβλητές *x*1, *x*2, *x*3 είναι τοπικές στη `main()`· π.χ. αν βάλεις μια "`cout << x1 << x2 << x3`" μέσα στην *max()* (ή μέσα στη *min()*) θα πάρεις μήνυμα λάθους από τον μεταγλωττιστή (unknown identifier ή κάτι παρόμοιο), που θα σου πει ότι δεν ξέρει αυτά τα ονόματα.
- Η *fvx* δηλώνεται στην *max()* και είναι γνωστή μόνο μέσα σε αυτήν. Αν προσπαθήσεις να χρησιμοποιήσεις την *fvx* στη `main()` ή στη *min()* θα πάρεις μήνυμα λάθους από το μεταγλωττιστή. Παρομοίως, η *fvn* είναι γνωστή μόνο στη *min()*.
- Ξαναγράφουμε τη *min()* ως εξής:

```
int min( int x, int y )
{
   int fvn;

   if ( x < y )  fvn = x;
          else  fvn = y;
   return fvn;
} // min
```

Ας έρθουμε στις *x*, *y*: υπάρχουν παράμετροι με το όνομα αυτό και στη *max()* και στη

Πλαίσιο 7.2

Κανόνας Εμβέλειας

Κάθε δήλωση ισχύει μόνον στη συνάρτηση όπου έγινε. Αυτό ισχύει και για τις παραμέτρους της συνάρτησης.

Κανόνας Διάρκειας Ζωής

Κάθε αντικείμενο μιας συνάρτσης υπάρχει όσο εκτελείται η συνάρτηση όπου έχει δηλωθεί.

min()· δεν γίνεται μπέρδεμα; Όχι! Οι παράμετροι της *min()* είναι γνωστές μόνο μέσα στη *min()*, ενώ οι παράμετροι της *max* είναι γνωστές μόνο μέσα στη *max()*.

Λέμε ότι η **εμβέλεια** (scope)

- των ονομάτων *x1*, *x2*, *x3* είναι η `main()`,
- των *x*, *y* (που δηλώνονται στη *max()*) και *fvx* είναι η *max* και
- των *x*, *y* (που δηλώνονται στη *min()*) και *fvn* είναι η *min*.

Τώρα ας έρθουμε σε ένα άλλο πρόβλημα: Για πόσο χρόνο «ζουν» οι μεταβλητές μιας συνάρτησης; Στην προηγούμενη παράγραφο λέγαμε ότι: «[*όταν κληθεί η min()*] *κατ' αρχάς παραχωρείται στη συνάρτηση min() μνήμη για όλα τα τοπικά αντικείμενα: α) μια θέση μνήμης για τιμές τύπου* `int` *για την τυπική παράμετρο t, β) μια θέση μνήμης για τιμές τύπου* `int` *για την τυπική παράμετρο u, γ) μια θέση μνήμης για τιμές τύπου* `int` *για τη μεταβλητή fvn*» και «*Με την εκτέλεση της* `return` *διακόπτεται η εκτέλεση της συνάρτησης. Η συνάρτηση επιστρέφει τη μνήμη που της παραχωρήθηκε (t, u, fvn).*» Γενικώς ισχύει ο «Κανόνας Διάρκειας Ζωής» που βλέπεις στο Πλ. 7.2.

Αργότερα θα επεκτείνουμε τους κανόνες του Πλ. 7.2.

7.4.2 Παράμετροι

Ας δούμε τώρα τώρα ένα άλλο θέμα. Λέγαμε στην προηγούμενη παράγραφο ότι, αφού υπολογισθούν οι τιμές των παραγματικών παραμέτρων και παραχωρηθεί η μνήμη που χρειάζεται για τις τυπικές, «*αντιγράφονται οι τιμές των πραγματικών παραμέτρων στις αντίστοιχες τυπικές.*» Η αντιγραφή γίνεται αφού πρώτα γίνει η κατάλληλη *αλλαγή τύπου*, αν αυτή είναι δυνατή βέβαια. Ας δούμε ένα παράδειγμα. Ας πούμε ότι έχουμε:

```
int ai( int x )
{    . . .
   cout << x << endl; . . . }

int al( long int x )
{    . . .
   cout << x << endl; . . . }

int ac( char x )
{    . . .
   cout << x << endl; . . . }

int ab( bool x )
{    . . .
   cout << x << endl; . . . }

int ad( double x )
{    . . .
   cout << x << endl; . . . }

int af( float x )
{    . . .
   cout << x << endl; . . . }
```

και καλούμε:

```
   x = ai( 65.789 );
   x = al( 65.789 );
   x = ac( 65.789 );
   x = ab( 65.789 );
   x = ad( 65.789 );
   x = af( 65.789 );
```

Όπως βλέπεις, σε όλες τις κλήσεις, άσχετα από τον τύπο του τυπικού ορίσματος, έχουμε βάλει ως όρισμα μια τιμή τύπου `double`.

Οι εντολές εξόδου που υπάρχουν στις συναρτήσεις θα δώσουν:

Πλαίσιο 7.3

Κλήση Συνάρτησης

Ο υπολογισμός κλήσης μιας συνάρτησης γίνεται ως εξής:

- πρώτα υπολογίζονται οι τιμές των πραγματικών παραμέτρων –για τις παραμέτρους τιμής,
- οι τιμές αυτές αντιγράφονται στις αντίστοιχες τυπικές παραμέτρους, αφού γίνουν μετατροπές τύπου, όπου είναι απαραίτητο,
- εκτελούνται οι εντολές του υποπρογράμματος και υπολογίζεται η τιμή της συνάρτησης,
- η τιμή της συνάρτησης επιστρέφεται με την εντολή **return**.

Αυτή η τιμή αντικαθιστά την κλήση της συνάρτησης στην παράσταση για να γίνουν οι άλλες πράξεις.

```
65
65
A
1
65.789
65.789
```

Καταλαβαίνεις ότι, όπου χρειάστηκε, έγιναν οι μετατροπές τύπου. Π.χ. στην πρώτη η *x* πήρε τιμή `int`(65.789) = 65, στην τρίτη `char`(`int`(65.789)) = `'A'`, στην τέταρτη `bool`(`int`(65.789)) = `true` = 1.

Θα μπορούσαμε να θεωρήσουμε το πέρασμα παραμέτρου σαν εντολή εκχώρησης; Π.χ. για την κλήση:

```
. . .min(x1, x2). . .
```

θα μπορούσαμε να πούμε ότι έχουμε τις εντολές:

```
   t = x1;  u = x2;
```

Με αυτά που έχουμε μάθει μέχρι τώρα, αυτό δεν είναι λάθος. Πάντως πιο σωστό είναι να το παρομοιάσουμε με δήλωση των *t*, *u* με αρχικές τιμές:

```
int  t( x1 ), u( x2 );
```

Μετά από αυτές τις αρχικές τιμές μπορούμε να αλλάζουμε τις τιμές των παραμέτρων, αλλά οι αλλαγές αυτές *δεν περνούν στη συνάρτηση που έκανε την κλήση*. Για παράδειγμα, ας πούμε ότι έχουμε:

```
#include <iostream>
using namespace std;

long int succ( long int n )
{
   n = n + 1;
   return n;
} // succ

int main()
{
   int x = 100, y;

   y = succ( x );
   cout << x << "  " << y << endl;
} // main
```

που θα δώσει:

```
100  101
```

Με την κλήση: "`y = succ(x)`" η τιμή της *x* (= 100) της `main()` έγινε αρχική τιμή της *n* της *succ()*. Στη *succ()* η τιμή της *n* αυξήθηκε κατά 1 και η αυξημένη τιμή (101) επιστράφηκε

ως τιμή της συνάρτησης και αποθηκεύτηκε στην *y* της `main`. Όπως φαίνεται και από το αποτέλεσμα, *η τιμή της x δεν άλλαξε*. Πάντως, αν έχεις καταλάβει όσα είπαμε για το πώς γίνεται το πέρασμα των τιμών των παραμέτρων, τα παραπάνω είναι αυτονόητα.

Οι παράμετροι που λειτουργούν σαν «μονόδρομοι», μεταφέροντας στοιχεία από την κλήση προς τη συνάρτηση μόνον λέγονται **παράμετροι τιμής** (value parameters). Αργότερα θα μάθουμε ότι υπάρχουν μηχανισμοί α) για να παίρνουμε τις αλλαγές τιμών των παραμέτρων[2] β) για να μην επιτρέπουμε τέτοιες αλλαγές.

7.4.3 Αρχικές Τιμές Μεταβλητών

Στο Κεφ. 2 λέγαμε ότι «*η C++ σου επιτρέπει μαζί με τη δήλωση να δώσεις στη μεταβλητή σου και αρχική τιμή.*» Ας δούμε τώρα αυτήν τη δυνατότητα πιο εκτεταμένα.

Για να δηλώσουμε μια μεταβλητή *v* τύπου *T* μπορούμε να δώσουμε:

```
T v( Π );
```

όπου *Π* μια παράσταση που μπορεί να περιέχει σταθερές, μεταβλητές που έχουν ήδη τιμή και συναρτήσεις, αν φυσικά η δήλωσή μας βρίσκεται μέσα στην εμβέλειά τους. Η τιμή της *Π* υπολογίζεται με τις τιμές που έχουν οι μεταβλητές που υπάρχουν σε αυτήν όταν εκτελείται η δήλωση, όταν δηλαδή παραχωρείται μνήμη για τη *v*.

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;

int f( double x )
{
   return (2*x+1)/(x*x+1);
} // f

int main()
{
   double x( 1.5 );
   int n, a( 1 );
   double q( sqrt(2) ), qp1( q + 1 + a/2.0 ), r( f(x/3) );

   cout << " n = " << n << "   a = " << a << endl;
   cout << q << "  " << qp1 << "  " << f(q) << "  " << r << endl;
   a = 2;
   cout << qp1 << endl;
} // main
```

που δίνει:

```
n = 50920484   a = 1
1.41421  2.91421  1  1
2.91421
```

Ας εξετάσουμε προσεκτικά τα αποτελέσματα:

- Στην πρώτη γραμμή παίρνουμε τις τιμές της *n* και της *a* μόλις αρχίσει η εκτέλεση του προγράμματος. Η *n* είναι αόριστη και η τιμή της είναι τυχαία. Η *a* όμως έχει αρχική 1.
- Η αρχική τιμή της *q* είναι *sqrt*(2) (≈1.41421). Η αρχική τιμή της *qp*1 ορίζεται με χρήση των τιμών των *q* και *a* που είναι ήδη ορισμένες. Η τιμή της *r* καθορίζεται με κλήση της συνάρτησης *f* (η `(2*x+1)/(x*x+1)` έχει τιμή τύπου `double` (1.6) αλλά αυτή μετατρέπεται σε `int` (1) αφού αυτός είναι ο τύπος της συνάρτησης).
- Παρομοίως μπορείς να ερμηνεύσεις και την τιμή (1) της *f*(*q*).

[2] Ήδη στην §4.5 μάθαμε ότι με την "`cin.get( a );`" αλλάζει η τιμή της *a*.

- Αλλάξαμε την τιμή της *a* σε 2. Η τιμή της *qp*1 δεν αλλάζει, όπως άλλωστε το περιμένουμε.

Οι τοπικές μεταβλητές μιας συνάρτησης μπορεί να έχουν αρχική τιμή την τιμή κάποιας παραμέτρου. Δες πώς μπορούμε να γράψουμε τη *max()*:

```
int max( int x, int y )
{
   int fvx( y );

   if (x > y)  fvx = x;

   return fvx;
} // max
```

7.5. Η Συνάρτηση "main"

Η `main()` δεν είναι τίποτε άλλο από μια ακόμη συνάρτηση. Αλλά η C++ βάζει μια υποχρέωση στον προγραμματιστή:

- ***Σε κάθε πρόγραμμα θα πρέπει να υπάρχει μια συνάρτηση*** `int main`***· από αυτήν αρχίζει η εκτέλεση του προγράμματος.***[3]

Δεν θα πρέπει να έχει και μια εντολή **return**; Βεβαίως! Και, πιθανότατα, η C++ που δουλεύεις να βγάζει και σχετική προειδοποίηση, αν ακολουθείς το παράδειγμά μας και δεν βάζεις στο τέλος της `main()` μια "`return 0`". Πάντως, το πρότυπο της C++ λέει ότι: *αν η εκτέλεση φτάσει στο τέλος της* `main()` (*επειδή όλα πήγαν καλά και δεν υπήρξε εντολή* **return**) *θα πρέπει να εκτελείται η εντολή* "`return 0`". Αυτό το "`0`" επιστρέφεται στο ΛΣ και σημαίνει ότι η εκτέλεση του προγράμματος ολοκληρώθηκε επιτυχώς. Το ΛΣ σου δίνει τρόπους για να ελέγξεις αυτήν την τιμή.

Αργότερα θα μάθουμε ότι η `main()` (μπορεί να) έχει και παραμέτρους.

7.6. Παράμετρος "unsigned";

Πριν προχωρήσουμε στα παραδείγματά μας θα αναδείξουμε ένα πρόβλημα που μπορεί να σου προκύψει αν βάζεις παραμέτρους `unsigned` (π.χ. `unsigned int`) στις συναρτήσεις σου.

Ας πούμε ότι γράφουμε μια συνάρτηση που υπολογίζει την «ακέραιη τετραγωνική ρίζα φυσικού αριθμού» –μόνον με προσθέσεις– με βάση αυτά που είδαμε στην άσκ. 6-12:

```
unsigned int intSqrt( unsigned int x )
{
   int  z( 0 ), u( 1 );
// x >= 0
   while ( u <= x )     // I: (z² <= x) && (u = (z+1)²)
   {
      z = z + 1;
      u = u + z + z + 1;
   }
// z² <= x < (z+1)²
   return z;
} // intSqrt
```

Τύπος παραμέτρου; `unsigned int`, τι πιο φυσιολογικό! Δες τώρα δυο δοκιμές χρήσης:

```
   n = 1024;
   cout << n << "  " << intSqrt(n) << endl;
   n = -1024;
   cout << n << "  " << intSqrt(n) << endl;
```

Αποτέλεσμα:

[3] Στα προγράμματα για Windows η αντίστοιχη συνάρτηση ονομάζεται *WinMain*.

```
1024  32
-1024  699732
```

Δηλαδή, δούλεψε και στη δεύτερη δοκιμή, αλλά... τι έβγαλε! Ξανακάνουμε τη δεύτερη δοκιμή αλλά βάζουμε μια εντολή εξόδου μέσα στη συνάρτηση:

```
unsigned int intSqrt( unsigned int x )
{
   int  z( 0 ), u( 1 );
cout << x << endl;
   . . .
```

Αποτέλεσμα:

```
4294966272
-1024  699732
```

Όπως βλέπεις, το -1024 περνάει στη συνάρτηση ως 4294966272 και όλα δουλεύουν χωρίς οποιαδήποτε ένδειξη για το ότι κάτι δεν πάει καλά. Αργότερα θα καταλάβεις πώς και γιατί συμβαίνουν αυτά.

Προς το παρόν:

- ***Μην βάζεις στις συναρτήσεις σου παραμέτρους τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

Στην περίπτωσή μας θα έχουμε:

```
unsigned int intSqrt( int x )
{
   int  z( 0 ), u( 1 );

   if ( x >= 0 )
   { // x >= 0
      while ( u <= x )     // I: (z² <= x) && (u = (z+1)²)
      { . . .
```

Στα παραδείγματα στη συνέχεια θα δεις έναν τρόπο αντίδρασης αν δεν ισχύει η προϋπόθεση. Αργότερα θα δούμε και άλλους.

7.7. Παραδείγματα

Να δούμε τώρα ολοκληρωμένα προγράμματα που περιέχουν ορισμούς και χρήσεις συναρτήσεων.

Παράδειγμα 1

Να γραφεί πρόγραμμα που να διαβάζει δυο ακέραιους, *m*, *n* και να υπολογίζει και να τυπώνει το πλήθος των συνδυασμών *m* αντικειμένων ανά *n*.

Όπως είναι γνωστό από τα Μαθηματικά, αυτό είναι:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

όπου, το *n*! (*n* παραγοντικό), ορίζεται ως εξής:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 1 \end{cases} \quad \text{ή ισοδυνάμως:} \quad n! = \begin{cases} 1 & n = 0 \\ 1 \cdot \ldots \cdot (n-1) \cdot n & n \geq 1 \end{cases}$$

Αν είχαμε μια συνάρτηση με το όνομα, ας πούμε, *factorial()*, που να μας υπολογίζει το παραγοντικό του ορίσματος, το πρόγραμμά μας θα ήταν πολύ απλό:

```
   cin >> m >> n;
   comb = factorial(m) / (factorial(n) * factorial(m-n));
   cout << comb << endl;
```

Ας τη γράψουμε, μεταφράζοντας τον δεύτερο ορισμό. Ξεκινούμε από την επικεφαλίδα. Ποια είναι τα πεδία ορισμού και τιμών της συνάρτησής μας:

!: $\mathbb{N} \rightarrow \mathbb{N}^*$

Πώς θα μεταφράσουμε τα $\mathbb{N}$, $\mathbb{N}^*$ στη C++; Προφανώς στον **unsigned int** και για τα δύο σύνολα. Επειδή όμως το *n*! αυξάνεται πολύ γρήγορα, καθώς αυξάνεται το *n*, καλύτερα να βάλουμε σύνολο τιμών το **unsigned long int**. Θα έπρεπε δηλαδή να γράψουμε μια συνάρτηση:

factorial: unsigned int → unsigned long int

αλλά μετά από αυτά που είδαμε στην προηγούμενη παράγραφο θα γράψουμε μια:

factorial: int ↛ unsigned long int

Προχωρούμε μεταφράζοντας τον μηχανισμό:

```
unsigned long int factorial( int a )
{
   unsigned long int fv;

   if ( a < 0 )
   {
      cout << " η factorial κλήθηκε με αρνητικό" << endl;
      exit( EXIT_FAILURE );
   }
   else
   {
      if ( a == 0 )
         fv = 1;
      else
         Υπολόγισε το fv = 1*2*...*(a-1)*a;
   }
    return fv;
} // factorial
```

Και αυτό το "**exit(EXIT_FAILURE)**" τι είναι; Η *exit()* είναι μια συνάρτηση που δεν επιστρέφει τιμή –σαν την *assert()*– που η δήλωσή της υπάρχει στο **cstdlib**. Η εκτέλεσή της έχει ως αποτέλεσμα τη *διακοπή της εκτέλεσης του προγράμματος* (όχι μόνον της συνάρτησης). Η τιμή της παραμέτρου που βάζουμε πηγαίνει στο ΛΣ και μπορεί να ελεγχθεί. Συνήθως, μια μη μηδενική τιμή της παραμέτρου δείχνει ότι η εκτέλεση του προγράμματος διακόπηκε επειδή υπήρξε κάποιο πρόβλημα. Στο **cstdlib** ορίζεται επίσης η σταθερά **EXIT_FAILURE** ως "1".

Με βάση τα παραπάνω, μπορούμε να γράψουμε μια πιο απλή αλλά ισοδύναμη μορφή:

```
unsigned long int factorial( int a )
{
   unsigned long int fv;

   if ( a < 0 )
   {
      cout << " η factorial κλήθηκε με όρισμα " << a << endl;
      exit( EXIT_FAILURE );
   }
   if ( a == 0 )
      fv = 1;
   else
      Υπολόγισε το fv = 1*2*...*(a-1)*a;
   return fv;
} // factorial
```

Από εδώ και πέρα κάπως έτσι θα βάζουμε στις συναρτήσεις μας τον έλεγχο προδιαγραφών.

Στο Πλ. 6.2β είδαμε πώς υπολογίζουμε ένα γινόμενο. Για την περίπτωσή μας:

```
      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
```

Οι *fv* και *k* είναι *τοπικές* μεταβλητές, που δηλώνονται *μέσα στη συνάρτηση*.

Πριν δώσουμε το τελικό πρόγραμμα, ας παρατηρήσουμε ότι ο διαχωρισμός της *a* == 0 είναι άχρηστος μια και ο υπολογισμός της περίπτωσης *a* > 0 μας δίνει σωστό αποτέλεσμα και για *a* == 0.

```
#include <iostream>
#include <cstdlib>
using namespace std;

   // factorial -- Υπολογίζει το a!
   unsigned long int factorial( int a )
   {
      unsigned long int fv;
      int k;

      if ( a < 0 )
      {
         cout << " η factorial κλήθηκε με όρισμα " << a << endl;
         exit( EXIT_FAILURE );
      }
      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
      return fv;
   } // factorial

int main()
{
   int m, n, comb;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   if ( n < 0 || m < 0 )
      cout << " θέλω m >= 0 και n >= 0 " << endl;
   else if ( m < n )
      cout << " θέλω m >= n" << endl;
   else
   {
      comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb << endl;
   }
} // main
```

Κοίταξε την εντολή:

```
      comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
```

Εδώ έχεις μια παράσταση με τρεις κλήσεις της συνάρτησης *factorial()*. Ας δούμε ένα παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m <= n <= 50: 5 2
```

Με την εκτέλεση της `cin >> m >> n` θα έχουμε: *m* = 5 και *n* = 2:

- Ο υπολογισμός της `factorial( m )` θα γίνει ως εξής: Η *a* της *factorial()* θα πάρει την τιμή του *m* = 5. Θα εκτελεσθούν οι εντολές της *factorial()* και τελικώς θα υπολογισθεί η τιμή που επιστρέφει (= 120).
- Παρομοίως θα γίνει ο υπολογισμός της κλήσης `factorial( n )` (= 2).
- Τέλος, για να υπολογισθεί η `factorial( m - n )`, πρώτα υπολογίζεται η τιμή του *m* – *n* (= 3). Αυτή γίνεται τιμή της *a* στην *factorial()* και μετά την εκτέλεση των εντολών της, επιστρέφεται η τιμή 6.

Η παράσταση που πρέπει να υπολογισθεί είναι πια η[4]: 120 / (2 * 6) και η *comb* παίρνει την τιμή 10:

[4] Ο προσεκτικός αναγνώστης θα έχει σημειώσει ότι κάνουμε περισσότερες πράξεις από όσες χρειάζεται. Αλλά, αυτό που θέλουμε να δείξουμε είναι η χρήση της συνάρτησης σε ένα πρόγραμμα.

```
Συνδυασμοί των 5 ανά 2 = 10
```

Παράδειγμα 2

Να γραφούν δύο συναρτήσεις που θα υπολογίζουν το Ελάχιστο Κοινό Πολλαπλάσιο (ΕΚΠ, least common multiple, lcm) και το Μέγιστο Κοινό Διαιρέτη (ΜΚΔ, gratest common divisor, gcd) δύο φυσικών αριθμών. Να γραφεί πρόγραμμα που θα διαβάζει δυο φυσικούς αριθμούς και καλώντας τις συναρτήσεις θα δίνει το ΕΚΠ και τον ΜΚΔ τους.

Να σου υπενθυμίσουμε εδώ ορισμένα πράγματα, από την Αριθμητική: Αν x, y φυσικοί αριθμοί τότε ΕΚΠ(x, y)·ΜΚΔ(x, y) = x·y. Αν βρούμε λοιπόν το ένα από τα δύο, τότε το άλλο υπολογίζεται εύκολα.

Αλλά, στην Αριθμητική είχες μάθει μια μέθοδο για να υπολογίζεις τον ΜΚΔ: τον Ευκλείδειο αλγόριθμο[5]. Ας την θυμίσουμε:

```
Πάρε το υπόλοιπο της ακέραιης διαίρεσης x δια y (x % y)
Ψάξε να βρεις τον ΜΚΔ(y, x % y).
Δηλ., βάλε Νέο x = y και Νέο y = x % y.
      ( ΜΚΔ(x,y) = ΜΚΔ(y, x % y) )
Συνέχισε έτσι, μέχρι να βρεις Νέο y == 0.
```

και ας ξεκινήσουμε από τη συνάρτηση για τον ΜΚΔ. Τα παραπάνω γράφονται σε ψευδοκώδικα:

```
while ( y != 0 )          // ΜΚΔ(x,y) = ΜΚΔ(y,x % y)
{
   b = y;                 // Φύλαξε την παλιά τιμή του y
   Νέο y = x % y;
   Νέο x = b;             // παλιά τιμή του y
}
return x;                 // ΜΚΔ(x,0) = x
```

Αυτά μεταφράζονται εύκολα σε C++, αλλά πρώτα να δούμε τα πεδία ορισμού και τιμών της συνάρτησής μας. Ο ΜΚΔ δεν ορίζεται όταν x == y == 0. Έχουμε λοιπόν:

ΜΚΔ: $\mathbb{N} \times \mathbb{N} \setminus \{ (0,0) \} \to \mathbb{N}$

Εδώ έχουμε δύο περιπλοκές.

- Το πεδίο ορισμού είναι καρτεσιανό γινόμενο. Τι κάνουμε στην περίπτωση αυτή; *Βάζουμε στη συνάρτησή μας δύο παραμέτρους!*
- Η συνάρτηση *gcd()* που θα γράψουμε στην C++ θα είναι μερική. Θα πρέπει να εξαιρέσουμε το (0,0).
- Τέλος, η *gcd()* θα είναι μερική και διότι, με βάση αυτά που είπαμε πιο πάνω, θα αποφύγουμε παραμέτρους `unsigned int` και θα χρησιμοποιήσουμε `int`:

```
gcd: int × int ⇸ unsigned int
```

Να η συνάρτηση:

```
unsigned int gcd( int x, int y )
{ // x == x0 && y == y0
   unsigned int  b;

   if ( (x == 0 && y == 0) || x < 0 || y < 0 )
   {
      cout << " η gcd κλήθηκε με " << x << ", " << y << endl;
      exit( EXIT_FAILURE );
   }
// (x != 0 || y != 0) && x >= 0 && y >= 0
   while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
   {
      b = y;  y = x % y;  x = b;
   } // while
```

[5] Ο Ευκλείδης αποκαλεί **ανθυφαίρεση** την πράξη εύρεσης υπολοίπου της διαίρεσης δύο φυσικών αριθμών. Για τον λόγο αυτόν η μέθοδος αυτή ονομάζεται και *ανθυφαιρετική*.

```
      return x;          // ΜΚΔ(x,0) = x
   } // gcd
```

Δες τώρα ολόκληρο το πρόγραμμα και συνεχίζουμε τη συζήτηση.

```
#include <iostream>
#include <cstdlib>
using namespace std;

   // gcd -- Υπολογίζει τον Μέγιστο Κοινό Διαιρέτη των ορισμάτων
   unsigned int gcd( int x, int y )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ

   // lcm -- Υπολογίζει το Ελάχ. Κοινό Πολλαπλάσιο των ορισμάτων
   //        Χρησιμοποιεί την gcd
   unsigned int lcm( int x, int y )
   {                              // ΕΚΠ(x,y)*ΜΚΔ(x,y) = x*y
      unsigned int fv;

      if ( (x == 0 && y == 0) || x < 0 || y < 0 )
      {
         cout << " η lcm κλήθηκε με " << x << ", " << y << endl;
         exit( EXIT_FAILURE );
      }
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      fv = x * y / gcd(x, y);
      return fv;
   } // lcm

int main()
{
   int x1, x2;

   cout << " Δώσε δυο θετικούς ακέραιους: ";   cin >> x1 >> x2;
   if ( x1 <= 0 || x2 <= 0 )
       cout << " Είπα: ΘΕΤΙΚΟΥΣ" << endl;
   else
      cout << " ΕΚΠ = " << lcm( x1, x2 )
           << "  ΜΚΔ = " << gcd( x1, x2 ) << endl;
} // main
```

Σε αυτό το πρόγραμμα βλέπεις τη **main()** και άλλες δυο συναρτήσεις, από τις οποίες η δεύτερη, η *lcm()*, χρησιμοποιεί την πρώτη, την *gcd()*.

Παρατηρήσεις: ▶

1. Αν $x \neq 0$ **||** $y \neq 0$ τότε ο *θετικός ΜΚΔ* υπολογίζεται ως

$$\text{θΜΚΔ}(x, y) = \text{ΜΚΔ}(|x|, |y|)$$

Τροποποίησε τη *gcd()* ώστε να υπολογίζει τον θΜΚΔ.

2. Ναι μεν κάνουμε επίδειξη πώς καλούμε μια συνάρτηση, την *lcm()*, που καλεί μια άλλη, την *gcd()*, αλλά αν θέλουμε να υπολογίσουμε ΕΚΠ και ΜΚΔ δύο φυσικών η *gcd()* θα κάνει τους ίδιους υπολογισμούς δύο φορές. Αργότερα θα το ξανασκεφτούμε. ◀

Παράδειγμα 3

Όπως ξέρουμε, η συνάρτηση τυποθεώρησης **static_cast<int>** αν κληθεί με όρισμα πραγματικό *αποκόπτει* το κλασματικό του μέρος. Το ίδιο κάνει και η **static_cast<long int>**.

Συχνά όμως θέλουμε να *στρογγυλοποιήσουμε* (round) μια πραγματική τιμή στον πλησιέστερο ακέραιο. Ας γράψουμε λοιπόν μια συνάρτηση που θα κάνει αυτή τη δουλειά:

myRound: $\mathbb{R} \to \mathbb{Z}$

Η συνάρτηση που θα γράψουμε θα παίρνει τιμές από το **double** και θα δίνει αποτέλεσμα στον **long int**.[6] Θα μπορεί να χειριστεί οποιαδήποτε τιμή τύπου **double**; Όχι βέβαια!

[6] Στο cmath μπορείς να βρεις την επικεφαλίδα μιας τέτοιας συνάρτησης· είναι η *lround*. Δες τον πίνακα του Παραρτ. D.

Θα μπορεί να στρογγυλοποιήσει τιμές στο διάστημα

(*LONG_MIN* − ½, *LONG_MIN* + ½]

στο *LONG_MIN* αλλά όχι τιμές μικρότερες από αυτές. Παρομοίως, θα μπορεί να στρογγυλοποιήσει τιμές στο διάστημα

[*LONG_MAX* − ½, *LONG_MAX* + ½)

στο *LONG_MAX* αλλά όχι τιμές μεγαλύτερες από αυτές. Επομένως πεδίο ορισμού της συνάρτησης που θα γράψουμε θα είναι το

(*LONG_MIN* − ½, *LONG_MAX* + ½)

και η

```
myRound: double ⇸ long int
```

θα είναι μερική συνάρτηση.

Και τώρα: πώς θα κάνουμε τη στρογγυλοποίηση; Ας πούμε ότι έχουμε το 7.3, που θα πρέπει να στρογγυλοποιηθεί στο 7. Εδώ τα πράγματα είναι απλά: με τη `static_cast<long int>(7.3)` πετυχαίνουμε τον στόχο μας. Αν όμως έχουμε το 7.8; Τώρα σκέψου το εξής τέχνασμα: αν το κλασματικό μέρος είναι μεγαλύτερο από (ή ίσο με) 0.5 (π.χ. 0.8, στον 7.8) – οπότε θα πρέπει να στρογγυλοποιηθεί στον αμέσως μεγαλύτερο ακέραιο (8)– και προσθέσουμε στον αριθμό μας 0.5 αυτός θα φτάσει ή και θα ξεπεράσει τον επόμενο ακέραιο (8.3). Αν λοιπόν πάρουμε τη `static_cast<long int>(7.8 + 0.5)` μας δίνει τη σωστή τιμή (8). Πρόσεξε ότι αυτό το τέχνασμα δεν θα αλλάξει το σωστό υπολογισμό για το 7.3: `static_cast<long int>(7.3 + 0.5) == 7`. Αν η τιμή είναι *αρνητική* τότε *αφαιρούμε* 0.5:

```
         if ( x >= 0 )  fv = static_cast<long int>(x + 0.5);
                  else  fv = static_cast<long int>(x - 0.5);
```

Στη συνέχεια βλέπεις ολόκληρη τη *myRound()* σε ένα πρόγραμμα που τη δοκιμάζει.

```
#include <iostream>
#include <cstdlib>
#include <climits>
using namespace std;

   // myRound -- στρογγυλοποιεί το όρισμα στον πλησιέστερο ακέραιο
   long int myRound( double x )
   {
      long int fv;

      if ( x <= LONG_MIN - 0.5 || LONG_MAX + 0.5 <= x )
      {
         cout << " η myRound κλήθηκε με όρισμα: " << x << endl;
         exit( EXIT_FAILURE );
      }
   // LONG_MIN - 0.5 < x && x < LONG_MAX + 0.5
      if ( x >= 0 )  fv = static_cast<long int>( x + 0.5 );
               else  fv = static_cast<long int>( x - 0.5 );
      return fv;
   } // myRound

int main()
{
   double t;

   cout << " Δώσε έναν πραγματικό. 0 για τέλος: ";   cin >> t;
   while ( t != 0 )
   {
      cout << " myRound(" << t << ") = " << myRound( t ) << endl;
      cout << " Δώσε έναν πραγματικό. 0 για τέλος: ";   cin >> t;
   } // while
   cout << " myRound(" << t << ") = " << myRound( t ) << endl;
} // main
```

Η *myRound()* είναι χρήσιμη συνάρτηση και, αφού στη συνέχεια θα αποδείξεις την ορθότητά της (Ασκ. 7-18), ας δούμε πώς μπορούμε να περιγράψουμε τη σχέση μεταξύ x και *myRound*(x);

Έστω ότι $x \geq 0$. Αν το κλασματικό μέρος, *κλάσμα*(x), είναι: $0 \leq$ *κλάσμα*(x) $< ½$ τότε η x στρογγυλοποιείται με αποκοπή του κλασματικού μέρους, δηλαδή:

$$myRound(x) = x - \text{κλάσμα}(x) \quad \text{ή} \quad \text{κλάσμα}(x) = x - myRound(x)$$

Από αυτήν παίρνουμε: $0 \leq x - myRound(x) < ½$ που ισοδυναμεί με:

$$myRound(x) \leq x < myRound(x) + ½ \quad \text{ή} \quad x - ½ < myRound(x) \leq x$$

Αν $½ \leq$ *κλάσμα*(x) < 1 τότε η x στρογγυλοποιείται στον μεγαλύτερο ακέραιο: είναι σαν να προσθέτουμε στη x το $1 -$ *κλάσμα*(x). Δηλαδή:

$$myRound(x) = x + 1 - \text{κλάσμα}(x) \quad \text{ή} \quad \text{κλάσμα}(x) = x + 1 - myRound(x)$$

Από αυτήν παίρνουμε: $½ \leq x + 1 - myRound(x) < 1$ που ισοδυναμεί με:

$$myRound(x) - ½ \leq x < myRound(x) \quad \text{ή} \quad x < myRound(x) \leq x + ½$$

Αν λοιπόν $x \geq 0$ τότε:

$$myRound(x) - ½ \leq x < myRound(x) + ½ \quad \text{ή} \quad x - ½ < myRound(x) \leq x + ½$$

Με παρόμοιους συλλογισμούς μπορείς να βρεις ότι αν $x < 0$ τότε:

$$myRound(x) - ½ < x \leq myRound(x) + ½ \quad \text{ή} \quad x - ½ \leq myRound(x) < x + ½$$

Παράδειγμα 4

Πολύ συχνά χρειάζεται να στρογγυλοποιήσουμε μια πραγματική τιμή *σε n ψηφία μετά την υποδιαστολή*[7]. Θα γράψουμε λοιπόν μια συνάρτηση *dRound()* που θα κάνει αυτή τη δουλειά:

dRound: $\mathbb{R} \times \mathbb{N} \to \mathbb{R}$

Πώς γίνεται αυτή η στρογγυλοποίηση; Δες ένα παράδειγμα. Έστω ότι θέλουμε να στρογγυλοποιήσουμε το 1.347 σε $n = 2$ ψηφία μετά την υποδιαστολή (1.35). Κοίτα πώς γίνεται αυτό:

$1.347 \cdot 10^2 = 134.7$

myRound(134.7) = 135

$135 / 10^2 = 1.35$

```
// dRound -- στρογγυλοποιεί το όρισμα σε n ψηφία μετά την
//           υποδιαστολή. Χρησιμοποιεί τη myRound
double dRound( double x, int n )
{
   double tenTo( pow(10, n) );

   return myRound(x*tenTo)/tenTo;
} // dRound
```

Και είναι ολική η συνάρτησή μας; Ούτε λόγος! Κατ' αρχάς, αφού χρησιμοποιούμε τη *myRound()* θα πρέπει να έχουμε:

$$LONG\_MIN - 0.5 \leq x \cdot 10^n \leq LONG\_MAX + 0.5$$

Το n περιορίζεται από την ακρίβεια του `long int` (όχι του `double`) αλλά και από το μέγεθος του x: Αν ο `long int` έχει 10 ψηφία το πολύ και το ακέραιο μέρος του x έχει 8 ψηφία δεν μπορείς να ζητάς $n = 5$.

Θα μπορούσαμε να βάλουμε ελέγχους για τα παραπάνω, αλλά θα είναι αρκετά πολύπλοκοι· πιο πολύ θα μπερδευτείς παρά θα διδαχθείς.

Όπως βρήκαμε τη σχέση μεταξύ x, *myRound*(x) μπορείς να βρεις και τη σχέση μεταξύ x, n, *dRound*(x, n) (Άσκ. 7-19).

Πάντως η συνάρτηση είναι πολύ χρήσιμη. Πρόσεξε ότι δουλεύει και για $n \leq 0$. Για $n = 0$ δουλεύει όπως η *myRound* (στρογγυλοποιεί σε ακέραιο), για $n = -1$ στρογγυλοποιεί στο

[7] Πρόσεξε: θέλουμε την αλλαγμένη τιμή για να τη χρησιμοποιήσουμε και όχι απλώς για να την τυπώσουμε· η εκτύπωση μπορεί να γίνει με αυτά που μάθαμε στην §1.11.

ψηφίο των δεκάδων, για *n* = −2 στο ψηφίο των εκατοντάδων κ.ο.κ. Μπορείς να την ξαναγράψεις χρησιμοποιώντας την *lround()* αντι για τη *myRound()*.

Με τα παραδείγματα αυτής της παραγράφου, σου είπαμε τα περισσότερα από αυτά που πρέπει να ξέρεις για τις συναρτήσεις. Τα υπόλοιπα στην επόμενη παράγραφο.

7.7.1 *exit()* ή *assert()*

Μήπως αντί για **if** και *exit()* θα μπορούσαμε να χρησιμοποιούμε την *assert()* (§4.3); Βεβαίως! Για να δούμε τη διαφορά από αυτά που εμείς λέμε κάνουμε ένα πείραμα: Αλλάζουμε τη `main()` στο Παράδ. 1 της προηγούμενης παραγράφου:

```
int main()
{
   int m, n, comb;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   comb = factorial( m ) / ( factorial(n)*factorial(m-n) );
   cout << " Συνδυασμοί των "
        << m << " ανά " << n << " = " << comb << endl;
} // main
```

Να ένα παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: 2 5
 η factorial κλήθηκε με όρισμα -3
```

Αλλάζουμε τη *factorial()* ως εξής:

```
   unsigned long int factorial( int a )
   {
      unsigned long int fv;
      int k;

      assert( a >= 0 );

      fv = 1;
      for ( k = 1;  k <= a;  k=k+1 )  fv = fv*k;
      return fv;
   } // factorial
```

και βάζουμε στην αρχή την "`#include <cassert>`". Παράδειγμα εκτέλεσης:

```
 Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: 2 5
Assertion failed: a >= 0, file combA.cpp, line 12
```

Ποιο είναι πιο καλό;

- Για τον «άσχετο» χρήστη του προγράμματος και οι δύο αντιδράσεις είναι το ίδιο κακές· μπορεί να προκαλέσουν πανικό!
- Για τον προγραμματιστή που δοκιμάζει το πρόγραμμά του πριν το παραδώσει στον πελάτη/χρήστη η δική μας μορφή δίνει πιο πλήρη περιγραφή του προβλήματος.

Χρησιμοποίησε όποια σου αρέσει περισσότερο· de gustibus et coloribus non disputandum… Πάντως σκέψου το εξής: αν γράφεις ένα μεγάλο πρόγραμμα είναι δυνατόν να περιλάβεις μια συνάρτηση που θα σταματήσει την εκτέλεσή του επειδή ζήτησες να υπολογιστούν «οι συνδυασμοί των 2 ανά 5»; Εντάξει, αυτό δείχνει ότι κάτι δεν πάει καλά με το πρόγραμμα, αλλά δεν θα αποφασίσει η *factorial()* αν θα διακοπεί η εκτέλεσή του. Αργότερα θα μάθουμε και άλλον, πιο ευέλικτο, τρόπο για να διαχειριζόμαστε τέτοια προβλήματα.

7.8. Πώς (μετα)Γράφουμε μια Συνάρτηση

Τουλάχιστον όσοι ασχολούνται με την τεχνολογία και τις θετικές επιστήμες, έχουν συχνά να γράψουν πρόγραμμα που θα υπολογίζει τιμές κάποιας συνάρτησης. Στην εισαγωγή και στα παραδείγματα είδαμε πώς γίνεται κάτι τέτοιο. Τώρα θα δούμε τη διαδικασία μεταγραφής πιο συστηματικά.

Όπως είπαμε, τα πρώτα πράγματα που πρέπει να κάνεις είναι:

- να καθορίσεις το πεδίο ορισμού και το πεδίο τιμών της συνάρτησης,
- να σιγουρευτείς ότι έχεις το μηχανισμό που σου δίνει τις εικόνες από τα πρότυπα.

Ύστερα από αυτά, πρέπει να βρεις τους τύπους της C++ που αντιστοιχούν στα πεδία ορισμού και τιμών. Οι εύλογες αντιστοιχίες είναι $\mathbb{Z}$ και υποσύνολά του στον `int` (`long int`), $\mathbb{R}$ και υποσύνολά του (εκτός από τα υποσύνολα ακεραίων) στο `double`. Αν για παράδειγμα έχουμε:

$\| \|_2 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$

θα πάμε σε:

`EuMetro: double × double × double → double`

Τώρα πρόσεξε *μήπως η συνάρτηση που θα γράψεις δεν είναι ολική* άσχετα από το αν η αρχική είναι ή δεν είναι. Π.χ., στο παράδ. 3 της προηγούμενης παραγράφου είχαμε:
myRound: $\mathbb{R} \to \mathbb{Z}$

`myRound: double ↛ long int`

Και στο παράδ. 2:
ΜΚΔ: $\mathbb{N} \times \mathbb{N} \setminus \{ (0,0) \} \to \mathbb{N}$

`gcd: unsigned int × unsigned int ↛ unsigned int`

Και τι κάνουμε αν καταλήξουμε σε μερική συνάρτηση; Θα πρέπει να γράψουμε τη συνάρτησή μας ώστε να μπορεί να αντιμετωπίσει την περίπτωση που γίνεται κλήση με πραγματικές παραμέτρους εκτός πεδίου ορισμού. Όπως θα μάθουμε αργότερα η C++, παρέχει μηχανισμούς **διαχείρισης εξαιρέσεων** (exception handling) για τέτοιες περιπτώσεις. Προς το παρόν τι κάνουμε;

- Στα παραδείγματά μας χρησιμοποιήσαμε την *exit()* για να σταματήσουμε την εκτέλεση του προγράμματος αμέσως.
- Θα μπορούσαμε να βγάλουμε ένα μήνυμα "domain error" ή κάτι παρόμοιο και να βάλουμε στην συνάρτηση μια απίθανη τιμή (αν υπάρχει) που θα μπορεί να ανιχνευθεί μετά την κλήση.
- Θα μπορούσαμε να βγάλουμε ένα μήνυμα "domain error" ή κάτι παρόμοιο και να μην κάνουμε οτιδήποτε. Η τιμή που θα επιστρέψει η συνάρτηση στην παράσταση, από όπου έγινε η αναφορά, θα είναι τυχαία και θα οδηγήσει σε παράλογο (;) αποτέλεσμα.

Εμείς θα επιμείνουμε στην πρώτη λύση, αφού ένα πρόγραμμα που ζητάει τον υπολογισμό συνάρτησης με παραμέτρους εκτός πεδίου ορισμού έχει σίγουρα λάθος.

Ας δούμε τώρα άλλη μια συνηθισμένη περιπλοκή: η συνάρτησή μας, f, είναι περιοδική, με περίοδο T, δηλ. $f(x+kT) = f(x)$ για ακέραιες τιμές του k. Στην περίπτωση αυτή θα πρέπει να έχουμε ένα (πρωτεύον) διάστημα, ας πούμε το $[a, b)$, με μήκος μια περίοδο ($b = a + T$), όπου να έχουμε μηχανισμό που μας δίνει τις εικόνες από τα πρότυπα. Το πρώτο πράγμα που κάνουμε στην περίπτωση αυτή είναι αναγωγή της τιμής της παραμέτρου (x) σε κάποιο x_0, στο πρωτεύον διάστημα, για το οποίο $f(x_0) = f(x)$. Αυτό μπορεί να γίνει με τις εντολές:

```
   x0 = x;
   while ( x0 >= b )  x0 = x0 - T;
// (x0 < b) && (f(x0) == f(x))
   while ( x0 < a )   x0 = x0 + T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Πρόσεξε ότι η $f(x_0)$ == $f(x)$ είναι αναλλοίωτη και για τις δύο **while**, αφού οι διαδοχικές τιμές που μπορεί να πάρει η x_0 διαφέρουν από την τιμή της x κατά ακέραιο πλήθος περιόδων.

Πάντως η αναγωγή μπορεί να γίνει και πιο γρήγορα. Η C++ μας δίνει τη συνάρτηση (γνωστή από τα μαθηματικά, όπου τη γράφουμε συνήθως ως $\lfloor x \rfloor$) *floor*: `double` → `double`, τέτοια ώστε *floor*(x) να είναι ο μέγιστος *ακέραιος* που είναι μικρότερος ή το πολύ ίσος με x. Με τη βοήθειά της μπορούμε να βρούμε το x_0 πιο γρήγορα (Ασκ. 7-11):

```
   m = floor((x-a)/T);
   x0 = x - m*T;
// (a ≤ x0 < b) && (f(x0) == f(x))
```

Παρατηρήσεις: ▶

1. Να επαναλάβουμε αυτό που είπαμε και πιο πάνω: Αν έχεις παραμέτρους τύπου `double` (ή `float` ή `long double`) και θέλεις να ελέγξεις συγκεκριμένες τιμές, συγκρίσεις όπως "`x0 == v`" δεν έχουν νόημα. Θα πρέπει να ελέγχεις με κάτι σαν: "`fabs(x0 - v) <= eps`".
2. Αν η συνάρτησή σου είναι περιοδική και έχεις όρισμα πολύ μεγαλύτερο από την περίοδο τότε, κατά την αναγωγή, η x `- m*T` μπορεί να σου δώσει 0 (μηδέν).
3. Αν η συνάρτησή σου είναι περιοδική και το διάστημα ορισμού του μηχανισμού είναι της μορφής $(a, b]$ έχεις το εξής πρόβλημα: η μέθοδος αναγωγής που δώσαμε σου εγγυάται ότι $a \le x_0 < b$. Αν έχεις a == x_0 θα πρέπει να προσθέσεις μια περίοδο για να πας στο b[8]:

```
   m = floor( (x-a)/T );
   x0 = x - m*T;
// (a ≤ x0 < b) && (f(x0) == f(x))
   if (x0 <= a)  x0 = x0 + T;
// (a < x0 ≤ b) && (f(x0) == f(x)) ◀
```

Τα παραπάνω συνοψίζονται σε μια «συνταγή» που τη βλέπεις στο Πλ. 7.4. Στη συνέχεια βλέπεις ένα παράδειγμα εφαρμογής της συνταγής.

Παράδειγμα 🡲

Μια συνάρτηση ορίζεται μαθηματικά ως εξής:

$$v(x) = -\frac{1}{|x+2|} - \frac{1}{|x|} - \frac{1}{|x-2|} \quad \text{για } x \in [-1,1)$$

Η συνάρτηση είναι περιοδική με περίοδο $T = 2$. Να γίνει η υλοποίησή της στη C++.

1. Η συνάρτησή μας ορίζεται σε όλα τα σημεία του $[-1,1)$ εκτός από το 0 (μηδέν). Αφού η συνάρτηση είναι περιοδική με περίοδο 2, η συνάρτηση ορίζεται σε ολόκληρο το $\mathbb{R}$ εκτός από τα σημεία $2k$ όπου k ακέραιος. Άρα:

   v: $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \} \to \mathbb{R}$

δηλαδή: πεδίο ορισμού είναι το $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \}$ και πεδίο τιμών είναι το $\mathbb{R}$.

2. Ο μηχανισμός που μας δίνει τις εικόνες από τα πρότυπα δίνεται με ακρίβεια.
3. Στο σύνολο $\mathbb{R} \setminus \{ k \in \mathbb{Z} \bullet 2k \}$ όπως και στο $\mathbb{R}$ αντιστοιχεί ο `double`.
4. Επικεφαλίδα:

```
double v( double x )
```

5. Ενώ, όπως είδαμε παραπάνω, η v στα μαθηματικά είναι ολική, στη C++ θα έχουμε:

```
v: double ⇸ double
```

που φυσικά *δεν είναι* ολική.

6. Η αναγωγή της παραμέτρου στο διάστημα: $[-1,1)$ μπορεί να γίνει με τις εντολές:

```
   m = floor( (x-(-1))/T );
   x0 = x - m*T;
// (-1 ≤ x0 < 1) && (v(x0) == v(x))
```

[8] Στην περίπτωση αυτή μπορείς να κάνεις την αναγωγή με τη «δίδυμη» της *floor*, τη *ceil*. Δες την Άσκ. 7-12.

Πλαίσιο 7.4

Πώς (μετα)γράφουμε μια Συνάρτηση στη C++

Για να (μετα)γράψεις μια μαθηματική συνάρτηση σε C++ κάνε τα εξής:

1. Καθόρισε με ακρίβεια το πεδίο ορισμού (κυρίως) και το σύνολο τιμών.

2. Καθόρισε με ακρίβεια το μηχανισμό που μας δίνει τις εικόνες ($f(x)$) από τα πρότυπα (x).

3. Καθόρισε προσεκτικά τους τύπους της C++ που αντιστοιχούν στο πεδίο ορισμού και στο σύνολο τιμών. Για αριθμητικές συναρτήσεις οι αντιστοιχίες είναι:

- $\mathbb{N}$, $\mathbb{N}^*$ ή υποσύνολά τους στον `int` ή τον `long int`, για το πεδίο ορισμού, στον `unsigned int` ή τον `unsigned long int`, για το πεδίο τιμών.
- $\mathbb{Z}$, $\mathbb{Z}^*$ ή υποσύνολά τους στον `int` ή τον `long int`.
- $\mathbb{R}$ ή υποσύνολο του $\mathbb{R}$ στον `double` ή τον `long double` ή τον `float`,

4. Τώρα μπορείς να γράψεις την επικεφαλίδα της συνάρτησης και να δηλώσεις τους τύπους των ορισμάτων.

5. Μετά το βήμα 3, μπορείς να αποφανθείς κατά πόσον η συνάρτηση που θα γράψεις είναι *ολική* (ορίζεται σε ολόκληρο το σύνολο αφετηρίας) ή *μερική*.

6. Αν η συνάρτηση είναι περιοδική, γράψε τις εντολές που βρίσκουν τιμή (x_0) στο διάστημα ορισμού, ισοδύναμη με την αρχική (x), δηλαδή τέτοια ώστε: $x_0 \in$ διάστημα ορισμού και $f(x_0) = f(x)$.

7. Αν η συνάρτηση είναι μερική, γράψε τις εντολές που εξαιρούν τις τιμές του ορίσματος για τις οποίες δεν ορίζεται η συνάρτηση:

```
if ( x δεν ανήκει στο πεδίο ορισμού )
{
   cout << " η ... κλήθηκε με x = " << x << endl;
   exit( EXIT_FAILURE );
}
else
   Υπολόγισε την τιμή της συνάρτησης
```

8. Μετάφρασε σε C++ το μηχανισμό που καθόρισες στο βήμα 2.

7. Η συνάρτησή μας είναι μερική. Μετά την αναγωγή της x στη x_0 στο $[-1,1)$, όλα τα σημεία στα οποία δεν ορίζεται η V ανάγονται στο 0. Θα πρέπει να εξαιρέσουμε αυτήν την τιμή και μόνο:

```
if ( x0 == 0 )
{
   cout << " η v κλήθηκε με όρισμα: " << x << endl;
   exit( EXIT_FAILURE );
}
```

Πρόσεξε ότι στο μήνυμα δεν βάζουμε τη *x0* αλλά τη *x*. Ακόμη, πρόσεξε ότι η σύγκριση "`x0 == 0`" δεν έχει και πολύ νόημα. Πιο σωστό θα ήταν κάτι σαν "`fabs(x0) < eps`", όπου *eps* μια μικρή τιμή που εξαρτάται από τον τύπο `double` και το πρόβλημά σου.

8. Η μετάφραση του μηχανισμού είναι τετριμμένη:

```
fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
```

Να ολόκληρη η συνάρτηση:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double x0( x ), fv, m;
```

```
      m = floor( (x-a)/T );
      x0 = x - m*T;
   // (-1 ≤ x0 < 1) && (v(x0) == v(x))
      if ( x0 == 0 )
      {
         cerr << " η v κλήθηκε με όρισμα: " << x << endl;
         exit(EXIT_FAILURE);
      }
   // (-1 ≤ x0 < 1) && (v(x0) == v(x))
      fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
      return fv;
   } // v
```

7.9. * Οι Συναρτήσεις στις Αποδείξεις

Όπως είδαμε και πιο πριν, στις παραστάσεις που είχαμε *sqrt*(*x*) βάζαμε (κάπως αισιόδοξα) $\sqrt{x}$, για την οποία τα μαθηματικά μας δίνουν πολύ συγκεκριμένες προδιαγραφές. Ζητούσαμε όμως να έχουμε $x \geq 0$.

Κάπως έτσι θα χειριζόμαστε και τις δικές μας συναρτήσεις. Βάζουμε προϋπόθεση στις παραμέτρους και απαίτηση στην τιμή που επιστρέφει:

```
   Tf f(T1 p1, T2 p2,... Tn pn)
   {
      Tf fv;
       :
   // Pd(p1, p2,... pn)
        :
   // Qd(p1, p2,... pn, fv)
      return fv;
   } // f
```

Η προϋπόθεση στις παραμέτρους (ή τουλάχιστον ένα μέρος της) μας είναι ήδη γνωστή: *θα πρέπει να ανήκουν στο πεδίο ορισμού της συνάρτησης*. Όπως κάνουμε στα προγράμματά μας, αλλά όπως κάναμε και στα παραδείγματα, παραπάνω, θα πρέπει και εδώ να ελέγχουμε την προϋπόθεση.

Ύστερα από αυτό, κάθε φορά που έχουμε κάποια κλήση της *f*, ας πούμε *f*(*t*1, *t*2,... *tn*), θα πρέπει να σιγουρεύουμε ότι ισχύει η *Pd*(*t*1, *t*2,... *tn*). Για την παράσταση στην οποία υπάρχει η *f*(*t*1, *t*2,... *tn*) θα έχουμε *Qd*(*t*1, *t*2,... *tn*, *f*(*t*1, *t*2,... *tn*)).

Προσοχή σε ένα σημείο: οι τιμές των παραμέτρων μπορεί να αλλάζουν μέσα στη συνάρτηση· αλλά στην *Qd*(*p*1, *p*2,... *pn*, *fv*) θεωρούμε ότι οι *p*1, *p*2,... *pn* έχουν τις τιμές που είχαν αρχικά. Ένας τρόπος για να μην μπλεχτείς είναι να αντιγράψεις τις τιμές των παραμέτρων σε τοπικές μεταβλητές. Δες και το

Παράδειγμα

Θα αποδείξουμε ότι η συνάρτηση που γράψαμε για τον υπολογισμό του ΜΚΔ δύο φυσικών αριθμών (την παραθέτουμε στη συνέχεια, κάπως αλλαγμένη, για ευκολία) είναι σωστή. Θα στηριχθούμε στο θεώρημα της Αριθμοθεωρίας που λέει:

(Π1) *Αν οι* $|\alpha| + |\beta| > 0$ *(αν δηλαδή δεν είναι και οι δύο μηδέν) τότε ΜΚΔ*$(\alpha,\beta) = ΜΚΔ(\beta,\upsilon)$, *όπου υ το υπόλοιπο της διαίρεσης α:β*.

και στο προφανές:

(Π2) *Αν* $\alpha \neq 0$ *τότε ΜΚΔ*$(0,\alpha) = \alpha$.

```
   unsigned int gcd( int x, int y )
   { // x == x0 && y == y0
      unsigned int  b;

      if ( (x == 0 && y == 0) || x < 0 || y < 0 )
```

```
      {
         cout << " η gcd κλήθηκε με " << x << ", " << y << endl;
         exit( EXIT_FAILURE );
      }
   // (x == x0 && y == y0) &&
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
      {
         b = y;  y = x % y;  x = b;
      } // while
      return x;         // ΜΚΔ(x,0) = x
   } // gcd
```

Τι έχουμε να αποδείξουμε εδώ; Το εξής:

```
   // (x == x0 && y == y0) &&
   // (x != 0 || y != 0) && x >= 0 && y >= 0
      while ( y != 0 )  // I: ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
      {
         b = y;  y = x % y;  x = b;
      } // while
   // x == ΜΚΔ( x0, y0 )
```

δηλαδή:

- $Pd(x_0, y_0)$ είναι η: (x_0 >= 0) && (y_0 >= 0) && (!(x_0 == 0 && y_0 == 0))
- $Qd(x_0, y_0, x)$ είναι η: x == $ΜΚΔ(x_0, y_0)$

Και με την **if** δεν θα ασχοληθούμε; Η **if** υπάρχει για να ελέγξει την προϋπόθεση ή αλλιώς: αν οι παράμετροι έχουν τιμές μέσα στο πεδίο ορισμού. Αν η συνθήκη της **if** ισχύει η εκτέλεση της συνάρτησης (και του προγράμματος) δεν θα ολοκληρωθεί.

Ας έρθουμε τώρα στην απόδειξη: Αν η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) είναι αναλλοίωτη τότε μετά τη **while** θα έχουμε:

!(y != 0) && (ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0))

ή αλλιώς:

(y == 0) && (ΜΚΔ(x,y) == ΜΚΔ(x_0, y_0))

από την οποία παίρνουμε ΜΚΔ(x,0) == ΜΚΔ(x_0, y_0) και με βάση την (Π2): x == ΜΚΔ(x_0, y_0).

Φτάνει λοιπόν να αποδείξουμε ότι

1. η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) ισχύει πριν από τη **while** και
2. παραμένει αναλλοίωτη από τις επαναλαμβανόμενες εντολές:

```
      // (y != 0) && (ΜΚΔ(x,y) == ΜΚΔ(x0,y0))
         b = y;  y = x % y;  x = b;
      // ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
```

Η 1. είναι προφανής: η ΜΚΔ(x,y) == ΜΚΔ(x_0,y_0) συνάγεται από την x == x_0 && y == y_0. Και η (x_0 >= 0) && (y_0 >= 0) && (!(x_0 == 0 && y_0 == 0)) δεν μας χρειάζεται; Χρειάζονται για να ορίζεται ο ΜΚΔ(x_0,y_0).

Στη συνέχεια βλέπεις την απόδειξη της 2.:

```
      // ΜΚΔ(y, x % y) == ΜΚΔ(x0,y0)
         b = y;
      // ΜΚΔ(b, x % y) == ΜΚΔ(x0,y0)
         y = x % y;
      // ΜΚΔ(b,y) == ΜΚΔ(x0,y0)
         x = b;
      // ΜΚΔ(x,y) == ΜΚΔ(x0,y0)
```

Συνάγεται η ΜΚΔ(y, x % y) == ΜΚΔ(x_0,y_0) από την προϋπόθεση; Ναι, διότι η προϋπόθεση μας δίνει: ΜΚΔ(x, y) == ΜΚΔ(x_0,y_0) και από το (Π1) έχουμε: ΜΚΔ(x, y) == ΜΚΔ(y, x % y). Από αυτές τις δύο και τη μεταβατικότητα της ισότητας παίρνουμε: ΜΚΔ(y, x % y) == ΜΚΔ(x_0,y_0). Άρα η συνάρτησή μας είναι σωστή.

Αν λοιπόν γράψουμε σε μια παράσταση, στο πρόγραμμά μας, *gcd*(Π_1, Π_2), όπου Π_1 και Π_2 παραστάσεις με τιμές τύπου **unsigned int**, από τις οποίες η μια τουλάχιστον δεν είναι μηδέν, τότε, μετά την εκτέλεση της *gcd* θα έχουμε *gcd*(Π_1,Π_2) **==** ΜΚΔ(Π_1,Π_2).

7.10. Αναδρομή

Ήταν κάποιος που δεν ήξερε καμιά ιστορία
κι όλο έλεγε «ξέρω πολλές ιστορίες· μια απ' αυτές λέει πως
Ήταν κάποιος που δεν ήξερε καμιά ιστορία
κι όλο έλεγε "ξέρω πολλές ιστορίες· μια απ' αυτές λέει πως
Ήταν κάποιος... "»
Γ. Αγγελάκας, Υπέροχο Τίποτα

Στην C++ υπάρχει η δυνατότητα αναδρομικής διατύπωσης μιας συνάρτησης, όπως και στα μαθηματικά υπάρχει η δυνατότητα αναδρομικής διατύπωσης μερικών ορισμών. Ένας πολύ γνωστός αναδρομικός ορισμός είναι αυτός για το *n*!, που τον επαναλαμβάνουμε:

$$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n \geq 1 \end{cases}$$

Ο παραπάνω αναδρομικός ορισμός μπορεί να μεταφρασθεί εύκολα στη C++ σε μια **αναδρομική συνάρτηση** (recursive function) με τον ακόλουθο τρόπο:

```
// rFactorial -- Υπολογίζει το a! με αναδρομική κλήση
unsigned long int rFactorial( int a )
{
   unsigned int k, fv;

   if ( a < 0 )
   {
      cout << "η rFactorial κλήθηκε με όρισμα " << a << endl;
      exit( EXIT_FAILURE );
   }
   if ( a == 0 )  fv = 1;
            else  fv = a*rFactorial( a-1 );
   return fv;
} // rFactorial
```

Βλέπουμε λοιπόν ότι η συνάρτηση *rFactorial()* καλεί τον εαυτό της μέσα στο τμήμα των εντολών της. Ο παραπάνω τρόπος διατύπωσης της λειτουργίας μιας συνάρτησης, όπου μέσα στο σώμα του υποπρογράμματος εμφανίζεται αναφορά στην ίδια τη συνάρτηση λέγεται *αναδρομικός τρόπος* διατύπωσης του αλγορίθμου, ενώ η ίδια η συνάρτηση ονομάζεται *αναδρομική*. Συγκρίνοντας τον παραπάνω αναδρομικό τρόπο διατύπωσης του αλγορίθμου με τον ισοδύναμο επαναληπτικό τρόπο, που είδαμε σε προηγούμενες παραγράφους, μπορούμε να διαπιστώσουμε ότι ο αναδρομικός τρόπος είναι πιό απλός, πιό σύντομος και πλησιέστερος προς τον μαθηματικό ορισμό.

Συχνά όμως ο αναδρομικός τρόπος είναι λιγότερο αποδοτικός από τον επαναληπτικό και σε χρόνο και σε μνήμη. Γι' αυτό πρέπει να χρησιμοποιείται με προσοχή. Μη βιάζεσαι λοιπόν να χρησιμοποιήσεις αυτόν τον τρόπο παρ' όλη την κομψότητα διατύπωσης που προσφέρει. Άφησέ τον για αργότερα, που θα έχεις μεγαλύτερη ωριμότητα και θα έχεις μάθει μερικά πράγματα παραπάνω.

Προς το παρόν δες άλλο ένα παράδειγμα: η συνάρτηση για το ΜΚΔ γραμμένη με αναδρομή.

```
// rGcd -- Υπολογίζει τον Μέγιστο Κοινό Διαιρέτη των ορισμάτων
//         της με αναδρομική κλήση
unsigned int rGcd( int x, int y )
{
   unsigned int  fv;
```

```
    if ( x < 0 || y < 0 || (x == 0 && y == 0) )
    {
       cout << " η rGcd κλήθηκε με " << x << ',' << y << endl;
       exit( EXIT_FAILURE );
    }
    if ( y == 0 ) fv = x;                       // ΜΚΔ(x,0) = x
             else fv = rGcd(y, x % y);// ΜΚΔ(x,y) = ΜΚΔ(y,x % y)
    return fv;
} // rGcd
```

7.11. Ανακεφαλαίωση

Αν η C++ δεν έχει έτοιμη κάποια συνάρτηση (με τύπο) που χρειάζεσαι στο πρόγραμμά σου –πράγμα πολύ πιθανό– θα πρέπει να μπορείς να τη γράψεις.

Μια συνάρτηση

- Αρχίζει με μια επικεφαλίδα όπου καθορίζεται *ο τύπος της τιμής που επιστρέφει* (αντιστοιχεί στο πεδίο τιμών), *το όνομά της* και *οι τυπικές παράμετροί της με τους τύπους τους* (αντιστοιχούν στο σύνολο αφετηρίας).
- Ακολουθεί το σώμα της συνάρτησης, δηλαδή είναι ένα κομμάτι προγράμματος που περιγράφει το πώς υπολογίζεται η τιμή της συνάρτησης από τις τιμές των παραμέτρων.
- Μέσα στη συνάρτηση (επικεφαλίδα και σώμα) μπορεί να δηλώνονται *τοπικές* μεταβλητές, σταθερές κλπ που είναι γνωστές μόνον μέσα στη συνάρτηση και ζουν όσο διαρκεί η εκτέλεσή της.
- Στο σώμα της συνάρτησης υπάρχει μια τουλάχιστον εντολή **return** που τερματίζει την εκτέλεση της συνάρτησης και αντικαθιστά την κλήση της συνάρτησης με την τιμή που υπολόγισε.

Παρ' όλο που στο σώμα μιας συνάρτησης μπορεί να υπάρχουν πολλές εντολές **return**, *είναι προτιμότερο να υπάρχει μια μόνον, στο τέλος*.

Μια συνάρτηση *καλείται με το όνομά της* και τις πραγματικές παραμέτρους που θα δώσουν τις τιμές τους στις αντίστοιχες τυπικές.

Με το να «κρύβεις» κάποιους υπολογισμούς του προγράμματος μέσα σε συναρτήσεις αυξάνεις την ευκολία επαλήθευσης, διόρθωσης, τροποποίησης, και γενικώς διαχείρισής του.

Ασκήσεις

Α Ομάδα

7-1 Γράψε συνάρτηση, που θα επιστρέφει ως τιμή, την τιμή της μέγιστης των παραμέτρων του *a*, *b*, *c* και *d* (πραγματικοί αριθμοί).

7-2 α)Ένας **ημιανορθωτής τάσης** είναι ηλεκτρονική διάταξη με μια είσοδο και μια έξοδο. Αν στην είσοδο βάλουμε μια τάση θετική τότε στην έξοδο παίρνουμε την τάση εισόδου· αλλιώς (μη θετική τάση στην είσοδο) στην έξοδο παίρνουμε 0 (μηδέν). Γράψε συνάρτηση που να προσομοιώνει τη λειτουργία του ημιανορθωτή.
β) Ένας **ανορθωτής τάσης** (χωρίς εξομάλυνση) είναι ηλεκτρονική διάταξη με μια είσοδο και μια έξοδο. Αν στην είσοδο βάλουμε μια τάση θετική τότε στην έξοδο παίρνουμε την τάση εισόδου· αλλιώς (μη θετική τάση στην είσοδο) στην έξοδο παίρνουμε την αντίθετη της τάσης εισόδου. Γράψε συνάρτηση που να προσομοιώνει τη λειτουργία του ανορθωτή.

7-3 (Σύνθεση των ασκ. 2-9 και 5-5) Ας υποθέσουμε ότι ο μισθός ενός εργαζόμενου προσαυξάνεται κατά 2%, επί του βασικού μισθού, για κάθε χρόνο υπηρεσίας. Ακόμη, ο μισθός

ενός εργαζόμενου προσαυξάνεται κατά 30 € για κάθε παιδί, αν έχει μέχρι τρία (3) παιδιά. Αν έχει περισσότερα από 3 παιδιά η προσαύξηση είναι 50 € για κάθε παιδί.

Γράψε μια:

```
double salary( double base, int years, int noOfCldr )
```

που θα επιστρέφει το συνολικό μισθό. Σκέψου τις πιθανές κακοτοπιές, π.χ.: αρνητικές τιμές παραμέτρων, παράλογα μεγάλες τιμές παραμέτρων κλπ. Αν σου χρειαστούν άλλοι τύποι στοιχείων, όρισέ τους και άλλαξε την παραπάνω επικεφαλίδα.

7-4 Με βάση το πρόγραμμα που έγραψες για την Άσκ. 5-6, γράψε μια:

```
double resistors( char mode, double r1, double r2 )
```

που θα επιστρέφει την τιμή της αντίστασης που προκύπτει αν οι *r*1, *r*2 συνδεθούν εν σειρά (*mode* == 'S') ή παράλληλα (*mode* == 'P').

7-5 Με βάση το πρόγραμμα για τους λογαριασμούς της ΔΕΗ της §5.3, γράψε μια:

```
double elEnergyCost( double cons )
```

που θα επιστρέφει το κόστος της κατανάλωσης. Ξαναγράψε το πρόγραμμα που έγραψες για την άσκ. 6-3 με χρήση της συνάρτησης.

Β Ομάδα

7-6 Γράψε μια:

```
double dTrunc( double x, int n )
```

που θα μας επιστρέφει την *x*, αφού αποκόψει όλα τα ψηφία της μετά το *n*-οστό ψηφίο μετά την υποδιαστολή.

Υπόδ.: Δες πώς γράψαμε τη dRound θυμίσου και αυτά που λέγαμε στην §2.8.

7-7 Έχοντας λύσει την Άσκ. 4-15, δεν θα έχεις πρόβλημα να γράψεις τις παρακάτω συναρτήσεις που συμπληρώνουν αυτές που μας δίνει η C++ για το διαχωρισμό των χαρακτήρων:

```
// isGAlpha -- Επιστρέφει τιμή true αν ο ch είναι γράμμα
//             του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGAlpha( char ch )
// isGUpper -- Επιστρέφει τιμή true αν ο ch είναι κεφαλαίο
//             γράμμα του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGUpper( char ch )
// isGLower -- Επιστρέφει τιμή true αν ο ch είναι μικρό
//             γράμμα του Ελληνικού αλφαβήτου. Αλλιώς false
bool isGLower( char ch )
```

Και τώρα ξαναγράψε το πρόγραμμα που έγραψες για την Άσκ. 4-15, με χρήση αυτών των συναρτήσεων.

7-8 Γράψε τις παρακάτω συναρτήσεις:

```
// upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
//           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
char upCase( char ch )
// loCase -- Αν ο ch είναι κεφαλαίο Λατινικό γράμμα
//           επιστρέφει το αντίστοιχο μικρό, αλλιώς τον ch
char loCase( unsigned char ch )
```

καθώς και τις αντίστοιχες για τα ελληνικά: *gUpCase()*, *gLoCase()*.

Γ Ομάδα

7-9 Με βάση αυτά που είδες στο παράδ. 2 της §6.3 γράψε μια

```
double nrsqrt( double a )
```

που θα μας δίνει την τετραγωνική ρίζα του *a*. Γράψε πρόγραμμα που θα την δοκιμάζει και θα τη συγκρίνει με τη *sqrt* της C++.

7-10 Αν δούμε την κυβική ρίζα ενός πραγματικού, *a*, ως λύση της εξίσωσης $x^3 - a = 0$, μπορούμε να την προσεγγίσουμε με τη μέθοδο *Newton-Raphson*, $x_n = x_{n-1} - f'(x_{n-1})/f(x_{n-1})$, όπου, για την περίπτωσή μας:

$$f(x) = x^3 - a, \qquad f'(x) = 3x^2$$

Θα έχουμε λοιπόν:

$$x_n = \frac{1}{3}\left(2x_{n-1} + \frac{a}{x_{n-1}^2}\right)$$

Μπορούμε να ξεκινήσουμε με $x_0 = a$ και να υπολογίζουμε όρους της ακολουθίας μέχρι να πάρουμε $|x_n^3 - a| < \varepsilon_{\texttt{double}} a$. Πρόσεξε, ότι για `a == 0` έχουμε πρόβλημα, αλλά για την περίπτωση αυτή ξέρουμε τη λύση. Γράψε μια συνάρτηση

```
double cbrt( double a )
```

που υπολογίζει την κυβική ρίζα. Γράψε πρόγραμμα που θα την δοκιμάζει για διάφορες τιμές (*x*) συγκρίνοντας το *cbrt*(*x*) με το *pow*(*x*, 1/3.0) της C++.

***7-11** Αν η *v()* είναι περιοδική συνάρτηση με περίοδο *T* και $b-a = T$ τότε:

α) απόδειξε ότι:

```
// true
   x0 = x;
   while ( x0 >= b )  x0 = x0 - T; // αναλλοίωτη: v(x0) == v(x)
   while ( x0 < a )   x0 = x0 + T; // αναλλοίωτη: v(x0) == v(x)
// (a ≤ x0 < b) && (v(x0) == v(x))
```

β) απόδειξε ότι:

```
// true
   m = floor( (x-a)/T );
   x0 = x - m*T;
// (a ≤ x0 < b) && (v(x0) == v(x))
```

Για τη *floor()* ισχύουν τα εξής: $floor(x) \in \mathbb{Z}$ και $x - 1 < floor(x) \le x$.

***7-12** Αν η *v* είναι περιοδική συνάρτηση με περίοδο *T* και $b-a = T$ τότε απόδειξε ότι:

```
// true
   m = ceil( (x-a)/T );
   x0 = x - m*T;
// (a < x0 ≤ b) && (v(x0) == v(x))
```

Για τη *ceil()* ισχύουν τα εξής: $ceil(x) \in \mathbb{Z}$ και $x \le ceil(x) < x + 1$.

7-13 Γράψε συνάρτηση με μια παράμετρο *k*, που θα επιστρέφει ως τιμή τον *k*-οστό όρο της ακολουθίας, που καθορίζεται από τον τύπο:

$a_0 = 0,\ a_{k+1} = a_k k + k,$ για $k \in \mathbb{N}^*$

Δώσε μια μη αναδρομική και μια αναδρομική λύση στο πρόβλημα.

***7-14** *Η ακολουθία Fibonacci* ορίζεται ως εξής: $f_0 = 0$, $f_1 = 1$ και $f_k = f_{k-2} + f_{k-1}$ για $k > 1$. Γράψε μια *αναδρομική*:

```
int rFib( int k )
```

που θα επιστρέφει ως τιμή το f_k. Γράψε και μια μη αναδρομική συνάρτηση `int fib( int k )` που να κάνει την ίδια δουλειά.

7-15 Γράψε συνάρτηση

```
double mp( double x, int n )
```

που να υλοποιεί την παρακάτω συνάρτηση:

$$m_p(x,n) = \prod_{k=0}^{n-1} \frac{1}{x-k} = \frac{1}{x-0} \times \frac{1}{x-1} \times \ldots \times \frac{1}{x-(n-1)},$$ όπου *n* φυσικός ≥ 1.

***7-16** Γράψε αναδρομική λύση της προηγούμενης άσκησης.

7-17 Μια συνάρτηση ορίζεται μαθηματικά ως εξής:

$$q(x,n)=\begin{cases}-1, & -n<x\le -1\\ x, & -1<x\le 1\\ 1, & 1<x\le n\end{cases}$$

όπου n φυσικός > 1. Η συνάρτηση είναι περιοδική στη x με περίοδο $2n$. Να γραφεί συνάρτηση που την υλοποιεί σε C++.

***7-18** Απόδειξε ότι η *myRound()* (§7.7, παράδ. 3) είναι σωστή, δηλαδή:

```
// true
   if (x >= 0)  fv = (long int) (x + 0.5);
          else  fv = (long int) (x - 0.5);
// (x ≥ 0 && x - ½ < myRound(x) ≤ x + ½) ||
// (x < 0 && x - ½ ≤ myRound(x) < x + ½)
```

Υπόδ.: Έλυσες την άσκ. 3-6; Αν δεν την έλυσες δε γίνεται τίποτε…

***7-19** Απόδειξε ότι για τη *dRound()* (§7.7, παράδ. 4) έχουμε:

Αν $x \ge 0$ τότε

$x - 0.5\cdot10^{-n} < dRound(x, n) \le x + 0.5\cdot10^{-n}$ και

$dRound(x, n) - 0.5\cdot10^{-n} \le x < dRound(x, n) + 0.5\cdot10^{-n}$

Αν $x < 0$ τότε

$x - 0.5\cdot10^{-n} \le dRound(x, n) < x + 0.5\cdot10^{-n}$ και

$dRound(x, n) - 0.5\cdot10^{-n} < x \le dRound(x, n) + 0.5\cdot10^{-n}$

κεφάλαιο

8

Αρχεία I – Text

Ο στόχος μας σε αυτό το κεφάλαιο:
Να μάθεις να διαχειρίζεσαι τη βοηθητική μνήμη με εργαλείο το *σειριακό μορφοποιημένο αρχείο* ή *αρχείο-κείμενο*.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να γράφεις προγράμματα που θα φυλάγουν δεδομένα σε αρχείο ή αρχεία ώστε να τα χρησιμοποιήσεις αργότερα με άλλα προγράμματα. Με λίγη δουλειά παραπάνω θα μπορείς να επεξεργαστείς και αρχεία text που βγάζουν διάφορα «πακέτα» (έτοιμα προγράμματα) ή να ετοιμάσεις στοιχεία εισόδου για τέτοια «πακέτα».

Έννοιες κλειδιά:

- *σειριακό αρχείο*
- *μορφοποιημένο αρχείο ή αρχείο-κείμενο* (*text*)
- *μη-μορφοποιημένο ή δυαδικό* (*binary*) *αρχείο*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Στις συνηθισμένες εφαρμογές των ΗΥ, τα στοιχεία που έχει να επεξεργαστεί ένα πρόγραμμα είναι τόσο πολλά, που δεν είναι δυνατό να χωρέσουν στην κύρια μνήμη του ΗΥ, όλα μαζί. Ακόμη, τα στοιχεία αυτά δεν πρέπει να χάνονται όταν ο ΗΥ δεν λειτουργεί,

πράγμα που δεν συμβαίνει με τα στοιχεία που υπάρχουν στην κύρια μνήμη. Για το λόγο αυτόν, τα στοιχεία αποθηκεύονται σε βοηθητικές μονάδες μνήμης –δίσκους, ταινίες κλπ– και έχουν μια λογική οργάνωση ώστε να μπορούν να χρησιμοποιηθούν εύκολα από τα προγράμματα.

Ο πιο απλός τρόπος οργάνωσης είναι το **σειριακό αρχείο** (serial ή sequential file).

Μπορούμε να φανταστούμε το σειριακό αρχείο σαν μια πεπερασμένη ακολουθία από όμοιες –δηλ. του ίδιου τύπου– τιμές[1]. Ονομάζουμε **μήκος** του αρχείου τον αριθμό των στοιχείων που έχει. Το μήκος του *κενού αρχείου* είναι 0.

Σημείωση:▶

Εκτός από *μήκος* υπάρχει και το **μέγεθος** (size) του αρχείου που είναι ο αριθμός των ψηφιολέξεων που καταλαμβάνει. Για τα αρχεία που θα δούμε σε αυτό το κεφάλαιο –*αρχεία χαρακτήρων*– έχουμε *μήκος* == *μέγεθος*.◀

Παράδειγμα

Το:

```
< -7, -15, 0, 14, 33, -8, 16, 114, 375 >
```

είναι ένα αρχείο με στοιχεία τύπου `int` και μήκος 9.

Το:

```
<'m', 'a', 'i', 'n', '(', ')', '{', '}'>
```

είναι ένα αρχείο με στοιχεία τύπου `char` και μήκος 8 (είναι ένα πρόγραμμα C++).

Το:

```
<true, true, false, true, false, false, true>
```

είναι ένα αρχείο με στοιχεία τύπου `bool` και μήκος 7.

Πώς μπορούμε να «δούμε» το περιεχόμενο ενός αρχείου; Αν είναι ένα αρχείο σαν το δεύτερο του παραδείγματος ξέρεις ήδη την απάντηση: το παίρνουμε σε κάποιον κειμενογράφο και το βλέπουμε. Αυτό είναι ένα παράδειγμα **κειμένου** (text) ή **μορφοποιημένου αρχείου** (formatted file)[2]. Υπάρχει όμως και μια άλλη κατηγορία αρχείων: αυτά που τα στοιχεία τους είναι αντίγραφα των εσωτερικών παραστάσεων του υπολογιστή. Αν, ας πούμε, το πρώτο παράδειγμα είναι γραμμένο έτσι, μπορείς να το πάρεις στον κειμενογράφο αλλά δεν θα μπορείς να καταλάβεις το περιεχόμενό του. Αυτά τα αρχεία λέγονται **δυαδικά** (binary) ή **μη μορφοποιημένα** (unformatted).

Ένα αρχείο-κείμενο:

- Μπορείς να το μεταφέρεις εύκολα σε διαφορετικά υπολογιστικά περιβάλλοντα και διαφορετικούς υπολογιστές.
- Μπορείς να το διαχειριστείς με πρόγραμμα, αλλά και με έναν απλό κειμενογράφο.

Αλλά:

- Αν έχει αριθμητικά δεδομένα, η διαχείρισή τους επιβραδύνεται από το ότι
  - οι τιμές της εσωτερικής παράστασης θα πρέπει να μεταφράζονται σε χαρακτήρες, όταν γράφονται στο αρχείο
  - οι χαρακτήρες (ψηφία) που διαβάζονται από το αρχείο θα πρέπει να μεταφράζονται στην εσωτερική παράσταση.

Ένα δυαδικό αρχείο:

---

[1] Θα χρησιμοποιήσουμε τα σύμβολα "<", ">", "," για να παραστήσουμε ένα αρχείο στο χαρτί και μόνο. Φυσικά: α) δεν τα χρησιμοποιεί ο ΗΥ όταν αποθηκεύει το αρχείο β) δεν θα πρέπει να τα μπερδεύεις με τα ειδικά σύμβολα της C++. Με αυτόν το συμβολισμό, το **κενό** αρχείο –δηλ. αυτό που δεν έχει στοιχεία– παριστάνεται ως: <>.

[2] Καμιά φορά θα το δεις και ως *αρχείο ascii*.

- Παίρνει τα στοιχεία του από τη μνήμη (ή τα δίνει σε αυτή) χωρίς καμμιά μετατροπή και έτσι η επεξεργασία του είναι πιο γρήγορη.

Αλλά:

- Δεν μπορείς να το μεταφέρεις εύκολα.
- Η διαχείρισή του απαιτεί ειδικό πρόγραμμα.

Για τους παραπάνω λόγους τα διάφορα «πακέτα», συνηθέστατα, μπορούν να γράφουν και να διαβάζουν αρχεία-κείμενα, ώστε να μπορούν να ανταλλάσσουν στοιχεία από τη μια εγκατάστασή τους στην άλλη. Άλλα αρχεία που δημιουργούν, για να διαβαστούν από άλλο «πακέτο»

- μπορεί να είναι αρχεία-κείμενα,
- μπορεί να είναι δυαδικά αρχεία,
- συχνά χρειάζονται ειδικά προγράμματα (φίλτρα) είτε είναι δυαδικά είτε είναι κείμενα.

Στο κεφάλαιο αυτό θα ασχοληθούμε αποκλειστικά με αρχεία-κείμενα.

8.1 Σειριακά Αρχεία στην C++

Σε κάθε Λειτουργικό Σύστημα (ΛΣ) υπάρχει ένα κομμάτι που λέγεται **διαχειριστής αρχείων** (file manager) ή **σύστημα για τα αρχεία** (file system). Ο διαχειριστής αρχείων κρατάει **καταλόγους** (directories) όπου για κάθε αρχείο υπάρχουν στοιχεία όπως: η θέση του πάνω στο μέσο αποθήκευσης, το μέγεθος του αρχείου, πότε δημιουργήθηκε το αρχείο, πότε ενημερώθηκε τελευταία φορά κλπ. Μέσα στον κατάλογο, το αρχείο έχει κάποιο *όνομα* που το έχει ορίσει αυτός που το δημιούργησε.

Παράδειγμα

Παρακάτω, βλέπεις την εικόνα καταλόγου που μας δίνει ο διαχειριστής αρχείων του ΛΣ Windows:

```
Volume in drive C has no label
 Serial Number of Volume is  50DA-631A

 Directory of C:\0809bea\t02

14/04/2009  03:10 AM    <DIR>          .
14/04/2009  03:10 AM    <DIR>          ..
13/04/2009  09:59 AM             7,875 AircraftType.cpp
13/04/2009  09:57 AM             3,319 AircraftType.h
24/03/2009  02:52 PM               250 aircraftTypes.txt
13/04/2009  09:05 AM            14,016 ask01a.cpp
13/04/2009  03:28 AM           522,116 ask01a.exe
13/04/2009  09:41 AM            14,538 ask01b.cpp
13/04/2009  09:41 AM           520,081 ask01b.exe
24/03/2009  03:36 PM             7,636 flightHours.txt
17/11/2008  10:08 PM             7,090 MyLib.cpp
13/04/2009  10:00 AM             3,856 nAircraftTypes.txt
13/04/2009  09:34 AM             3,856 nPilots.txt
13/04/2009  10:16 AM           233,472 ooPr_t02.doc
13/04/2009  10:18 AM               604 ooPr_t02.log
13/04/2009  10:20 AM           362,918 ooPr_t02.pdf
13/04/2009  10:17 AM         1,146,318 ooPr_t02.prn
13/04/2009  10:21 AM           371,671 ooPr_t02.ZIP
13/04/2009  06:50 AM           367,616 ooPr_t02b.doc
13/04/2009  03:23 AM             7,126 Pilot.cpp
13/04/2009  03:19 AM             3,441 Pilot.h
24/03/2009  04:03 PM             1,583 pilots.txt
09/04/2009  02:12 PM           677,110 t02.rar
              20 file(s)     4,276,492 bytes
               2 Directories 50,009,026,560 bytes free
```

Κάθε καταχώριση (γραμμή) του καταλόγου έχει: όνομα αρχείου, μέγεθος (σε ψηφιολέξεις), την ημερομηνία και την ώρα που ενημερώθηκε για τελευταία φορά.

Για να χειριστούμε ένα αρχείο μέσα από το πρόγραμμά μας θα πρέπει:

1. το ΛΣ να μας επιτρέψει πρόσβαση στο αρχείο αυτό,
2. να δημιουργήσουμε ένα κανάλι ή, όπως το λέει η C++, ένα **ρεύμα** (stream) από το αρχείο προς το πρόγραμμά μας (όταν διαβάζουμε) ή από το πρόγραμμά μας προς το αρχείο (όταν γράφουμε).
3. να βρούμε την αρχή του, από όπου θα αρχίσει η ανάγνωση ή η εγγραφή στο αρχείο.

Λέμε ότι πρέπει να **ανοίξουμε** (open) το ρεύμα του αρχείου.

Εδώ θα γνωρίσουμε τρεις τύπους (κλάσεις) ρευμάτων της C++:

- τον `ifstream` για ρεύματα από το αρχείο προς το πρόγραμμά μας (**i**nput **f**ile **stream**),
- τον `ofstream` για ρεύματα από το πρόγραμμά μας προς το αρχείο (**o**utput **f**ile **stream**),
- τον `fstream` για ρεύματα που μπορεί να είναι και διπλής κατεύθυνσης.

Για να χρησιμοποιήσεις αυτούς τους τύπους θα πρέπει να περιλάβεις στο πρόγραμμά σου το αρχείο fstream (`#include <fstream>`).

Πριν προχωρήσουμε πιο κάτω θα κάνουμε μια παρένθεση για να πούμε δύο λόγια για τις κλάσεις. Μια **κλάση** (class) είναι ένας τύπος στοιχείων. Οι μεταβλητές και οι τιμές μιας κλάσης ονομάζονται και **αντικείμενα** (objects). Αυτά έχουν **περικλεισμένες** (encapsulated) **μεθόδους** (methods) ή **συναρτήσεις-μέλη** (member functions), για τη διαχείριση των στοιχείων τους. Π.χ. οι τρεις κλάσεις που αναφέραμε πιο πάνω έχουν μια μέθοδο που ονομάζεται *open* για άνοιγμα ρεύματος. Αν λοιπόν έχουμε δηλώσει:

```
ifstream a;
```

ανοίγουμε το ρεύμα *a* από το αρχείο text1.txt προς το πρόγραμμά μας με την:

```
a.open( "text1.txt" );
```

Ένα άλλο χαρακτηριστικό των κλάσεων είναι η **κληρονομιά** (inheritance) που μπορεί να αφήσει μια κλάση σε μια άλλη κλάση-παιδί της, π.χ. διάφορες μέθοδοι, τελεστές κλπ. Όπως θα δεις στη συνέχεια, μετά από τις παραπάνω δηλώσεις θα μπορούμε να διαβάζουμε από το αρχείο text1.txt ως εξής:

```
a >> x;
```

όπως ακριβώς διαβάζουμε από το πληκτρολόγιο με τη:

```
cin >> x;
```

παρ' όλο που το *cin* είναι ρεύμα κλάσης *istream*. Η *ifstream* είναι απόγονος της *istream* και –μέσω αυτής– μιας άλλης κλάσης, της *ios_base*, από την οποίαν κληρονόμησαν και οι δύο τον τελεστή "`>>`".

Αυτά τα ολίγα για τις κλάσεις προς το παρόν· θα τις ξαναδούμε αργότερα.

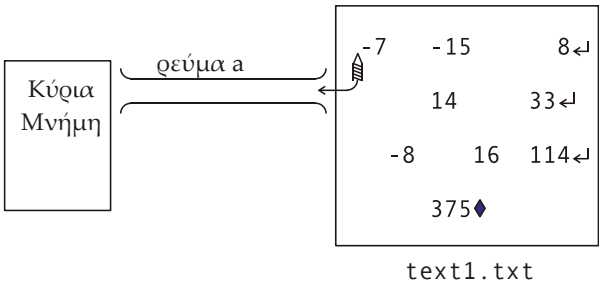
8.2 Αρχεία και Ρεύματα – Μια Εικόνα

Θα δώσουμε τώρα μια εικόνα για το (σειριακό) αρχείο-κείμενο και με αυτή θα δούμε τη διαχείρισή του.

Ας πούμε λοιπόν ότι το αρχείο είναι «γραμμένο» πάνω σε μια μαγνητοταινία (Σχ. 8-1).

Το αρχείο έχει μια αρχή και ένα τέλος. Το τέλος, εδώ στην εικόνα, το σημειώνουμε με το "♦". Αυτό δεν σημαίνει ότι ο ΗΥ σημειώνει το τέλος αρχείου με τον ίδιο τρόπο, αλλά, όπως θα δεις παρακάτω, αυτό δεν μας ενδιαφέρει. Το αρχείο-κείμενο είναι χωρισμένο σε γραμμές· στο σχήμα μας σημειώνουμε το τέλος γραμμής με το "↵".

Μια μαγνητική κεφαλή (παρόμοια με αυτήν του μαγνητοφώνου) διαβάζει το (γράφει στο) αρχείο. Λέγεται **κεφαλή ανάγνωσης/εγγραφής** (read/write head).



Σχ. 8-1 Παράσταση αρχείου-κειμένου (text1.txt) και ρεύματος *a* για την ανάγνωσή του. Παριστάνουμε το αρχείο όπως περίπου θα το δεις σε έναν κειμενογράφο. Τα "↵" και "♦" σημειώνουν το τέλος γραμμής και αρχείου αντιστοίχως. Το "⇧" παριστάνει την κεφαλή ανάγνωσης/εγγραφής

Η παράσταση του Σχ. 8-1 είναι απλουστευμένη αλλά θα μας βοηθήσει να καταλάβουμε τη διαχείριση του αρχείου-κειμένου. Βέβαια κρύβουμε λεπτομέρειες και περιπλοκές. Μια από αυτές είναι η ύπαρξη και ο ρόλος του *ενταμιευτή* που θα μας χρειαστεί σε κάποιο σημείο στη συνέχεια. Παρακάτω λέμε μερικά πράγματα γι' αυτόν.

Με τον όρο **ενταμιευτής** αποδίδουμε την αγγλική λέξη «buffer». Στα αγγλικά σημαίνει: αυτός που απαλλάσσει κάποιον από μια ενοχλητική δουλειά. Και αυτό ακριβώς κάνει ο ενταμιευτής: απαλλάσσει την ΚΜΕ (Κεντρική Μονάδα Επεξεργασίας, CPU) από τη συνεχή απασχόληση με τις περιφερειακές μονάδες –στην περίπτωσή μας μονάδες βοηθητικής μνήμης. Ο έλεγχος των μονάδων αυτών είναι αρκετά χρονοβόρος. Γι' αυτό, το ρεύμα περιλαμβάνει μια περιοχή μνήμης αρκετά μεγάλη (συνηθισμένες τιμές 1 *kB*, 2 *kB*). Όταν λοιπόν ζητήσουμε ανάγνωση μιας τιμής από το αρχείο δεν φέρνει μόνον αυτό που ζητήσαμε (π.χ. `'-'` και `'7'`) αλλά, π.χ., 1024 χαρακτήρες. Έτσι, οι επόμενες αναγνώσεις δεν απαιτούν πρόσβαση στο δίσκο αλλά μεταφορές μέσα στην κύρια μνήμη. Παρομοίως, όταν γράφουμε στο αρχείο, στην πραγματικότητα τα στοιχεία γράφονται στον ενταμιευτή. Όταν αυτός «γεμίσει», το περιεχόμενό του αντιγράφεται στο αρχείο.

8.3 Πώς Διαβάζουμε Ένα Αρχείο

Ας πούμε λοιπόν ότι στο δίσκο μας υπάρχει το αρχείο-κείμενο text1.txt. Τώρα θα δούμε πώς θα γράψουμε ένα πρόγραμμα που θα διαβάζει αυτό το αρχείο.

Όπως είπαμε και παραπάνω, μας χρειάζεται ένα ρεύμα κλάσης *ifstream*.Θα πρέπει λοιπόν να δηλώσουμε:

```
ifstream  a;
```

και να το ανοίξουμε ώστε να συνδέσει το αρχείο με το πρόγραμμά μας:

```
a.open( "text1.txt" );
```

Το στιγμιότυπο που βλέπεις στο Σχ. 8-1 δείχνει την κατάσταση που βρισκόμαστε μετά την *open()*. Η κεφαλή ανάγνωσης/εγγραφής βρίσκεται στην αρχή του αρχείου.

Ας πούμε ότι έχουμε δηλώσει ακόμη:

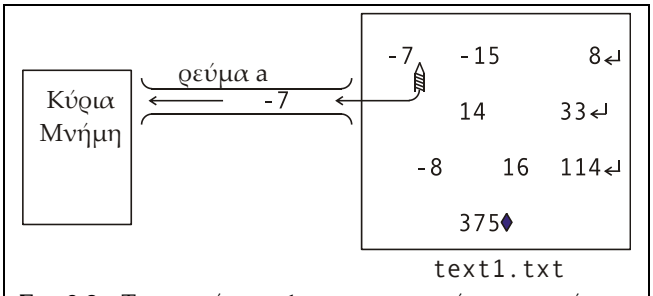
```
short int x, y;
```

και θέλουμε να διαβάσουμε την πρώτη τιμή του αρχείου και να την αποθηκεύσουμε στη *x*. Δίνουμε την εντολή:

```
a >> x;
```

Με την εκτέλεση αυτής της εντολής γίνονται τα εξής:

1. Η κεφαλή περνάει το αρχικό κενό και φτάνει στο `'-'`.
2. Η κεφαλή διαβάζει τους χαρακτήρες `'-'` και `'7'` και σταματάει όταν βρει το κενό.
3. Οι χαρακτήρες διαβιβάζονται στην κύρια μνήμη.



Σχ. 8-2 Το αρχείο text1.txt και το ρεύμα *a* μετά την ανάγνωση της πρώτης τιμής.

4. Οι δύο χαρακτήρες μεταφράζονται στην εσωτερική παράσταση του αριθμού "**-7**" που αποθηκεύεται στη θέση της μνήμης *x*.

Στο Σχ. 8-2 δείχνουμε την κατάσταση μετά την ανάγνωση της πρώτης τιμής.

Στη συνέχεια μπορούμε να επεξεργαστούμε αυτήν την τιμή· μπορούμε να δώσουμε, για παράδειγμα:

```
y = x*x;
cout << "   x = " << x << "   x^2 = " << y << endl;
```

Παίρνουμε αποτέλεσμα:

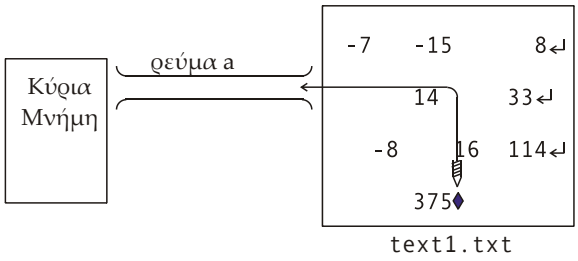
```
x = -7   x^2 = 49
```

Αν τώρα θέλουμε να επεξεργαστούμε τη δεύτερη τιμή, ζητούμε και πάλι εκτέλεση της εντολής:

```
a >> x;
```

Η κεφαλή θα ξεπεράσει τα κενά και θα διαβάσει και θα στείλει μέσω του *a* τους χαρακτήρες '**-**', '**1**', '**5**'. Με τον ίδιο τρόπο διαβάζεται και το 8. Αν ξαναζητήσουμε ανάγνωση τιμής, η κεφαλή θα ξεπεράσει το σημάδι τέλους γραμμής και τα κενά για να διαβάσει τους '**1**', '**4**'.

Συνεχίζοντας με αυτόν τον τρόπο φτάνουμε κάποτε στην κατάσταση που βλέπεις στο Σχ. 8-3. Αν ζητήσεις και πάλι εκτέλεση της `a >> x` αυτή θα αποτύχει και η τιμή της *x* δεν θα αλλάξει. Κάπου μέσα στον υπολογιστή σου «θα σηκωθεί μια σημαία» –κάποιο δυαδικό ψηφίο θα αλλάξει από 0 σε 1– που θα δείχνει ότι φτάσαμε στο τέλος του αρχείου. Μπορούμε να δούμε αυτή τη σημαία; Ναι! Η *ifstream* έχει μια μέθοδο, που ονομάζεται *eof* (χωρίς παραμέτρους) από τις αγγλικές λέξεις **e**nd **o**f **f**ile (τέλος αρχείου). Αν ελέγξεις την τιμή της `a.eof()`



Σχ. 8-3 Τι θα γίνει αν στην κατάσταση αυτή εκτελεστεί η εντολή `a >> x`; Η τιμή της *x* δεν θα αλλάξει. Αν ελέγξουμε την τιμή της `a.eof()` θα είναι μη μηδενική (`true`).

- μετά από μια προσπάθεια ανάγνωσης που απέτυχε διότι φτάσαμε στο τέλος του αρχείου αυτή θα είναι μη μηδενική (ως συνθήκη μεταφράζεται σε `true`)
- μετά από μια επιτυχή προσπάθεια ανάγνωσης αυτή θα είναι μηδενική (που ως συνθήκη θεωρείται `false`).

Φυσικά, αν η ανάγνωση δεν γίνει δεν έχει νόημα να επεξεργαστούμε την τιμή της *x*.

Το παρακάτω απλό πρόγραμμα κάνει μια στοιχειώδη επεξεργασία στο αρχείο που χρησιμοποιήσαμε στα παραδείγματά μας μέχρι τώρα. Υπολογίζει το τετράγωνο του κάθε στοιχείου. Επειδή το πρόγραμμα εκτελείται σε μια υλοποίηση της C++ όπου *SHRT_MAX* = 32767, αν η απόλυτη τιμή κάποιου στοιχείου είναι μεγαλύτερη από 181 (που είναι η ακέραια προσέγγιση της $\sqrt{SHRT\_MAX}$) τότε υπολογίζει το τετράγωνο του `x % 181`.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream  a;
   short int x, xr, y;

   a.open( "text1.txt" );
   a >> x;
   while ( !a.eof() )
```

```
   {
      if ( abs(x) > 181 )  xr = x % 181;
                     else  xr = x;
      y = xr*xr;
      cout << "   x =";  cout.width(4);  cout << x;
      cout << "    xr =";  cout.width(4);  cout << xr;
      cout << "    xr^2 =";  cout.width(5);  cout << y << endl;
      a >> x;
   } // while
   a.close();
} // main
```

Αυτό το πρόγραμμα δίνει τελικώς:

```
x =  -7    xr =  -7    xr^2 =   49
x = -15    xr = -15    xr^2 =  225
x =   8    xr =   8    xr^2 =   64
x =  14    xr =  14    xr^2 =  196
x =  33    xr =  33    xr^2 = 1089
x =  -8    xr =  -8    xr^2 =   64
x =  16    xr =  16    xr^2 =  256
x = 114    xr = 114    xr^2 =12996
x = 375    xr =  13    xr^2 =  169
```

Πρόσεξε την τελευταία εντολή:

```
   a.close();
```

Με την εκτέλεσή της το ρεύμα *a* αποσυνδέεται από το αρχείο text1.txt. Αν θέλουμε μπορούμε να το χρησιμοποιήσουμε για να επεξεργαστούμε άλλο αρχείο.

Το πρόγραμμα αυτό είναι παράδειγμα επεξεργασίας σειριακού αρχείου. Στο Πλ. 8.1 βλέπεις το γενικό σχήμα αυτής της επεξεργασίας. *Ξεκινώντας με a.open(..) και προχωρώντας μέχρις ότου η* `a.eof()` *δώσει μη μηδενική τιμή είναι σίγουρο ότι θα διαβάσουμε όλες τις τιμές του αρχείου*. Αυτό δεν σημαίνει ότι πρέπει να τις επεξεργαστείς όλες υποχρεωτικώς· μπορείς να κάνεις επιλεκτική επεξεργασία. Όπως βλέπεις, το σχήμα επεξεργασίας σειριακού αρχείου είναι όμοιο με το σχήμα επανάληψης με φρουρό.

Παρατηρήσεις ▶

1. Ας δούμε τώρα το εξής πρόβλημα: θέλουμε να διαβάσουμε την *πέμπτη* τιμή του αρχείου (το 33). Πρέπει να κάνουμε τα εξής:

i. Να φέρουμε το αρχείο στην αρχή του (Σχ. 8-1).

ii. Να διαβάσουμε και να αγνοήσουμε τις τέσσερις πρώτες τιμές του αρχείου.

iii. Να διαβάσουμε την πέμπτη τιμή.

Στις διαδικασίες αυτές φαίνεται πολύ καλά ένα βασικό χαρακτηριστικό των σειριακών αρχείων:

- ***Για να διαβάσουμε (ή να γράψουμε) το ν-οστό στοιχείο ενός σειριακού αρχείου πρέπει να περάσουμε κατ' ανάγκη τις ν – 1 προηγούμενες τιμές.***

Για να γίνει όμως αυτό,

- ***Η επεξεργασία κάθε σειριακού αρχείου ξεκινάει πάντα από την αρχή του.***

Πλαίσιο 8.1

Επεξεργασία Σειριακού Αρχείου

```
ifstream a;
   :
a.open( όνομα αρχείου );
a >> x;
while (!a.eof())
{
   Εντολές επεξεργασίας της x
   a >> x;
} // while
a.close();
```

2. Κάτι άλλο που φαίνεται στο πρόγραμμά μας είναι το εξής: μετά από την εκτέλεση της "`a >> x`" υπολογίζεται η "`!a.eof()`" και μόνο αν βρεθεί `true`, αν η ανάγνωση έγινε κανονικώς, επεξεργαζόμαστε την τιμή της *x*. Και αυτή είναι μια γενική αρχή:

- ***Πριν επεξεργαστούμε μια τιμή που (υποτίθεται ότι) διαβάσαμε από ένα αρχείο ελέγχουμε την eof για να δούμε αν η ανάγνωση έγινε κανονικά.***

3. Όταν προσπαθείς να πάρεις μια τιμή από ένα αρχείο-κείμενο με τον "`>>`", εκτός από τα κενά (διαστήματα) και τα σημάδια τέλους γραμμής, «τρώγονται» και οι στηλοθέτες (tabs). Όλα αυτά η C++ τα ονομάζει **λευκά διαστήματα** (white spaces).

4. Αν το αρχείο που επεξεργάζεσαι δεν είναι στον ίδιο υποκατάλογο με το πρόγραμμά σου θα πρέπει να περιλάβεις στο όνομα και τη **διαδρομή** (path). Αν δουλεύεις σε περιβάλλον Windows η διαδρομή θα περιλαμβάνει τον χαρακτήρα `'\'` που, όπως έχουμε πει, πρέπει να γράφεται δύο φορές. Αν έχεις π.χ.:

```
c:\students\datfl\text1.txt
```

θα πρέπει να δώσεις:

```
a.open( "c:\\students\\datfl\\text1.txt" );
```

4. Έχεις τη δυνατότητα να ανοίξεις το αρχείο όταν το δηλώνεις, π.χ.:

```
ifstream a( "text1.txt" );
```

Στην περίπτωση αυτή, δεν θα δώσεις `a.open("text1.txt")` στη συνέχεια.◀

8.3.1 *cin.eof()*

Μπορούμε να ελέγχουμε για «τέλος αρχείου» όταν παίρνουμε δεδομένα από το πληκτρολόγιο μέσω του *cin*; Ναι! Ο χρήστης «πληκτρολογεί το eof» δίνοντας

- `char(3)` (**E**nd of **Te**x**t**, ETX) με πληκτρολόγηση <ctrl-C> ή
- `char(4)` (**E**nd **O**f **T**ransmission, EOT) με πληκτρολόγηση <ctrl-D> ή
- `char(26)` με πληκτρολόγηση <ctrl-Z> (συνήθως σε περιβάλλον Windows).

Δυστυχώς η πληκτρολόγηση εξαρτάται από το ΛΣ και τον μεταγλωττιστή. Για παράδειγμα σε εφαρμογές «κονσόλας» στα Windows ο ETX δεν μπορεί να χρησιμοποιηθεί διότι του «έχει ανατεθεί» άλλος ρόλος (διακόπτει την εκτέλεση.) Χρησιμοποιείται, από την εποχή του MS-DOS, ο <ctrl-Z>. Θα το δεις να δουλεύει αν χρησιμοποιείς gcc (π.χ. Dev C++) ή BC++ 5.02.

8.3.2 Για να Ξαναχρησιμοποιήσεις το Ρεύμα

Ας πούμε ότι μετά την

```
ifstream a( "text1.txt" );
```

διάβασες το αρχείο `text1.txt` μέχρι το τέλος του και σταμάτησες την ανάγνωση μόλις πήρες *a.eof()*. Σε ποια κατάσταση είναι το ρεύμα; Σε «κακή κατάσταση»[3] ή, με άλλα λόγια, έχει ανασταλεί η λειτουργία του: δεν μπορεί να δεχθεί οποιαδήποτε εντολή· σε ορισμένες περιπτώσεις (μεταγλωττιστές) δεν εκτελεί ούτε την *close()*! Αν το ρεύμα έχει ανοιχτεί για διάβασμα δεν πάθαμε ζημιά εκτός από μια περίπτωση: Αν θέλουμε να ξαναχρησιμοποιήσουμε το ρεύμα για να δουλέψουμε με το ίδιο αρχείο ή με κάποιο άλλο.

Για να μπορούμε να ξαναχρησιμοποιήσουμε το ρεύμα θα πρέπει το *ανατάξουμε* δίνοντας την εντολή:

```
a.clear();
```

Ας πούμε ότι μετά την *a.eof()* θέλουμε να χρησιμοποιήσουμε το *a* για να διαβάσουμε ένα άλλο αρχείο, ας πούμε το `text2.txt`. Στην περίπτωση αυτή οι εντολές:

```
a.close();
a.open( "text2.txt" );
```

δεν αρκούν. Το σωστό είναι να γράψουμε:

```
a.clear();
a.close();
a.open( "text2.txt" );
```

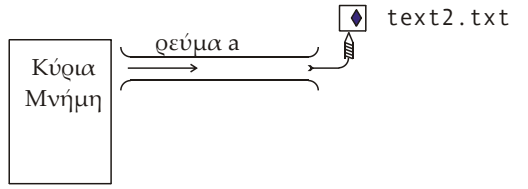
8.4 Πώς Γράφουμε Ένα Αρχείο

Τώρα θα δούμε πώς δημιουργούμε ένα αρχείο-κείμενο. Όπως καταλαβαίνεις τώρα θα πρέπει να δημιουργήσουμε ένα ρεύμα *από το πρόγραμμά μας προς το αρχείο*. Θα πρέπει λοιπόν να δηλώσουμε:

```
ofstream a;
```

Και η κλάση *ofstream* έχει μέθοδο *open()* για το άνοιγμα του αρχείου:

```
a.open( "text2.txt" );
```



Σχ. 8-4 Παράσταση αρχείου-κειμένου (text2.txt) και ρεύματος *a* για τη δημιουργία του. Το στιγμιότυπο δείχνει την κατάσταση ακριβώς μετά την εκτέλεση της `a.open('text2.txt')`. Το αρχείο είναι άδειο.

Εδώ όμως έχουμε διαφορά από την *open()* της *ifstream*: *Αν το αρχείο text2.txt υπήρχε, με την εκτέλεση της open() χάθηκε το περιεχόμενό του!* Το αρχείο καθαρίζεται και είναι έτοιμο για γράψιμο. Στο Σχ. 8-4 βλέπεις σχηματικά την κατάσταση μετά την εκτέλεση της *open()*.

Και τώρα θέλω να γράψω στο αρχείο την τιμή "`23`". Πώς θα γίνει αυτό; Μάλλον το έχεις μαντέψει: όπως η *ifstream* έχει τον "`>>`", που τον ξέρουμε από το *cin*, και η *ofstream* έχει τον, γνωστό μας από τη *cout*, "`<<`". Αν λοιπόν δώσουμε:

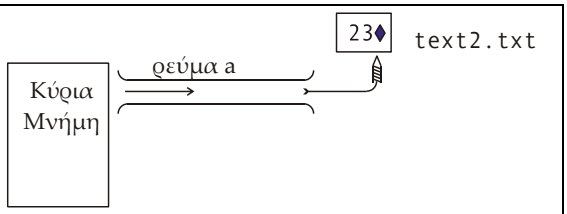
```
a << 23;
```

θα γίνουν αυτά που προσπαθούμε να δείξουμε στο Σχ. 8-5:

1. Η ακέραιη σταθερά «μεταφράζεται» στους χαρακτήρες `'2'`, `'3'`.
2. Οι χαρακτήρες διαβιβάζονται στη μαγνητική κεφαλή.
3. Οι χαρακτήρες γράφονται στο αρχείο.

Στη συνέχεια δίνουμε την εντολή:

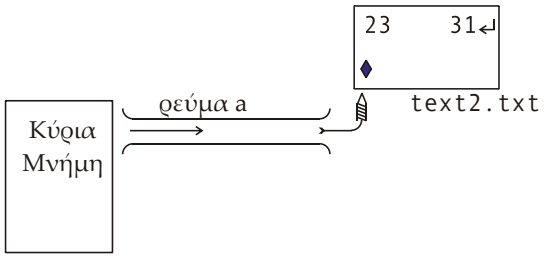
```
a << "    " << 31 << endl;
```



Σχ. 8-5 Παράσταση αρχείου-κειμένου (text2.txt) και ρεύματος *a* μετά την εκτέλεση της εντολής: `a << 23`.

[3] Όρος της C++ "(*std*::*ios*::)*bad*", θα τον δούμε αργότερα...

και το αποτέλεσμα είναι αυτό του Σχ. 8-6. Πρόσεξε τα τέσσερα διαστήματα: ζητήσαμε να γραφούν και γι' αυτό γράφηκαν. Αν δίναμε: "`a << 31`", το "`31`" θα «κολλούσε» στο "`23`" και θα είχαμε "`2331`".



Σχ. 8-6 Η κατάσταση μετά την εκτέλεση της εντολής: "`a << "  " << 31 << endl`". Λόγω του "`endl`" γράφηκε το σημάδι τέλους γραμμής και η κεφαλή είναι έτοιμη να γράψει στην αρχή της επόμενης γραμμής.

Το `endl` μας πηγαίνει στην αρχή νέας γραμμής. Είχαμε πει ότι γράφοντας στο *cout* τον `'\n'` έχουμε το ίδιο αποτέλεσμα· ισχύει αυτό και για τα αντικείμενα της κλάσης *ofstream*, όπως είναι το *a*; Ναι! Για την ακρίβεια το `'\n'` (= `char`(10)) είναι το σημάδι τέλους γραμμής για αρχεία-κείμενα που γράφει η C++. Τώρα όμως μπορούμε να δούμε τη διαφορά:

- Ζητώντας να γραφεί το `endl` πετυχαίνεις δύο πράγματα:
  - να πας σε νέα γραμμή και
  - να αντιγραφεί το όποιο περιεχόμενο του ενταμιευτή (η γραμμή που συμπληρώθηκε) στο αρχείο.
- Ζητώντας να γραφεί το `'\n'` πετυχαίνεις μόνο να πας σε νέα γραμμή. Το περιεχόμενο του ενταμιευτή θα αντιγραφεί όταν αυτός γεμίσει ή όταν ζητηθεί κάτι τέτοιο.

Το `endl` κάνει το πρόγραμμά σου *πιο αργό αλλά ασφαλέστερο*. Δηλαδή: αν, ας πούμε, έχεις διακοπή στην παροχή ηλεκτρικής ενέργειας την ώρα που δημιουργείς το αρχείο, όσες γραμμές ζήτησες να γραφούν, θα έχουν γραφεί στο αρχείο. Σε μια τέτοια περίπτωση, αν γράφεις με το `'\n'`, θα χάσεις όλο το περιεχόμενο του ενταμιευτή.

Όταν τελειώσεις το γράψιμο του αρχείου μην ξεχάσεις να το κλείσεις με μια *close()*. Η *close()* θα φροντίσει να αντιγράψει στο αρχείο όλο το περιεχόμενο του ενταμιευτή.

Το παρακάτω πρόγραμμα διαβάζει από το πληκτρολόγιο ζεύγη ακέραιων τιμών και τα γράφει, ένα ζεύγος ανά γραμμή σε αρχείο-κείμενο με όνομα text2.txt. Η ανάγνωση τελειώνει με το ζεύγος (0,0).

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ofstream a;
   int x, y;

   a.open( "text2.txt" );
   cout << " Δωσε τα x, y. 0,0 για τελος: ";  cin >> x >> y;
   while ( x != 0 || y != 0 )
   {
      a.width( 5 );  a << x;
      a.width( 5 );  a << y << endl;
      cout << " Δωσε τα x, y. 0,0 για τελος: ";  cin >> x >> y;
   } // while
   a.close();
} // main
```

Παρατηρήσεις: ▶

1. "Τι είναι το `a.width(5)`;" Ακριβώς σαν το `cout.width(5)`. Ζητάει να γραφεί το επόμενο στοιχείο στο αρχείο που είναι συνδεμένο με το *a* σε 5 θέσεις τουλάχιστον. Παρομοίως, για τα μέλη της κλάσης *ofstream* δουλεύουν όπως ξέρουμε οι μέθοδοι *precision* και *setf*.
2. Ας αλλάξουμε τη συνθήκη της **while**:

```
   cout << " Δωσε τα x, y. <ctrl-Z> για τελος: ";  cin >> x >> y;
```

```
   while ( !cin.eof() )
   {
      a.width( 5 );  a << x;
      a.width( 5 );  a << y << endl;
      cout << " Δωσε τα x, y. <ctrl-Z> για τελος: ";
      cin >> x >> y;
   } // while
```

Η εκτέλεση των επαναλήψεων θα τερματισθεί αν πληκτρολογήσουμε <ctrl-Z>.[4] Έτσι, δεν μας χρειάζεται φρουρός! ◄

8.5 Ένα «Πραγματικό» Πρόβλημα

Το παράδειγμα επεξεργασίας αρχείου που είδαμε στις προηγούμενες παραγράφους δεν είχε και πολύ νόημα. Εκεί θέλαμε να τραβήξουμε την προσοχή σου στις πράξεις του αρχείου. Τώρα ας δούμε ένα πιο «πραγματικό» πρόβλημα.

Έχουμε ένα αρχείο –με όνομα στο δίσκο exp4.txt– με τιμές τύπου **double**. Το πλήθος τους δεν είναι γνωστό. Θέλουμε να βρούμε και να τυπώσουμε το άθροισμα, τη Μέση Αριθμητική Τιμή και το πλήθος των πραγματικών αριθμών που υπάρχουν στο αρχείο. Ακόμη θέλουμε: το πλήθος, το άθροισμα και τη μέση τιμή όσων από τις τιμές που διαβάζει είναι θετικές και μέχρι 10.

Η αντίδρασή σου θα πρέπει να είναι: «Ωχ, πάλι αυτό!» ή «Μα αυτό το ξέρω!». Ας υποθέσουμε ότι είναι η δεύτερη και ας προχωρήσουμε. Γύρισε στο Κεφ. 6 (§6.1.2) για να δεις το πρόγραμμα *Μέση Τιμή* 4, ξαναδές το γενικό σχήμα επεξεργασίας σειριακού αρχείου (Πλ. 8.1) και ας γράψουμε το *Μέση Τιμή* 5.

Ας πούμε ότι το αρχείο θα το δουλέψουμε με το ρεύμα *a*. Θα πρέπει να το δηλώσουμε:

```
   ifstream a;
```

Με την τιμή - φρουρό τι θα κάνουμε; Τίποτε! *Δεν μας χρειάζεται!* Αντί να ψάχνουμε για φρουρό θα ελέγχουμε μήπως φτάσαμε στο τέλος του αρχείου. Να το γενικό σχήμα επεξεργασίας του αρχείου:

```
   a.open( "exp4.txt" );
   a >> x;
   while ( !a.eof() )
   {
      E
      a >> x;
   } // while
   a.close();
```

Ποιές είναι οι *E*; Νάτες:

```
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
```

Στη συνέχεια βλέπεις το πρόγραμμα *Μέση Τιμή* 5. Οι διαφορές του από το *Μέση Τιμή* 4 είναι πολύ μικρές. Και ο μόνος λόγος που ξαναπαραθέτουμε ολόκληρο, είναι ακριβώς για να δεις αυτήν την ομοιότητα και να επισημάνεις τις διαφορές.

```
// πρόγραμμα: Μέση Τιμή 5
#include <iostream>
#include <fstream>
using namespace std;
int main()
```

[4] Δοκίμασέ το και αν δουλέψει…

```
{
   ifstream a;

   int n;          // Μετρητής όλων των στοιχείων
   int selN;       // Μετρητής επιλεγόμενων στοιχείων
   double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;     // Το άθροισμα όλων των στοιχείων
   double selSum;  // Άθροισμα επιλεγόμενων στοιχείων
   double avrg;    // Μέση Αριθμητική Τιμή όλων των στοιχείων
   double selAvrg; // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων

   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
   a.open( "exp4.txt" );
   a >> x;
   while ( !a.eof() )
   {
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
      a >> x;
   } // while
   a.close();
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   } // if ( n
   cout << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
   if ( selN > 0 )
   {
      selAvrg = selSum/selN;
      cout << " ΑΘΡΟΙΣΜΑ = " << selSum
           << "  <x> = " << selAvrg << endl;
   } // if ( selN
} // main
```

Πρόσεξε ότι κλείνουμε το αρχείο αμέσως μόλις παύουμε να ασχολούμαστε μαζί του.

Ας πούμε τώρα ότι έχουμε το εξής πρόβλημα: Θέλουμε τα αποτελέσματα της επεξεργασίας όχι στην οθόνη αλλά σε ένα αρχείο-κείμενο με το όνομα report.txt. Τι θα πρέπει να αλλάξουμε στο *Μέση Τιμή* 5;

- Θα πρέπει να δηλώσουμε ένα ρεύμα *b* κλάσης *ofstream*, που θα το ανοίξουμε προς το report.txt.
- Ό,τι στέλνουμε στην οθόνη μέσω του *cout* να το στείλουμε στο report.txt μέσω του *b*.

```
// πρόγραμμα: Μέση Τιμή 6
#include <fstream>
using namespace std;
int main()
{
   ifstream a;
   ofstream b;
// . . .
   b.open( "report.txt" );
   b << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      b << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   } // if ( n
   b << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
```

```
   if ( selN > 0 )
   {
      selAvrg = selSum/selN;
      b << " ΑΘΡΟΙΣΜΑ = " << selSum
        << "  <x> = " << selAvrg << endl;
   } // if ( selN
```

Ε! Ξεχάσαμε την "`#include <iostream>`"! Δεν την ξεχάσαμε! Την παραλείψαμε διότι δεν τη χρειαζόμαστε: ούτε το *cin* ούτε το *cout* υπάρχει στο πρόγραμμά μας.

Να πώς περίπου θα είναι το περιεχόμενο του report.txt:

```
Διάβασα 37 αριθμούς
ΑΘΡΟΙΣΜΑ = 560.767  <x> = 15.1559
Διάλεξα 12 αριθμούς ε (0,10]
ΑΘΡΟΙΣΜΑ = 46.6869  <x> = 3.89057
```

8.6 Και Διάβασμα και Γράψιμο

Ας δούμε τώρα το εξής πρόβλημα: Θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει άγνωστο πλήθος πραγματικών αριθμών x_1, x_2, ... –που καθένας τους είναι πολύ μικρότερος από 10000– και θα τους αποθηκεύει σε ένα αρχείο fltnum.txt. Θέλουμε ακόμη:

α) να υπολογίζει και να τυπώνει τη Μέση Τιμή τους <*x*>,

β) να υπολογίζει και να τυπώνει τις τιμές της συνάρτησης: $e^{\frac{<x>-x}{<x>}}$ για τους αριθμούς αυτούς. Δηλαδή τα $y_k = e^{\frac{<x>-x_k}{<x>}}$.

Ξέρουμε να κάνουμε τα πάντα! Ένα σημείο χρειάζεται προσοχή: για να υπολογίσουμε τη μέση τιμή θα πρέπει να έχουμε όλους τους αριθμούς· δηλαδή *θα την έχουμε υπολογίσει όταν θα έχουμε γράψει όλους τους αριθμούς στο αρχείο*. Για να υπολογίσουμε τα y_k θα πρέπει να ξαναδιαβάσουμε τους αριθμούς, όχι βέβαια από το πληκτρολόγιο (όχι και να τους ξαναπληκτρολογήσουμε), αλλά από το αρχείο.

Να λοιπόν το σχέδιό μας:

```
Πέρασε τους αριθμούς στο αρχείο και υπολόγισε πλήθος και μέση τιμή
Διάβασε όλους τους αριθμούς από το αρχείο και για κάθε έναν από
                    αυτούς (xk) υπολόγισε και γράψε το yk = e^((<x>-xk)/<x>)
```

Φυσικά θα πρέπει να προσέξουμε: Το δεύτερο βήμα θα εκτελεσθεί μόνο αν το πλήθος των αριθμών που δώσαμε (*n*) δεν είναι μηδέν:

```
Πέρασε τους αριθμούς στο αρχείο και υπολόγισε πλήθος και μέση τιμή
if ( n > 0 )
{  Διάβασε όλους τους αριθμούς από το αρχείο και για κάθε έναν από
                    αυτούς (xk) υπολόγισε και γράψε το yk = e^((<x>-xk)/<x>)  }
```

Το πρώτο βήμα είναι κατά βάση το *Μέση Τιμή* 3 με τις εξής διαφορές:

- Θα πρέπει να φυλάγουμε στο αρχείο κάθε τιμή που δίνεται.
- Θα πρέπει να αλλάξουμε φρουρό: το 0 δεν αποκλείεται ενώ μια καλή επιλογή είναι το 9999.

```
   const double sentinel = 9999.0;
   ofstream  a;
// . . .
   sum = 0;  n = 0;
   a.open( "fltnum.txt" );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
```

```
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   a.close();
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
   }
```

Για το δεύτερο βήμα τροποποιούμε το πρόγραμμα της προηγούμενης παραγράφου:

```
   ifstream  b;      // για διάβασμα αρχείου

      b.open( "fltnum.txt" );   m = 0;
      b >> x;
      while ( !b.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         b >> x;
      } // while
      b.close();
```

Όλα αυτά θα γίνουν βέβαια αν και μόνον αν $n > 0$.

Να ολόκληρο το πρόγραμμα:

```
// πρόγραμμα: Μέση Τιμή 3+
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const double sentinel = 9999.0;
   ofstream  a;
   ifstream  b; // για διάβασμα αρχείου
   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)
   int m;
   double y;

   sum = 0;  n = 0;
   a.open( "fltnum.txt" );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   a.close();
   cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
   if ( n > 0 )
```

```
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
      b.open( "fltnum.txt" );   m = 0;
      b >> x;
      while ( !b.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         b >> x;
      } // while
      b.close();
   }
} // main
```

Στο πρόγραμμα αυτό χρησιμοποιήσαμε δύο διαφορετικά ρεύματα για να διαχειριστούμε το ίδιο αρχείο· δύο διαφορετικούς «μονόδρομους». Δεν θα μπορούσαμε να χρησιμοποιήσουμε έναν μόνο «δρόμο διπλής κατεύθυνσης»; Ναι! Αρκεί να δηλώσουμε ρεύμα κλάσης *fstream*. Τι διαφορές θα έχουμε στην περίπτωση αυτή;

- Στη δήλωση:

```
   fstream  a;
```

- Στο άνοιγμα: Η *open()* παίρνει και μια δεύτερη παράμετρο, που καθορίζει τον τρόπο διαχείρισης του αρχείου: αν δώσουμε "`ios_base::in`" ανοίγουμε το αρχείο για διάβασμα, αν δώσουμε "`ios_base::out`" το ανοίγουμε για γράψιμο, αν δώσουμε

  "`ios_base::in|ios_base::out`"

  το ανοίγουμε και για διάβασμα και για γράψιμο.[5] Θα πρέπει λοιπόν να το ανοίξουμε ως εξής:

```
   a.open( "fltnum.txt", ios_base::in|ios_base::out );
```

- Στο κλείσιμο και την επαναφορά: Θα κλείσουμε το ρεύμα μια φορά μόνον στο τέλος. Πώς όμως θα επαναφέρουμε το αρχείο στην αρχή του για να το ξαναδιαβάσουμε; Η *fstream* (και η *ifstream*) έχει μια μέθοδο, τη *seekg()*, που όταν κληθεί με όρισμα 0 – στην περίπτωσή μας: "`a.seekg(0)`"– ξαναφέρνει το ρεύμα στην αρχή του αρχείου για να (ξανα)διαβαστεί.

  Τα υπόλοιπα παραμένουν ίδια:

```
// πρόγραμμα: Μέση Τιμή 3+ inout
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const double sentinel = 9999.0;
   fstream  a;  // για διάβασμα αρχείου και γράψιμο
   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)
```

[5] Μοιάζουν λιγάκι με κινέζικα; Δεν πειράζει! Θα τα χρησιμοποιείς όπως ακριβώς τα δίνουμε και αργότερα θα καταλάβεις τι ακριβώς συμβαίνει. Όπως θα μάθουμε αργότερα, αυτό μπορεί να το δεις και ως: "`ios_base::in+ios_base::out`".

```
   int m;
   double y;

   sum = 0;  n = 0;
   a.open( "fltnum.txt", ios_base::in|ios_base::out );
   cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
   // γράψε στο αρχείο την τιμή που πήρες
      a << x << endl;
   // πρόσθεσέ την στην sum και μέτρησέ την
      n = n + 1;
      sum = sum + x;
      cout << " Δώσε αριθμό - 9999 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα και έγραψα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum
           << "  <x> = " << avrg << endl;
      a.seekg( 0 );
      a >> x;
      while ( !a.eof() )
      {
         m = m + 1;
         y = exp( (avrg - x)/avrg );
         cout << " x[";
         cout.width(3);  cout << m << "] = ";
         cout.width(6);  cout << x << "   y[";
         cout.width(3);  cout << m << "] = " ;
         cout.width(6);  cout << y << endl;
         a >> x;
      } // while
   }
   a.close();
} // main
```

8.7 Μυστικά και Ψέματα

Το πρόγραμμα της §8.5 ήταν «ψεύτικο», όπως «μισοψεύτικα» είναι και τα προγράμματα της προηγούμενης παραγράφου! Δηλαδή; δεν δουλεύουν; Δουλεύουν μια χαρά, αλλά κανείς δεν θα καθήσει να δημιουργήσει ένα αρχείο-κείμενο τροφοδοτώντας ένα πρόγραμμα σαν τα παραπάνω! Και πώς γίνεται η δουλειά;

Ας πούμε ότι έχεις κάνει κάποιες μετρήσεις και έχεις μαζέψει μερικές σελίδες με νούμερα που θέλεις να τα επεξεργαστείς με κάποιο πρόγραμμα. Τι θα κάνεις; *Θα περάσεις τα νούμερα σε ένα αρχείο-κείμενο από κάποιον κειμενογράφο!* Αλλά, προσοχή: αν δεν χρησιμοποιήσεις έναν απλόν κειμενογράφο, σαν το Notepad των Windows (ή κάτι παρόμοιο) και προτιμήσεις έναν επεξεργαστή κειμένου, σαν το Word, φρόντισε *να φυλάξεις το αρχείο σου σε μορφή text* (ή ascii ή όπως αλλιώς το λέει).

Αυτός ο τρόπος έχει τα εξής πλεονεκτήματα:

- Δεν χρειάζεται να γράψεις πρόγραμμα.
- Έχεις δυνατότητα να κάνεις διορθώσεις[6].

Δηλαδή, τζάμπα μάθαμε να δημιουργούμε αρχεία με πρόγραμμα; Όχι βέβαια! Αλλά θα δημιουργούμε αρχεία με στοιχεία που θα δημιουργούνται από το πρόγραμμά μας ή θα

[6] Αν προσπαθήσεις να κάνεις το πρόγραμμά σου με δυνατότητα διόρθωσης, γίνεται πιο πολύπλοκο και… άσε καλύτερα.

προκύπτουν από επεξεργασία άλλων αρχείων. Όχι αρχεία που θα τα γεμίζουμε από το πληκτρολόγιο.

8.8 Πάγια Ρεύματα

Εκτός από τις κλάσεις *fstream*, *ifstream*, *ofstream*, που δηλώνονται στο `fstream`, υπάρχουν: η *istream* για ρεύματα από το αρχείο προς το πρόγραμμα και η *ostream* για ρεύματα από το πρόγραμμα προς το αρχείο.

Όλες οι διάλεκτοι της C++ παρέχουν τρία ρεύματα τα οποία μπορείς να υποθέσεις ότι δηλώνονται ως εξής:

```
istream  cin;
```

Αυτό είναι το **πάγιο ρεύμα εισόδου** (standard input) και, όπως έχουμε δει, αποκαθιστά ροή από το πληκτρολόγιο προς το πρόγραμμά μας.

```
ostream  cout;
```

Πρόκειται για το **πάγιο ρεύμα εξόδου** (standard output) και, όπως έχουμε δει, αποκαθιστά ροή από το πρόγραμμα προς την οθόνη. Τέλος:

```
ostream  cerr;
```

Αυτό είναι το ρεύμα όπου γράφονται τα μηνύματα λάθους (standard error)· συνήθως αντιστοιχεί στην οθόνη. Για την ίδια δουλειά υπάρχει και ένα τέταρτο πάγιο ρεύμα, το:

```
ostream  clog;
```

Αυτό είναι το ρεύμα προς το **πάγιο αρχείο καταγραφών** (standard log file)· συνήθως αντιστοιχεί στην οθόνη.

Μπορείς να θεωρήσεις ότι αυτά τα ρεύματα ανοίγονται αυτόματα, όταν αρχίζει η εκτέλεση του (κάθε) προγράμματός σου, το πρώτο για διάβασμα και τα άλλα για γράψιμο.

8.9 Αρχείο–Κείμενο: Άλλες Επεξεργασίες

Ο τρόπος που παρουσιάσαμε το αρχείο-κείμενο (text) είναι πρωτοφανής διεθνώς: κανείς δεν ξεκινάει να δουλεύει με τέτοια αρχεία διαβάζοντας και γράφοντας αριθμούς. Τώρα θα δούμε και τις επεξεργασίες που κάνει όλος ο κόσμος.

Όπως είπαμε, μπορούμε να σκεφτόμαστε ένα αρχείο-κείμενο ως εξής::

γραμμή
γραμμή
. . .
γραμμή, τέλος αρχείου

και *γραμμή*:

χαρακτήρας, ..., χαρακτήρας, τέλος γραμμής

ή

τέλος γραμμής

Η δεύτερη περίπτωση είναι η **κενή γραμμή**.

Για να χειριστείς αρχεία-κείμενα μπορείς να χρησιμοποιήσεις, εκτός από τους τελεστές "`<<`" και "`>>`" που ήδη είδαμε, τις μεθόδους *get()*, *peek()*, των κλάσεων *ifstream* και *fstream* και *put()* των κλάσεων *ofstream* και *fstream*.

Το τέλος της γραμμής τί είναι; Κάθε κειμενογράφος έχει ένα ειδικό σημάδι για να το σημειώνει. Όπου χρησιμοποιείται το σύνολο χαρακτήρων ASCII, συνήθως, το τέλος γραμμής σημειώνεται με δυο (μη-εκτυπώσιμους) χαρακτήρες:

- CR (Carriage Return) που είναι 13ος στον πίνακα (`'\r'` για τη C++) και
- LF (Line Feed), που είναι 10ος στο πίνακα (`'\n'` για τη C++).

Με αυτόν τον τρόπο αποθηκεύονται στο κείμενο οι λειτουργίες της οθόνης: όταν πιέζουμε το <enter> για να αλλάξουμε γραμμή: πρέπει να πάμε στην αρχή της γραμμής (CR) και να κατέβουμε στην επόμενη γραμμή (LF).

Όταν διαβάζουμε ένα αρχείο ως κείμενο (text mode) η C++ κάνει τα εξής: όταν βρίσκει ζευγάρι CR LF μας δίνει στο πρόγραμμα μόνο το LF (`'\n'`). Όταν γράφουμε κάνει το αντίστροφο: κάθε φορά που γράφουμε `'\n'`, βάζει CR LF.

Μερικοί κειμενογράφοι βάζουν για τέλος γραμμής τον έναν μόνον από τους δύο χαρακτήρες ή, με την αντίστροφη σειρά, LF CR.

Ας υποθέσουμε λοιπόν ότι θέλουμε να διαβάσουμε και να επεξεργαστούμε χαρακτήρα προς χαρακτήρα, ένα αρχείο text (με τη βοήθεια του `ifstream t`). Ήδη στην §4.4 είδαμε ότι αυτή η δουλειά δεν μπορεί να γίνει με τον τελεστή "`>>`" διότι αυτός «τρώει» τα κενά, τις αλλαγές γραμμής κλπ. Όπως εκεί, έτσι και εδώ θα χρησιμοποιήσουμε τη μέθοδο *get()*. Δίνοντας:

```
t.get( ch );
```

ζητάμε από το ρεύμα *t* να μας φέρει τον επόμενο χαρακτήρα από το αρχείο και να τον αποθηκεύσει ως τιμή της μεταβλητής *ch* (τύπου `char`).

Συνηθέστατα, όταν επεξεργαζόμαστε ένα αρχείο-κείμενο πρέπει να ξεχωρίζουμε τις γραμμές· εκτός από την επεξεργασία κάθε χαρακτήρα χωριστά, κάνουμε και άλλες επεξεργασίες συνολικά στη γραμμή. Πώς ανιχνεύουμε το *τέλος γραμμής*; Σύμφωνα με αυτά που είπαμε, τέλος γραμμής έχουμε όταν: "`ch == '\n'`".

Σημείωση: ▶
Συνήθως –και για διάφορα «σημάδια» τέλους γραμμής– η C++ θα σου επιστρέψει ένα απλό `'\n'`· τουλάχιστον όταν αυτό το σημάδι περιλαμβάνει και τον `'\n'`.◀

Μπορείς να σκεφτείς την επεξεργασία αρχείου text ως εξής:

```
while ( δεν τελείωσε το αρχείο )
{
   Διάβασε μια γραμμή
   Επεξεργάσου τη γραμμή
}
```

Το γενικό σχήμα επεξεργασίας ενός αρχείου text, που βλέπεις στο Πλ. 8.2, έχει περισσότερες λεπτομέρειες. Όπως βλέπεις, έχουμε δύο **while**, τη μια φωλιασμένη μέσα στην άλλη. Η εσωτερική **while** είναι αυτό που λέμε «Διάβασε γραμμή» αλλά έχει ενσωματωμένη την επεξεργασία του κάθε χαρακτήρα που διαβάζεται.

Πλαίσιο 8.2

Επεξεργασία Αρχείου Text

```
ifstream t;
    :
t.open( όνομα αρχείου );
t.get( ch );
while ( !t.eof() )
{
// προχώρα μέχρι το τέλος της γραμμής
   while ( !t.eof() && δεν βρήκαμε τέλος γραμμής )
   {
      Εντολές επεξεργασίας του ch
      t.get( ch );
   } // while (δεν βρήκαμε τέλος γραμμής…
   Εντολές επεξεργασίας γραμμής
   if ( !t.eof() ) πήγαινε στην αρχή της επόμενης γραμμής;
   t.get( ch );
}// while (!t.eof())
t.close();
```

Η εσωτερική **while** γράφεται:

```
while ( !t.eof() && ch != '\n' )
{
   Εντολές επεξεργασίας του ch
   t.get(ch);
} // while (δεν βρήκαμε τέλος γραμμής…
```

Για το πέρασμα στη νέα γραμμή αρκεί η `t.get(ch)` που υπάρχει στο τέλος της περιοχής της εξωτερικής **while**.

Όταν τελειώσει η εκτέλεση της εσωτερικής **while** θα συμβαίνει ένα από τα παρακάτω:

- `t.eof()` (τέλος αρχείου),
- `ch == '\n'` (τέλος γραμμής).

Φυσικά, θα προσπαθήσουμε να περάσουμε σε νέα γραμμή αν δεν έχουμε την πρώτη περίπτωση. Γι' αυτό, στο Πλ. 8.2 λέμε πιο προσεκτικά:

```
   if ( !t.eof() ) πήγαινε στην αρχή της επόμενης γραμμής;
```

Σημείωση: ▶

Με την ευκαιρία να πούμε ότι: Αν είμαστε στο τέλος του αρχείου, δηλαδή η `t.eof()` δίνει `true`, η ακριβώς προηγούμενη `t.get(ch)` έχει βάλει στη *ch* τιμή `char(-1)` (ή `unsigned char(255)`).◀

Όπως είπαμε και παραπάνω, στις κλάσεις *ofstream* και *fstream* υπάρχει η μέθοδος *put()*. Αν λοιπόν γράφουμε σε κάποιο αρχείο-κείμενο μέσω του ρεύματος *s* (ας πούμε, κλάσης *ofstream*), η

```
      s.put( ch );
```

γράφει την τιμή της *ch* (τύπου `char`) στο αρχείο.

Η μέθοδος *get()*, όπως ξέρουμε, υπάρχει και στην κλάση *istream* (την έχει το ρεύμα *cin*). Η μέθοδος *put()* υπάρχει και στην κλάση *ostream* (την έχει και το ρεύμα *cout*).

Στη συνέχεια βλέπεις ένα πολύ απλό πρόγραμμα: αντιγράφει ένα αρχείο-κείμενο (το text1.txt) σε ένα άλλο (το numdta.txt).

```
#include <fstream>
using namespace std;

int main()
{
   ifstream s( "text1.txt" );
   ofstream t( "numdta.txt" );
   char ch;

   s.get(ch);
   while ( !s.eof() )
   {  t.put(ch);  s.get(ch);  } // while
   t.close();  s.close();
} // main
```

Στην επόμενη παράγραφο θα δεις παραδείγματα με πιο πλήρη εφαρμογή του γενικού σχήματος.

8.10 Παραδείγματα

Θα δούμε τώρα δύο χαρακτηριστικά παραδείγματα επεξεργασίας αρχείου text. Και τα δύο θα επεξεργαστούν το εξής κείμενο, που είναι αντιγραμμένο από τον πρόλογο του C.A.R. Hoare στο (Hoare & Shepherdson 1985).

```
The strong connection between the formalization of
mathematical logic and the formalization of computer
programming languages was clearly recognized by Alan
Turing as early as 1947, when, in a talk on 20 February
to the London Mathematical Society, he reported his
```

```
expectation
   'that digital computing machines will eventually
   stimulate a considerable interest in symbolic logic
   and mathematical philosophy. The language in which
   one communicates with these machines, i.e. the language
   of instruction tables, forms a sort of symbolic logic'.
```

Χρησιμοποιώντας έναν κειμενογράφο, γράψε αυτό το κείμενο και φύλαξέ το σε ένα αρχείο με όνομα alturing.txt.

Παράδειγμα 1

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο που έχει το παραπάνω κείμενο και θα μετράει: πόσες γραμμές έχει, πόσα κεφαλαία γράμματα έχει και πόσα ψηφία έχει.

Το τι θα κάνουμε μας είναι γνωστό. Μας χρειάζονται τρεις μετρητές, για τα τρία μεγέθη που θα μετράμε, που αρχικά θα πρέπει να μηδενιστούν. Τα αποτελέσματά μας δεν μπορούν να βγουν πριν από το τέλος της επεξεργασίας του αρχείου. Το πρόγραμμά μας θα είναι περίπου:

```
int main()
{
   ifstream t;    // ρευμα απο το αρχειο με το κειμενο
   int  nRows;    // μετρητής γραμμών
   int  nUpCase;  // μετρητής κεφαλαίων
   int  nDigits;  // μετρητής ψηφίων
   char ch;       // χαρακτήρας που διαβάζουμε

   Μηδένισε τους μετρητές;
   Επεξεργάσου το αρχείο;
   Λέγε τα αποτελέσματα;
}
```

Η πρώτη «εντολή» υλοποιείται εύκολα:

```
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
```

το ίδιο και η τρίτη:

```
// Λέγε τα αποτελέσματα
   cout << " Διάβασα " << nRows << " γραμμές" << endl;
   cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
        << nDigits << " ψηφία" << endl;
```

Ας δούμε τώρα τι γίνεται με την «Επεξεργάσου το αρχείο». Αυτή θα ακολουθεί το γενικό σχήμα επεξεργασίας αρχείου text:

```
// Επεξεργάσου το αρχειο
   t.open( "alturing.txt" );
   t.get( ch );
   while ( !t.eof() )
   {
   // προχώρα μέχρι το τέλος της γραμμής
      while ( !t.eof() && ch == '\n' )
      {
         Εντολές επεξεργασίας του ch
         t.get( ch );
      } // while (δεν βρήκαμε τέλος γραμμής…
      Εντολές επεξεργασίας γραμμής
      if ( !t.eof() ) t.get( ch );
   }// while (!t.eof())
   t.close();
```

Το πρόβλημά μας τώρα είναι να δούμε τι θα είναι οι «*Εντολές επεξεργασίας του ch*» και οι «*Εντολές επεξεργασίας γραμμής*».

Οι «*Εντολές επεξεργασίας του ch*» τι θα κάνουν; Θα ελέγχουν τον κάθε χαρακτήρα που διαβάζεται και εφόσον είναι κεφαλαίο γράμμα ή ψηφίο θα αυξάνεται ο αντίστοιχος μετρητής. Μπορούμε να την υλοποιήσουμε με την:

```
         if ( isupper(ch) ) nUpCase = nUpCase + 1;
```

```
         else if ( isdigit(ch) )  nDigits = nDigits + 1;
```

Τις *isupper* και *isdigit* τις θυμάσαι; Αν όχι, γύρισε πίσω, στον Πίν. 4-3. Για να τις χρησιμοποιήσουμε θα πρέπει να βάλουμε `#include <cctype>`.

Τέλος, ας δούμε και τις «*Εντολές επεξεργασίας γραμμής*»: Κάθε φορά που τελειώνει μια γραμμή, θα πρέπει να αυξάνουμε το μετρητή γραμμών κατά 1:

```
      nRows = nRows + 1;
```

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cctype>

using namespace std;

int main()
{
   ifstream  t;  // ρεύμα απο το αρχείο με το κείμενο
   int  nRows;   // μετρητής γραμμών
   int  nUpCase; // μετρητής κεφαλαίων
   int  nDigits; // μετρητής ψηφίων
   char ch;      // χαρακτήρας που διαβάζουμε
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
// Επεξεργάσου το αρχείο
   t.open( "alturing.txt" );
   t.get( ch );
   while ( !t.eof() )
   {
      while ( !t.eof() && ch != '\n' )
      {
         if ( isupper(ch) ) nUpCase = nUpCase + 1;
         else if ( isdigit(ch) )  nDigits = nDigits + 1;
         t.get(ch);
      }
      nRows = nRows + 1;
      if ( !t.eof() )  t.get(ch);
   } // while (!t.eof())
   t.close();
// Λέγε τα αποτελέσματα
   cout << " Διάβασα " << nRows << " γραμμές" << endl;
   cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
        << nDigits << " ψηφία" << endl;
} // main
```

Το αποτέλεσμα που μας δίνει είναι:

```
 Διάβασα 11 γραμμές
 Μέτρησα 8 κεφαλαία γράμματα και 6 ψηφία
```

Παράδειγμα 2

Θέλουμε να επεξεργαστούμε ξανά το κείμενο που μετρήσαμε στο προηγούμενο παράδειγμα, αλλά με διαφορετικό στόχο: Θέλουμε να πάρουμε το ίδιο κείμενο γραμμένο με κεφαλαία.

Κατ' αρχάς θα πρέπει να δηλώσουμε τα δύο ρεύματα με τα οποία θα χειριστούμε τα αρχεία:

```
   ifstream  a;    // το αρχείο με το κείμενο
   ofstream  b;    // το αρχείο με τα κεφαλαία
```

και να τα ανοίξουμε:

```
   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
```

Την άσκ. 7-8 την έλυσες; Ζητούσαμε να γράψεις μια:

```
   // upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
   //           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
   char upCase( char ch )
```

Αν την έλυσες, τότε τα πράγματα είναι πολύ απλά:

- Οι «*Εντολές επεξεργασίας του ch*» θα μετατρέπουν το χαρακτήρα σε κεφαλαίο και θα τον γράφουν στο νέο αρχείο, δηλαδή:

  `co = upCase(ch);  b.put(co);`

- Οι «*Εντολές επεξεργασίας γραμμής*» δεν είναι τίποτε ιδιαίτερο: απλώς μετά το τέλος της γραμμής θα πρέπει να βάζουμε ένα `endl`.

  Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

   char upCase( char ch );

int main()
{
   ifstream  a;  // το αρχείο με το κείμενο
   ofstream  b;  // το αρχείο με τα κεφαλαία
   char ch, co;

   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
   a.get( ch );
   while ( !a.eof() )
   {
      while ( !a.eof() && (ch != '\n') )
      {  co = upCase(ch);  b.put(co);  a.get(ch);  }
      b << endl;
      if ( !a.eof() )  a.get( ch );
   }
   b.close();
   a.close();
} // main

   // upCase -- Αν ο ch είναι μικρό Λατινικό γράμμα επιστρέφει
   //           το αντίστοιχο κεφαλαίο, αλλιώς τον ch
   char upCase( char ch )
   {
      char fv;

      if ( islower(ch) )  fv = char( int(ch)-32 );
                    else  fv = ch;
      return fv;
   } // upCase
```

Παρατήρηση: ▶

Φυσικά, θα μπορούσαμε πάρουμε τη λύση πιο απλά:

- Θυμήσου ότι η C++ σου δίνει την *toupper* που κάνει αυτά που ζητούμε από την *upCase*.
- Πάρε το πρόγραμμα αντιγραφής της προηγούμενης παραγράφου και άλλαξε την:

```
{  t.put(ch);  s.get(ch);  }
```

σε:

```
{  b.put(toupper(ch));  a.get(ch);  }
```

Δηλαδή:

```
   a.open( "alturing.txt" );
   b.open( "upcaltur.txt" );
   a.get( ch );
   while ( !a.eof() )
   {  b.put( toupper(ch) );  a.get( ch );  }
```

```
   b.close();  a.close(); ◀
```

🖘🖘🖘

8.11 Δουλεύοντας με Σιγουριά

Όταν κάνεις μια πράξη εισόδου/εξόδου (I/O) πολλά άσχημα μπορεί να συμβούν. Π.χ.

- να προσπαθήσεις να ανοίξεις για διάβασμα ένα αρχείο που δεν υπάρχει,
- να προσπαθήσεις να γράψεις σε μια δισκέτα που δεν έχει χώρο κ.ο.κ.

Η C++ σου δίνει τη δυνατότητα να ελέγχεις αν όλα πηγαίνουν καλά. Η μέθοδος *fail()* επιστρέφει `true` αν η προηγούμενη πράξη εισόδου/εξόδου απέτυχε αλλιώς `false`. Ας πούμε ότι έχεις δηλώσει:

```
   ifstream s;
```

και στη συνέχεια ζητάς:[7]

```
   s.open( "text1.txt" );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.txt" << endl;
   else // ok, προχώρα
   { // . . .
```

Τον ίδιο έλεγχο μπορείς να κάνεις και μετά την πράξη "`s.get(ch)`" ή την "`s >> x`".

Στη συνέχεια βλέπεις το πρόγραμμα της αντιγραφής αρχείου με πολλούς ελέγχους:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream s;
   ofstream t;
   char ch;

   s.open( "text1.txt" );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.txt" << endl;
   else // ok, προχώρα
   {
      t.open( "numdta.txt" );
      if ( t.fail() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το numdta.txt" << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
      {
         s.get(ch);
         while ( !s.fail() && !t.fail() )
         {  t.put( ch );  s.get(ch);  } // while
         if ( !s.eof() )
            cerr << "πρόβλημα κατά την αντιγραφή" << endl;
         t.close();
         s.close();
      } // if ( t.fail
   } // if ( s.fail
} // main
```

[7] Πάντως, σε πολλά βιβλία, θα δεις και έναν άλλον τρόπο: Αν δεις την *s* ως *συνθήκη*: αν το άνοιγμα του αρχείου έγινε επιτυχώς θα έχει τιμή που ερμηνεύεται ως `true`, αλλιώς θα έχει τιμή που ερμηνεύεται ως `false`. Έτσι, αντί για `if (s.fail())...` θα δεις το `if (!s)...`

Εξηγούμε και με λόγια το τι γίνεται:

- Αν δεν μπορέσαμε να ανοίξουμε το ρεύμα *s* από το αρχείο που διαβάζουμε δεν προχωρούμε.
- Αν τα καταφέραμε, ελέγχουμε αν ανοίξαμε το ρεύμα προς το αρχείο *t* που γράφουμε. Αν δεν άνοιξε, *κλείνουμε και το s* και σταματάμε.
- Αν όλα πήγαν καλά αντιγράφουμε με τη **while**. Πρόσεξε τη συνθήκη της που ελέγχει τα δύο ρεύματα. Η εκτέλεση των επαναλήψεων διακόπτεται αν εμφανιστεί πρόβλημα σε κάποιο από τα δύο.
- Αν τελειώσει το αρχείο η `s.fail()` θα δώσει `true` και οι επαναλήψεις θα σταματήσουν.
- Η **while** τελειώνει αν βρούμε πρόβλημα στο *s* ή στο *t*. Αν όμως το «πρόβλημα» είναι η *s.eof*`()` τότε όλα πήγαν καλά! Το μήνυμα "*πρόβλημα κατά την αντιγραφή*" εμφανίζεται μόνον αν βγήκαμε από τη **while** χωρίς να έχουμε *s.eof*`()`.

Ένας άλλος τρόπος για να δούμε αν το ρεύμα είναι ανοικτό είναι η κλήση της μεθόδου *is_open()*. Αυτή επιστρέφει τιμή `true` αν το ρεύμα είναι ανοικτό, αλλιώς `false`:

```
// . . .
   s.open( "text1.dta" );
   if ( !s.is_open() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο text1.dta" << endl;
   else
   {
      t.open( "numdta.txt" );
      if ( !t.is_open() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το numdta.txt" << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
// . . .
```

8.12 Τρόποι Ανοίγματος (Ρεύματος) Αρχείου

Είπαμε παραπάνω ότι η *open()* δέχεται και μια δεύτερη παράμετρο που έχει να κάνει με τον τρόπο χρήσης του αρχείου και είδαμε ήδη τις εξής περιπτώσεις:

- "`ios_base::in`" για διάβασμα από αρχείο (δεν χρειάζεται αν το ρεύμα είναι *ifstream*).
- "`ios_base::out`" για γράψιμο σε αρχείο (δεν χρειάζεται αν το ρεύμα είναι *ofstream*).
- Ο συνδυασμός "`ios_base::in | ios_base::out`" για αρχείο που το ανοίγουμε για διάβασμα και για γράψιμο (δεν χρειάζεται αν το ρεύμα είναι *ofstream*).

Παρ' όλο που εδώ δεν έχουμε μεταβλητές τύπου `bool`, οι "`ios_base::in`" και "`ios_base::out`" ονομάζονται **σημαίες** (flags) της *open()*. Το γιατί θα το καταλάβεις αργότερα. Θα τις δεις ακόμη και ως *τιμές τύπου* (*ios_base*`::`)*openmode*.

Εκτός από αυτούς μπορείς να χρησιμοποιήσεις τα εξής (και συνδυασμούς τους):""

"`ios_base::binary`"

για αρχεία που έχουν το περιεχόμενό τους όχι σε μορφή κειμένου αλλά όπως είναι στην εσωτερική παράσταση. Θα τα δούμε στη συνέχεια.

"`ios_base::trunc`"

για να σβήσεις το περιεχόμενο του αρχείου με το άνοιγμά του (όπως γίνεται με το άνοιγμα ενός *ofstream*).

Ας πούμε ότι έχουμε ένα αρχείο –το `temp.txt`– και θέλουμε να το ανοίξουμε για γράψιμο και διάβασμα. Πριν από όλα όμως θέλουμε να σβήσουμε το παλιό του περιεχόμενο. Πώς μπορούμε να το κάνουμε;

- Με όσα ξέρουμε μέχρι τώρα: Το ανοίγουμε ως

```
   fstream temp( "temp.txt", ios_base::out );
```

```
temp.close();
```

για να σβήσουμε το αρχικό του περιεχόμενο και μετά το ανοίγουμε πάλι ως:

```
fstream temp( "temp.txt", ios_base::in|ios_base::out );
```

- Με χρήση της "`ios_base::trunc`": Το ανοίγουμε, μια φορά μόνον, ως

```
fstream temp( "temp.txt", ios_base::in|ios_base::out|ios_base::trunc );
```

"`ios_base::ate`"
με το άνοιγμα του αρχείου τοποθετείται στο τέλος του.
"`ios_base::app`"
το αρχείο δεν σβήνεται και κάθε γράψιμο γίνεται στο τέλος του.

Στη συνέχεια δίνουμε ένα παράδειγμα για να συγκρίνεις τις δύο τελευταίες. Χρησιμοποιούμε την εντολή "`temp.seekp(0)`" που σημαίνει «πήγαινε *να γράψεις* στην αρχή του αρχείου». Η μέθοδος *seekp()* είναι διαθέσιμη για ρεύματα *ofstream* και *fstream*.

Παράδειγμα

Έστω ότι έχουμε το αρχείο `temp.txt` με περιεχόμενο:

```
Ιανουάριος
Φεβρουάριος
Μάρτιος
Απρίλιος
```

Μετά το άνοιγμα:

```
fstream temp( "temp.txt",
              ios_base::in|ios_base::out|ios_base::ate );
```

και τις εντολές:

```
temp << "Μάιος" << endl;
temp.seekp( 0 );
temp << "Ιούνιος" << endl;
temp.close();
```

το περιεχόμενο του αρχείου γίνεται:

```
Ιούνιος
ς
Φεβρουάριος
Μάρτιος
Απρίλιος
Μάιος
```

Δηλαδή:

- Με το άνοιγμα πηγαίνουμε στο τέλος του αρχείου και γράφουμε "`Μάιος`".
- Μετά την `temp.seekp(0)`, το γράψιμο της λέξης "`Ιούνιος`" γίνεται στην αρχή του αρχείου.

Αν ανοίξουμε το *temp* με την:

```
fstream temp( "temp.txt", ios_base::out|ios_base::app );
```

οι ίδιες εντολές θα κάνουν το αρχείο:

```
Ιανουάριος
Φεβρουάριος
Μάρτιος
Απρίλιος
Μάιος
Ιούνιος
```

Δηλαδή, παρά την `temp.seekp(0)`, όταν έρχεται η στιγμή να γραφεί η λέξη "`Ιούνιος`" το γράψιμο γίνεται στο τέλος του αρχείου.

Σχετικώς με τη χρήση συνδυασμών από τέτοιες σημαίες να επισημάνουμε τα εξής:

- Όλοι οι επιτρεπτοί συνδυασμοί φαίνοντα στον Πιν. 8-1.

- Μπορείς να τους χρησιμοποιήσεις σε ρεύματα *fstream*, *ifstream*, *ofstream*. Αν κατά το άνοιγμα δεν βάλεις δεύτερη παράμετρο
  - Το *fstream* ανοίγει με `ios_base::in | ios_base::out`.
  - Το *ifstream* ανοίγει με `ios_base::in`.
  - Το *ofstream* ανοίγει με `ios_base::out`.
- Η ύπαρξη της `ios_base::out` στον συνδυασμό έχει ως αποτέλεσμα να σβηστεί το αρχείο, αν υπάρχει. Αυτό *αναιρείται* αν ο συνδυασμός περιέχει και κάποιαν από τις `ios_base::in`, `ios_base::app` ενώ *επιβάλλεται* αν περιέχει την `ios_base::trunc`.

8.13 Χειρισμός Αρχείων με τα Εργαλεία της C

Παρ' όλο που η C++ διαθέτει ένα πολύ πλούσιο και ευέλικτο σύστημα για τη διαχείριση αρχείων, θα δεις συχνά προγράμματα που χρησιμοποιούν τα αντίστοιχα εργαλεία της C. Θεωρούμε λοιπόν αναγκαίο να ρίξουμε μια ματιά στα εργαλειά αυτά αν και δεν θα τα χρησιμοποιούμε στη συνέχεια.

Στη συνέχεια βλέπεις (ξανά) το πρόγραμμα «*Μέση Τιμή 6*» με διαχείριση αρχείων όπως την κάνει η C:

```
0: // πρόγραμμα: Μέση Τιμή 6 C
1: #include <cstdio>
2: using namespace std;
3: int main()
4: {
```

Συνδυασμός τιμών `ios_base`					Ισοδύναμος ορμαθός `cstdio`
`binary`	`in`	`out`	`trunk`	`app`	x
		+			"w"
		+		+	"a"
				+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
	+	+		+	"a+"
	+			+	"a+"
+		+			"wb"
+		+		+	"ab"
+				+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"
+	+	+		+	"a+b"
+	+			+	"a+b"

Πίν. 8-1 (Από το πρότυπο της C++) Αντίστοιχοι τρόποι ανοίγματος αρχείων στη C++ και στη C.
Ένα παράδειγμα για το διάβασμά του: Η τελευταία γραμμή σημαίνει: αν στην *open* της C++ βάλεις
`ios_base::binary | ios_base::in | ios_base::app`
στη *fopen()* της C θα βάλεις `"a+b"`.

```
 5:    FILE* a;
 6:    FILE* b;
 7:
 8:    int n;         // Μετρητής όλων των στοιχείων
 9:    int selN;      // Μετρητής επιλεγόμενων στοιχείων
10:    double x;      // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
11:    double sum;    // Το άθροισμα όλων των στοιχείων
12:    double selSum; // Άθροισμα επιλεγόμενων στοιχείων
13:    double avrg;   // Μέση Αριθμητική Τιμή όλων των στοιχείων
14:    double selAvrg;
15:                   // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων
16:    sum = 0;  n = 0;
17:    selSum = 0;  selN = 0;
18:    a = fopen( "exp4.dta", "r" );
19:    fscanf( a, "%lf", &x );
20:    while ( !feof(a) )
21:    {
22:       n = n + 1;              // Αυτά γίνονται για
23:       sum = sum + x;          // όλους τους αριθμούς
24:       if ( 0 < x && x <= 10 ) // 'Ελεγχος - Επιλογή
25:       {
26:          selSum = selSum + x;  // Αυτά γίνονται μόνο για
27:          selN = selN + 1;      // τους επιλεγόμενους αριθμούς
28:       } // if
29:       fscanf( a, "%lf", &x );
30:    } // while
31:    fclose( a );
32:    b = fopen( "report.txt", "w" );
33:    fprintf( b, " Διάβασα %d αριθμούς\n", n );
34:    if ( n > 0 )
35:    {
36:       avrg = sum / n;
37:       fprintf( b, " ΑΘΡΟΙΣΜΑ = %lf   <x> = %lf\n",
38:                   sum, avrg );
39:    }
40:    fprintf( b, " Διάλεξα %d αριθμούς ε (0,10]\n", selN );
41:    if ( selN > 0 )
42:    {
43:       selAvrg = selSum/selN;
44:       fprintf( b, " ΑΘΡΟΙΣΜΑ = %lf  <x> = %lf\n",
45:                   selSum, selAvrg );
46:    }
47:    fclose( b );
48: } // main
```

Ας δούμε τα κρίσιμα σημεία:

- Στη γρ. 1 έχουμε "`#include <cstdio>`" αντί της γνωστής μας "`#include <fstream>`". Θυμίσου ότι το ίδιο είχαμε κάνει και όταν χρησιμοποιούσαμε τις `printf` (§1.12) και `scanf` (§2.12).
- Στις γρ. 5-6 βλέπεις τις δηλώσεις:

```
FILE* a;
FILE* b;
```

Με αυτά τα δύο βέλη θα χειριστούμε τα δύο ρεύματα. Αργότερα θα πάρουμε μια ιδέα για το τι μπορεί να είναι ο τύπος "`FILE`".

- Στη γρ. 18 ανοίγουμε το πρώτο ρεύμα με την "`a = fopen( "exp4.dta", "r" )`" που δίνει τιμή στο βέλος *a*. Η *fopen()* είναι μια συνάρτηση που παίρνει δύο ορίσματα:
  - το όνομα του αρχείου (στην περίπτωσή μας `"exp4.dta"`) και
  - τον τρόπο ανοίγματος (στην περίπτωσή μας `"r"`).

  ανοίγει το ρεύμα και επιστρέφει τιμή τύπου *FILE*`*`. Αν αποτύχει το άνοιγμα επιστρέφει 0 (`NULL`). Στον πίνακα 8.1 μπορείς να δεις πώς γράφεται ένας τρόπος ανοίγματος της C

αντίστοιχος τρόπου ανοίγματος της C++. Στην περίπτωσή μας (`"r"`) έχουμε το ισοδύναμο του "`ios_base::in`".

- Στη γρ. 19 κάνουμε το πρώτο διάβασμα: "`fscanf( a, "%lf", &x )`". Αυτή είναι ίδια με τη `scanf` (§2.12) αλλά έχει ένα επι πλέον όρισμα, στην αρχή, που καθορίζει το ρεύμα (στη `scanf` το ρεύμα είναι το `stdin`). Στη γρ. 29 διαβάζουμε ξανά με τον ίδιο ακριβώς τρόπο.
- Στη γρ. 20, στη συνθήκη της **while**, ελέγχουμε για τέλος αρχείου καλώντας την *feof*. Όπως βλέπεις, της δίνουμε ως όρισμα το βέλος *a*, που καθορίζει το ρεύμα.
- Στη γρ. 31 κλείνουμε τπ ρεύμα με την "`fclose( a )`".
- Στη γρ. 32 ανοίγουμε ρεύμα για να γράψουμε το αρχείο `report.txt` με την "`b = fopen( "report.txt", "w" )`". Ο τρόπος χρήσης του ρεύματος (`"w"`) είναι ισοδύναμος του "`ios_base::out`".
- Στις γρ. 33, 37, 40, 44 γράφουμε στο αρχείου χρησιμοποιώντας την `fprintf`. Αυτή είναι σαν την `printf` αλλά έχει ένα επί πλέον όρισμα στην αρχή που καθορίζει το ρεύμα.
- Θα μπορούσαμε να είχαμε ανοίξει τα ρεύματα με τις δηλώσεις των *a*, *b*; Βεβαίως, έτσι:

```
FILE* a( fopen("exp4.dta", "r") );
FILE* b( fopen("report.txt", "w") );
```

Στη συνέχεια (ξανα)δίνουμε και το πρόγραμμα αντιγραφής αρχείου με χρήση των εργαλείων της C:

```
 0: #include <cstdio>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    FILE* s;
 6:    FILE* t;
 7:    int ch, res( 0 );
 8:
 9:    s = fopen( "text1.dta", "r" );
10:    if ( !s )
11:       fprintf( stderr,
12:                "δεν μπορώ να ανοίξω το αρχείο text1.dta\n" );
13:    else
14:    {
15:       t = fopen( "numdta.txt", "w" );
16:       if ( !t )
17:       {
18:          fclose( s );
19:          fprintf( stderr,
20:                   "δεν μπορώ να δημιουργήσω το numdta.txt\n" );
21:       }
22:       else // ok και τα δύο αρχεία ανοικτά
23:       {
24:          ch = getc( s );
25:          while ( ch != EOF && res != EOF )
26:          {  res = putc( ch, t );  ch = getc( s );  } // while
27:          if ( !feof(s) )
28:             fprintf( stderr,
29:                      "πρόβλημα κατά την αντιγραφή\n" );
30:          fclose( t );
31:          fclose( s );
32:       } // if ( t.fail
33:    } // if ( s.fail
34: } // main
```

Εδώ, τα καινούρια πράγματα είναι στις γρ. 24, 25, 26:

- Η συνάρτηση *getc()* παίρνει ένα όρισμα –το βέλος που καθορίζει το ρεύμα (που διαβάζουμε)– και μας επιστρέφει τον επόμενο χαρακτήρα από το αρχείο αλλά σε τύπο `int`. Αν υπάρξει πρόβλημα επιστρέφει `EOF` (-1).

Μέθοδος / Τελεστής	Κλάση	Τι κάνει, πού την είδαμε
`>>`	`ifstream`	Φέρνει τιμή από το αρχείο σε μεταβλητή. §8.3, 2.3, 4.5, 4.6
`<<`	`ofstream`	Γράφει τιμές στο αρχείο. §8.4, 1.2, 1.11, 4.4.1
`clear`	`ifstream, ofstream`	Ανατάσσει το ρεύμα μετά από *eof* ή άλλη αποτυχία κάποιας λειτουργίας. §8.3.2
`close`	`ifstream, ofstream`	Αποσυνδέει ένα ρεύμα από το αρχείο. §8.3, 8.4
`endl`	`ofstream`	Γράφει τέλος γραμμής και αντιγράφει τον ενταμιευτή στο αρχείο. §8.4
`eof`	`ifstream`	Μας πληροφορεί αν φτάσαμε στο τέλος του αρχείου. §8.3, 8.3.1, 8.11
`fail`	`ifstream, ofstream`	Επιστρέφει `true` αν η προηγούμενη λειτουργία στο ρεύμα απέτυχε. §8.11
`get`	`ifstream`	Φέρνει ένα χαρακτήρα από το αρχείο στο όρισμά της. §8.9, 8.12, 4.5
`open`	`ifstream, ofstream`	Συνδέει το ρεύμα με το αρχείο. §8.1, 8.3, 8.4
`is_open`	`ifstream, ofstream`	Επιστρέφει τιμή `true` αν το ρεύμα είναι ανοικτό, αλλιώς `false`, §8.11.
`peek`	`ifstream`	Δίνει το χαρακτήρα που θα διαβάσει η επόμενη *get()*. §8.9
`put`	`ofstream`	Γράφει το όρισμά της στο αρχείο. §8.9
`seekg(0)`	`ifstream`	Το ρεύμα ετοιμάζεται να διαβάσει από την αρχή του αρχείου. §8.6
`seekp(0)`	`ofstream`	Το ρεύμα ετοιμάζεται να γράψει στην αρχή του αρχείου. §8.12

Πίν. 8-2 Οι μέθοδοι για τη διαχείριση αρχείων με ρεύματα που μάθαμε. Όλες οι μέθοδοι υπάρχουν και στην κλάση `fstream`. Οι μέθοδοι της `ifstream` υπάρχουν και στην `istream` (τύπος του `cin`). Οι μέθοδοι της `ofstream` υπάρχουν και στην `ostream` (τύπος των `cout`, `cerr`, `clog`).

- Η *putc()* παίρνει δύο ορίσματα: τον χαρακτήρα που θέλουμε να γράψουμε στο αρχείο και το βέλος για το ρεύμα. Αν γράψει τον χαρακτήρα τον επιστρέφει και ως τιμή. Αν αποτύχει επιστρέφει `EOF`. Όπως βλέπεις, στη *res* κρατούμε την τιμή που επιστρέφει κάθε φορά η *putc()*. Όσο δεν παίρνουμε `EOF` ούτε στη *res* ούτε στη *ch* ("`while (ch != EOF && res != EOF)`") συνεχίζουμε τη διαδικασία αντιγραφής.

8.14 Σύνοψη

Στο κεφάλαιο αυτό μάθαμε πώς να χειριζόμαστε αρχεία-κείμενα (text) είτε έχουν αριθμητικά δεδομένα είτε τα θεωρούμε ως αρχεία χαρακτήρων. Τα εργαλεία μας είναι ρεύματα των κλάσεων *ifstream*, *ofstream* και *fstream* και οι μέθοδοί τους που βλέπεις στον Πίν. 8-2.

Θα επαναλάβουμε ακόμη και μερικές βασικές αρχές που μάθαμε στο κεφάλαιο αυτό.

- ***Για να διαβάσουμε (ή να γράψουμε) το ν-οστό στοιχείο ενός σειριακού αρχείου πρέπει να περάσουμε κατ' ανάγκη τις ν −1 προηγούμενες τιμές.***
- ***Η επεξεργασία κάθε σειριακού αρχείου ξεκινάει πάντα από την αρχή του.***
- ***Πριν επεξεργαστούμε μια τιμή που (υποτίθεται ότι) διαβάσαμε από ένα αρχείο ελέγχουμε την eof() ή τη fail() για να βεβαιωθούμε ότι η ανάγνωση έγινε κανονικώς.***

Λύσε οπωσδήποτε τις Ασκ. 8-5, 8-6 (και αργότερα την 9-13) και δες προσεκτικά τις λύσεις που προτείνουμε. Αναφέρονται στην *ενημέρωση* (μεταβολή, εισαγωγή, διαγραφή στοιχείων) σειριακού αρχείου με βάση την αρχή:

- ***Για να ενημερώσουμε ένα σειριακό αρχείο το αντιγράφουμε σε ένα άλλο και κατά την αντιγραφή κάνουμε τις αλλαγές που θέλουμε.***

Ασκήσεις

Α Ομάδα

8-1 Γράψε ένα πρόγραμμα που θα αντιγράφει ένα αρχείο text στην οθόνη. Αριστερά από κάθε γραμμή θα γράφεται ο αριθμός της, ξεκινώντας από 1.

8-2 Σε ένα αρχείο-κείμενο, με όνομα στο δίσκο real.txt, έχουμε πραγματικούς αριθμούς. Γράψε ένα πρόγραμμα που θα μετράει:

- πόσοι αριθμοί μεταξύ 4.5 και 5.5 υπάρχουν στο αρχείο,
- πόσες φορές υπάρχει η τιμή 5.0

Ακόμα, θα δίνει τα ποσοστά των παραπάνω αριθμών σε σχέση με τον συνολικό αριθμό τιμών του αρχείου.

8-3 Σε δύο αρχεία text, με ονόματα 'mydata.txt', 'moredata.txt' και το ίδιο πλήθος γραμμών, υπάρχουν αριθμοί, ένας σε κάθε γραμμή. Γράψε πρόγραμμα που θα τα διαβάζει και θα δημιουργεί ένα νέο αρχείο text, με όνομα "comb.txt", που θα έχει το ίδιο πλήθος γραμμών με κάθε ένα από τα αρχικά και στη ν-οστή γραμμή του θα έχει τέσσερις αριθμούς:

- αυτόν που υπάρχει στην ν-οστή γραμμή του πρώτου αρχείου,
- αυτόν που υπάρχει στην ν-οστή γραμμή του δεύτερου αρχείου,
- το άθροισμα των δύο προηγούμενων,
- το γινόμενο των δύο προηγούμενων.

8-4 Σε ένα αρχείο text, με όνομα protmet.txt, υπάρχουν πρότυπες μετρήσεις από τον έλεγχο μιας συσκευής –ένας αριθμός σε κάθε γραμμή. Το αρχείο testmet.txt είναι ίδιο και έχει τον ίδιο αριθμό γραμμών με το πρώτο αλλά όχι και τους ίδιους αριθμούς· περιέχει δοκιμαστικές μετρήσεις που κάναμε στη συσκευή για να δούμε αν είναι εντάξει. Έστω α ο αριθμός που υπάρχει στην k γραμμή του πρώτου αρχείου και β αυτός που υπάρχει στην k γραμμή του δεύτερου αρχείου. Η συσκευή μας είναι εντάξει αν

για όλα τα k έχουμε: $0.95\alpha \le \beta \le 1.05\alpha$

δηλαδή, αν υπάρχουν αποκλίσεις που δεν υπερβαίνουν το 5%.

Γράψε πρόγραμμα που θα διαβάζει τα δυο αρχεία και

- για κάθε γραμμή (k) που βρίσκει απόκλιση μεγαλύτερη από 5% θα μας τυπώνει το k, το α και το β,
- αν δεν βρει απόκλιση μεγαλύτερη από 5% θα μας τυπώνει στο τέλος το μήνυμα "ΣΥΣΚΕΥΗ ΕΝΤΑΞΕΙ".

Β Ομάδα

8-5 (Ενημέρωση – μεταβολή στοχείων) Σε ένα αρχείο-κείμενο, με όνομα στο δίσκο idid.txt, έχουμε σε κάθε γραμμή δύο πραγματικούς αριθμούς που παριστάνουν μια τάση (ο πρώτος) και ένα ρεύμα (ο δεύτερος), όπως μετρήθηκαν στο εργαστήριο. Αλλά οι τιμές του ρεύματος διαβάστηκαν λαθεμένα στο αμπερόμετρο. Στις πρώτες πέντε (5) γραμμές η τιμή που υπάρχει στο αρχείο είναι 1000πλάσια της πραγματικής και στις επόμενες 11 είναι 10πλάσια της πραγματικής. Οι υπόλοιπες γραμμές είναι σωστές.

Γράψε λοιπόν μια συνάρτηση που θα παίρνει τη θέση της γραμμής (k) και το ρεύμα (i) και θα μας δίνει το σωστό ρεύμα:

```
double correctI( int k, double i )
```

Θέλουμε να αλλάξουμε κάθε τιμή ρεύματος (*i*) του αρχείου σε *correctI*(*k*, *i*), όπου *k* (1η, 2η, 3η κλπ) η γραμμή του αρχείου όπου βρίσκεται η *i*. Αυτό θα το κάνεις με ένα πρόγραμμα που θα αντιγράφει μια προς μια τις γραμμές του idid.txt σε ένα άλλο, με το όνομα midid.txt, αλλά, κάθε φορά θα αλλάζει την τιμή ρεύματος (*i*) σε *correctI*(*k*, *i*) με τη συνάρτηση που έγραψες όπως είπαμε παραπάνω.

8-6 (Ενημέρωση – διαγραφή στοιχείων) Έστω τώρα ότι θέλουμε να σβήσουμε από το αρχείο idid.όλες τις γραμμές με λαθεμένα στοιχεία, δηλαδή τις πρώτες δεκαέξη (16) γραμμές.

Και πάλι, θα πρέπει να γράψεις ένα πρόγραμμα που να δημιουργεί ένα νέο αρχείο με όνομα dідid.txt, όπου θα αντιγράψει μόνον τις γραμμές με σωστά στοιχεία (από τη 17η και μετά). Φυσικά, τα στοιχεία που θα διαγράψεις δεν θα πάνε στα σκουπίδια: θα αντιγραφούν σε ένα άλλο αρχείο με όνομα xidid.txt.

8-7 Ένα αρχείο text, με όνομα στο δίσκο misthoi.txt, είναι γραμμένο ως εξής: Σε κάθε γραμμή έχει τρεις αριθμούς. Ο πρώτος είναι ο βασικός μισθός ενός υπαλλήλου, ο δεύτερος ο αριθμός χρόνων υπηρεσίας του ίδιου υπαλλήλου και ο τρίτος ο αριθμός των παιδιών του. Το πλήθος των στοιχείων είναι άγνωστο. Το αρχείο θα χρησιμοποιηθεί για τη μισθοδοσία των υπαλλήλων ενός οργανισμού.

Το περιεχόμενο του αρχείου δεν έχει ελεγχθεί. Θα γράψεις ένα πρόγραμμα που θα ελέγχει τα εξής:

- αν 800 ≤ βασικός μισθός ≤ 3500,
- αν 0 ≤ χρόνια υπηρεσίας ≤ 35,
- αν 0 ≤ αριθμός παιδιών ≤ 12.

Για κάθε λάθος που θα βρίσκει, θα πρέπει να τυπώνει τον αριθμό του υπαλλήλου, δηλαδή τη σειρά που έχουν τα στοιχεία του στο αρχείο και τη λαθεμένη τιμή. Π.χ.:

```
 7ΟΣ ΥΠΑΛΛΗΛΟΣ. ΜΙΣΘΟΣ: 35650
 7ΟΣ ΥΠΑΛΛΗΛΟΣ. ΑΡΙΘΜΟΣ ΠΑΙΔΙΩΝ: -4
10ΟΣ ΥΠΑΛΛΗΛΟΣ. ΧΡΟΝΙΑ ΥΠΗΡΕΣΙΑΣ: 51
23ΟΣ ΥΠΑΛΛΗΛΟΣ. ΜΙΣΘΟΣ: 955000
```

8-8 Με βάση τα στοιχεία του διορθωμένου αρχείου, dmisthoi.txt, της προηγούμενης άσκησης, υπολογίζονται οι αποδοχές των υπαλλήλων ως εξής:

- για κάθε χρόνο υπηρεσίας ο μισθός προσαυξάνεται κατά 2% πάνω στο βασικό,
- αν ο εργαζόμενος έχει μέχρι τρία (3) παιδιά ο μισθός προσαυξάνεται κατά 30 € για κάθε παιδί, αλλιώς, αν έχει περισσότερα από τέσσερα (4) ο μισθός προσαυξάνεται κατά 50 € για κάθε παιδί.

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει και θα τυπώνει την κατάσταση αποδοχών των υπαλλήλων.

Γ Ομάδα

8-9 Δίνεται ένα αρχείο text. Γράψε ένα πρόγραμμα που θα αντιγράφει το αρχείο σε ένα άλλο, αφού πρώτα πετάξει τα περιττά κενά του πρώτου. Δηλαδή, όπου υπάρχουν στο παλιό ένα ή περισσότερα συνεχόμενα κενά, στο νέο θα υπάρχει μόνο ένα.

8-10 Γράψε πρόγραμμα που θα διαβάζει ένα αρχείο κείμενο που περιέχει πρόγραμμα C++ και θα το αντιγράφει στην οθόνη με αρίθμηση των γραμμών. Π.χ. θα μας δείχνει:

```
1: #include <iostream>
2: #include <math>
3:
4: int main()
5: {
```

```
6:    int a[100], b[25];
       . . .
```

8-11 Γράψε πρόγραμμα που θα διαβάζει ένα αρχείο κείμενο που περιέχει πρόγραμμα C++ και θα το αντιγράφει σε δύο αρχεία ως εξής:

α) Το ένα θα έχει μόνον τις γραμμές που περιέχουν μόνον σχόλια, δηλαδή γραμμές που ξεκινούν με `//`.

β) Το άλλο θα έχει όλες τις υπόλοιπες γραμμές.

8-12 Τεχνικοί, που σχεδίασαν και κατασκεύασαν μια συσκευή, πιστεύουν ότι δύο βασικά της μεγέθη q και u συνδεόνται με τον απλό νόμο: $q = \eta\mu u$. Μια άλλη ομάδα τεχνικών πιστεύει ότι η εξάρτηση είναι πιο πολύπλοκη: $q = g(u)$, όπου g συνάρτηση περιοδική, με περίοδο 2π, που στο $[-\pi,\pi]$ ορίζεται ως εξής:

$$g(u) = \begin{cases} \frac{4}{\pi^2}u(u+\pi), & -\pi \le u \le 0 \\ -\frac{4}{\pi^2}u(u-\pi), & 0 \le u \le \pi \end{cases}$$

Γράψε συνάρτηση που να υλοποιεί τη g σε C++.

Σε ένα αρχείο text, με όνομα στο δίσκο irnapr.txt, υπάρχουν στοιχεία μετρήσεων της συσκευής. Σε κάθε γραμμή του αρχείου υπάρχουν δύο πραγματικοί αριθμοί: ο πρώτος αντιπροσωπεύει τιμή από μέτρηση της u, ενώ ο δεύτερος είναι η αντίστοιχη τιμή της q. Θέλουμε, με επεξεργασία αυτών των τιμών, να δούμε ποιο από τα δύο μοντέλα είναι καλύτερο.

Ας πούμε λοιπόν ότι διαβάζουμε μια γραμμή του αρχείου και αποθηκεύουμε τις τιμές που διαβάσαμε στις u_k και q_k. Υπολογίζουμε τα $\eta\mu u_k$ και $g(u_k)$ και θα λέμε:

- ότι «είναι καλύτερο το μοντέλο της g» αν $|g(u_k) - q_k| < |\eta\mu u_k - q_k|$·

αλλιώς θα λέμε

- ότι «καλύτερο είναι το μοντέλο του ημιτόνου».

Θέλουμε ένα πρόγραμμα που θα διαβάζει ολόκληρο το αρχείο και θα μας λέει στο τέλος:

- πόσες φορές (για πόσες γραμμές του αρχείου) ήταν καλύτερο το μοντέλο του ημιτόνου και πόσες φορές το μοντέλο της g,
- τις μέσες τιμές των $|g(u_k)-q_k| = \frac{1}{N}\sum_{k=1}^{N}|g(u_k)-q_k|$ και $|\eta\mu u_k-q_k| = \frac{1}{N}\sum_{k=1}^{N}|\eta\mu u_k-q_k|$ όπου N το πλήθος των γραμμών του αρχείου.

Ακόμη, το πρόγραμμα θα αντιγράφει το αρχείο σε δύο άλλα αρχεία text με ονόματα στο δίσκο girnapr.txt, sirnapr.txt, ως εξής:

- Αν για μια γραμμή είναι καλύτερο το μοντέλο της g τότε η γραμμή θα αντιγράφεται «εμπλουτισμένη» στο girnapr.txt. Για την ακρίβεια θα γράφονται: u_k, q_k, $g(u_k)$, $|g(u_k) - q_k|$.
- Αν για μια γραμμή είναι καλύτερο το μοντέλο του ημιτόνου τότε μια «εμπλουτισμένη» γραμμή με τα u_k, q_k, $\eta\mu u_k$, $|\eta\mu u_k - q_k|$ θα γράφεται στο αρχείο sirnapr.txt.

8-13 Για το παρόν πρόβλημα θεωρούμε ως λέξη μια ακολουθία χαρακτήρων που είναι γράμματα του λατινικού αλφαβήτου. Π.χ. στο κείμενο

```
if <x, f, x'> is a state transition
```

υπάρχουν 8 λέξεις με μήκη 2, 1, 1, 1, 2, 1, 5, 10 αντιστοίχως.

Μας δίνεται το αρχείο-κείμενο alturing.txt. Γράψε πρόγραμμα που θα το διαβάζει και θα μας λέει α) το πλήθος των λέξεων που έχει β) το μέγιστο μήκος λέξης που συνάντησε.

κεφάλαιο

9

Πίνακες Ι

Ο στόχος μας σε αυτό το κεφάλαιο:

Συχνά έχουμε πολλές μεταβλητές με τις ίδιες ιδιότητες που πρέπει να υποστούν την ίδια επεξεργασία. Θα μάθεις πώς να τις οργανώνεις σε *πίνακες* ώστε να τις χειρίζεσαι με τις γνωστές εντολές επανάληψης.

Προσδοκώμενα αποτελέσματα:

Ο πίνακας είναι ένα πολύ καλό εργαλείο για πάρα πολλές χρήσεις. Εδώ θα δεις μερικές επεξεργασίες πινάκων που θα σου είναι χρήσιμες πολύ συχνά.

Έννοιες κλειδιά:

- *πίνακας*
- *στοιχείο πίνακα, δείκτης*
- *αναζήτηση τιμής σε πίνακα*
- *ταξινόμηση πίνακα*
- *συγχώνευση πινάκων*
- *αποδοτικότητα αλγόριθμου*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει 100 πραγματικούς αριθμούς x_1, x_2, ..., x_{100} και

α) θα υπολογίζει και θα τυπώνει τη Μέση Αριθμητική Τιμή τους <*x*>,

β) θα υπολογίζει και θα τυπώνει τις τιμές της συνάρτησης: $e^{\frac{<x>-x}{<x>}}$ για τους αριθμούς αυτούς. Δηλαδή τα $y_k = e^{\frac{<x>-x_k}{<x>}}$.

Ε, αυτό το λύσαμε στο προηγούμενο κεφάλαιο και μάλιστα όχι για 100 αλλά για όσους αριθμούς και να έχουμε! Γράφουμε σε ένα αρχείο τους αριθμούς όταν πληκτρολογούνται και όταν έχουμε υπολογίσει την <*x*>, διαβάζουμε το αρχείο από την αρχή.

Πόσο καλή είναι αυτή η λύση; Όχι και πολύ καλή. Δεν είναι παραδεκτό να στέλνουμε 100 αριθμούς σε βοηθητική μνήμη και να τους ξαναδιαβάζουμε για να χρησιμοποιηθούν δεύτερη φορά από το ίδιο πρόγραμμα. Πρέπει να τους κρατήσουμε στην κύρια μνήμη όσο τους χρειαζόμαστε.

Αλλά, σύμφωνα με όσα ξέρουμε, θα πρέπει δηλώσουμε 100 μεταβλητές: *x*1, *x*2, ...,*x*100 τύπου `double` και να γράψουμε 100 εντολές:

```
cin >> x1; cin >> x2; ... ; cin >> x100;
```

για να τους διαβάσουμε.

Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει έναν πιο άνετο τρόπο για να λύσουμε το πρόβλημά μας. Το εργαλείο που θα χρησιμοποιήσουμε λέγεται **πίνακας** και θα το γνωρίσουμε στις παραγράφους που ακολουθούν.

9.1 Πίνακες Στοιχείων

Η έννοια του **πίνακα** (array, matrix) είναι γνωστή από τα μαθηματικά. Στη C++ ο πίνακας δεδομένων αποτελείται από έναν αριθμό *ομοειδών στοιχείων*, δηλ. από *στοιχεία του ίδιου τύπου*. Ο τύπος αυτός καθορίζεται όταν δηλώνουμε τη μεταβλητή - πίνακα.

Τα ομοειδή στοιχεία, που αποτελούν τον πίνακα, έχουν πάντα ένα κοινό όνομα, π.χ. τα pinA, pinB και flNm στο παρακάτω παράδειγμα δήλωσης πινάκων:

```
int    pinA [ 20 ];
double pinB [ 50 ];
char   flNm [ 120 ];
```

Η πρώτη δήλωση του παραδείγματος καθορίζει ότι ο πίνακας pinA αποτελείται από 20 στοιχεία που το καθένα έχει τα χαρακτηριστικά μιας μεταβλητής τύπου `int`, η δεύτερη γραμμή δηλώνει ότι ο πίνακας pinB αποτελείται από 50 στοιχεία που έχουν χαρακτηριστικά μεταβλητής τύπου `double` και η τρίτη ότι ο πίνακας flNm αποτελείται από 120 στοιχεία τύπου `char`.

Σχηματικά μπορούμε να βλέπουμε, π.χ. τον πίνακα pinA, σαν ένα μονοδιάστατο πίνακα που αποτελείται από μια γραμμή και 20 στήλες ή από μια στήλη και 20 γραμμές. Το Σχ. 9-1 δείχνει την πρώτη περίπτωση δομής.

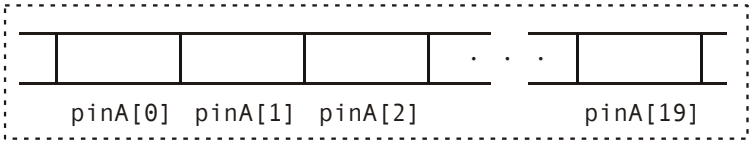
Τα **στοιχεία** (elements) ή οι **συνιστώσες** (components) του πίνακα σημειώνονται πάντα με το κοινό όνομα της μεταβλητής-πίνακα που συνοδεύεται από τον λεγόμενο **δείκτη** (index).

- ***Αν ο πίνακας έχει δηλωθεί με n στοιχεία, οι επιτρεπτές τιμές του δείκτη είναι όλοι οι φυσικοί από 0 μέχρι n – 1.***

Ο δείκτης του στοιχείου πίνακα δίνεται πάντα μέσα σε αγκύλες, όπως δείχνουν τα παρακάτω παραδείγματα:

```
pinA[1],  pinA[5],  pinA[15]
pinB[2],  pinB[10],  pinB[49]
```

Έτσι, τα συμβολικά ονόματα της πρώτης γραμμής καθορίζουν το δεύτερο, το έκτο και το δέκατο έκτο στοιχείο του πίνακα pinA. Ενώ τα ονόματα της δεύτερης γραμμής καθορίζουν το τρίτο, το ενδέκατο και το τελευταίο στοιχείο του πίνακα pinB. Τέλος το pinA`[`*k*`]` καθορί-



Σχ. 9-1 Σχηματική παράσταση πίνακα και των στοιχείων του.

ζει γενικώς το (*k*+1)-οστό στοιχείο του πίνακα pinA, όπου *k* = 0, 1,... 19. Αντιστοίχως το pinB`[`*k*`]` καθορίζει το (*k*+1)-οστό στοιχείο του πίνακα pinB, όπου *k* = 0, 1,... 49.

Το γενικό συντακτικό δήλωσης του μονοδιάστατου πίνακα μπορεί να δοθεί ως εξής:

τύπος, *αναγνωριστικό*, "`[`", *πλήθος*, "`]`"

- Ο *τύπος*, που λέγεται και **τύπος συνιστωσών** (component type), μπορεί να είναι οποιοσδήποτε τύπος, π.χ. `int`, `double`, `char`, `bool`, απαριθμητός κλπ.
- Το *αναγνωριστικό* είναι σαν και αυτά των μεταβλητών.
- Το *πλήθος* είναι μια παράσταση που μπορεί να περιέχει σταθερές, μεταβλητές που έχουν ήδη τιμή και συναρτήσεις, αν φυσικά η δήλωσή μας βρίσκεται μέσα στην εμβέλειά τους. Η τιμή της πρέπει να είναι ακέραιου τύπου και μεγαλύτερη από 0. Το *πλήθος* μπορεί να παραλείπεται.

Κοίταξε τα παρακάτω παραδείγματα:

```
const int N( 50 ), N1( 63 ), N2( 114 );

typedef int  PinAk[ N+1 ];
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };

PinAk    p, q, r;
int      l[ N+1 ], m[ N2-N1+1 ];
bool     signal[ (N1+1)/4 ];
ifstream keimena[ 4 ];
DecDigit bv[ N/2+1 ];
PinAk    mat[ 11 ];
```

Κατ' αρχάς ορίζουμε έναν τύπο, τον *PinAk*· κάθε αντικείμενο αυτού του τύπου, όπως οι p, q, r που δηλώνουμε παρακάτω, είναι ένας πίνακας με 51 (= *N* + 1) στοιχεία τύπου `int`, που αριθμούνται από 0 μέχρι 50, π.χ. p`[`0`]`, p`[`1`]`, …, p`[`50`]`.

Στη συνέχεια αντιγράφουμε τον ορισμό του τύπου *DecDigit* από το Κεφ. 4 και παρακάτω δηλώνουμε τον πίνακα bv που έχει 26 (= *N*/2 + 1) στοιχεία τύπου *DecDigit*.

Ο πίνακας `l` έχει 51 στοιχεία τύπου `int`· θα μπορούσαμε να είχαμε δηλώσει "`PinAk l`". Ο πίνακας m έχει 52 στοιχεία τύπου `int`.

Ο keimena είναι ένας πίνακας που κάθε στοιχείο του είναι ένα ρεύμα τύπου *ifstream*.

Τέλος, στην τελευταία δήλωση, ο mat έχει 11 στοιχεία που το καθένα τους είναι ένας πίνακας τύπου *PinAk*· είναι ένας πίνακας πινάκων ή δισδιάστατος πίνακας. Με τέτοιους πίνακες θα ασχοληθούμε αργότερα.

Μαζί με τη δήλωση ενός πίνακα μπορείς να δώσεις και αρχικές τιμές στα στοιχεία του. Π.χ.:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Στην περίπτωση αυτή δεν χρειάζεται να δηλώσεις και το πλήθος (ο μεταγλωττιστής ξέρει να μετράει). Θα μπορούσαμε να γράψουμε:

```
int monthLength[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

Αν δηλώσεις το πλήθος των στοιχείων και στη λίστα αρχικών τιμών βάλεις λιγότερες τιμές τότε οι υπόλοιπες θεωρούνται μηδέν. Π.χ. η

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31 };
```

είναι ισοδύναμη με την:

```
int monthLength[ 12 ] = { 31,28,31,30,31,30,31,31, 0, 0, 0, 0 };
```

Όπως καταλαβαίνεις, αν θέλεις να βάλεις αρχική τιμή "`0`" σε όλα τα στοιχεία ενός πίνακα αρκεί να γράψεις ένα "`0`" ανάμεσα στα άγκιστρα:

```
double x[ 100 ] = { 0.0 };
```

Ωραία λοιπόν, δηλώσαμε τον πίνακα. Τώρα τι κάνουμε; Με ολόκληρο τον πίνακα δεν μπορούμε να κάνουμε και πολλά πράγματα, αλλά:

- ***Κάθε συνιστώσα ενός πίνακα έχει όλα τα χαρακτηριστικά μιας μεταβλητής του τύπου συνιστωσών και μπορείς να τη χειριστείς όπως και κάθε μεταβλητή αυτού του τύπου.***

Έτσι, π.χ. το `l[17]` είναι μεταβλητή τύπου `int`. Μπορούμε λοιπόν να της δώσουμε τιμή με μια εντολή εκχώρησης:

```
l[17] = 3215;
```

ή να τη διαβάσουμε από το πληκτρολόγιο:

```
cin >> l[17];
```

Αν η `l[17]` έχει τιμή μπορούμε να τη χρησιμοποιήσουμε σε παραστάσεις:

```
x = l[17] / 5 + 100;
```

να γράψουμε την τιμή της στην οθόνη ή σε κάποιο αρχείο:

```
cout << l[17] << endl;
```

Το signal`[11]` είναι μεταβλητή τύπου `bool` –μπορεί να πάρει τιμές `true` ή `false`, π.χ.:

```
signal[11] = ( l[17] >= x );
if ( signal[11] && x > 1000 )  { ...
```

Όπως είδαμε και πιο πάνω, με τον δείκτη ξεχωρίζουμε τις συνιστώσες ενός πίνακα. Ο δείκτης είναι, γενικά, μια παράσταση. Η τιμή αυτής της παράστασης θα πρέπει να είναι μη αρνητική και μικρότερη από το πλήθος στοιχείων που έχουμε δηλώσει. Μετά τη δήλωση:

```
int  metr[ 26 ];
```

μπορείς να δώσεις:

```
metr[0] = -16;
```

αλλά, δεν μπορείς να δώσεις:

```
metr[27] = 24;
```

ούτε

```
metr[-5] = 33;
```

Αυτό είναι λάθος που, δυστυχώς, *δεν θα το δείξει* ο μεταγλωττιστής. Ακόμη, αν *c* `==` 22, μπορείς να δώσεις:

```
metr[c+1] = c - 17;
```

αλλά, αν *c* `==` 25, η παραπάνω εντολή είναι λάθος: `metr[26]` *δεν υπάρχει!* Αυτό το λάθος φυσικά δεν μπορεί να το εντοπίσει ο μεταγλωττιστής, αλλά, δυστυχώς, μπορεί να μη φανεί ούτε κατά την εκτέλεση. Θα οδηγήσει, πιθανότατα, σε παράλογα αποτελέσματα (μπορεί και να «παγώσει» ο υπολογιστής σου).

- ***Για πίνακα με n στοιχεία είναι υποχρέωση του προγραμματιστή να περιορίζει την τιμή του δείκτη στις επιτρεπτές τιμές (από 0 μέχρι n – 1).***

Όπως φαίνεται από τα παραπάνω παραδείγματα, ο δείκτης του πίνακα μπορεί να είναι, γενικά, μια παράσταση. Έχουμε λοιπόν τη δυνατότητα πρόσβασης στα διάφορα στοιχεία ενός πίνακα, δίνοντας την κατάλληλη τιμή στο δείκτη. Αν θέλουμε να χειριστούμε όλα τα στοιχεία ενός πίνακα, μπορούμε να το πετύχουμε με μια επανάληψη όπου η μεταβαλλόμενη τιμή του δείκτη μας δίνει το ένα στοιχείο μετά το άλλο.

Η C++ δεν μας επιτρέπει διαχείριση ολόκληρου του πίνακα. Για παράδειγμα η εκχώρηση πρέπει να γίνεται στοιχείο προς στοιχείο. Ας πούμε ότι έχουμε δηλώσει:

```
int  a[5], b[5];
```

και κάποτε θέλουμε να εκχωρήσουμε την τιμή που έχει εκείνη τη στιγμή ο a στον b. *Δεν επιτρέπεται να δώσουμε:* `b = a`· θα πρέπει να δώσουμε την εντολή:

```
   for ( k = 0; k <= 4; k = k+1 )  b[k] = a[k];
```

Τώρα, μπορούμε να ξαναγυρίσουμε στο πρόβλημα της εισαγωγικής παραγράφου και να δούμε μια πιο όμορφη λύση.

Παράδειγμα

Ας έλθουμε τώρα στο πρόβλημα που δώσαμε στην εισαγωγή. Στο πρόγραμμα-λύση που γράψαμε στο προηγούμενο κεφάλαιο η *x* ήταν μια απλή μεταβλητή. Κάθε φορά που διαβάζαμε μια νέα τιμή, η προηγούμενη τιμή της *x* χάνονταν, αφού φυσικά την είχαμε φυλάξει στο αρχείο.

Τώρα θα δηλώσουμε τη x ως:

```
   double   x[ N ];
```

Σε κάθε εκτέλεση της περιοχής της (πρώτης) **for** θα δίνουμε τιμή και σε διαφορετικό στοιχείο του *x*. Όταν υπολογίσουμε τη <*x*>, οι 100 αριθμοί που αποθηκεύτηκαν στον x θα είναι στη διάθεσή μας για δεύτερη χρήση:

```
// πρόγραμμα: Μέση Τιμή 1+
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
   const int N( 100 );// το πλήθος των στοιχείων
   double x[ N ];     // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;        // Το μερικό (τρέχον) άθροισμα.
                      // Στο τέλος έχει το ολικό άθροισμα.
   double avrg;       // Μέση Αριθμητική Τιμή των x (<x>)
   int m;
   double y;

   sum = 0;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x[m];
   // πρόσθεσέ την στη sum
      sum = sum + x[m];
   } // for
   avrg = sum / N;
   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 3 );
   cout << " ΑΘΡΟΙΣΜΑ = " << sum
        << "  <x> = " << avrg << endl;
   for ( m = 0; m <= N-1; m = m+1 )
   {
      y = exp( (avrg - x[m])/avrg );
      cout << " x[";
      cout.width(3);  cout << m << "] = ";
      cout.width(6);  cout << x[m] << "   y[";
      cout.width(3);  cout << m << "] = " ;
      cout.width(6);  cout << y << endl;
   } // for
} // main
```

Το παραπάνω πρόγραμμα έχει δύο **for**, όπου η μεταβλητή *m* παίζει διπλό ρόλο. Χρησιμοποιήθηκε:

α) ως μεταβλητή ελέγχου της **for** και

β) ως δείκτης πίνακα (που διατρέχει τις τιμές: 0 . . *N*-1).

Αυτή η λύση είναι σαφώς καλύτερη από αυτήν της προηγούμενης παραγράφου. Αυτό το πρόγραμμα είναι φανερά πιο γρήγορο από το πρώτο και αυτό διότι αποφεύγει τη χρονοβόρα διαδικασία γραψίματος/διαβάσματος από τη βοηθητική μνήμη.

Στη σύνθεση του προγράμματος, μπορεί κανείς να παρασυρθεί από τον τύπο $y_k = e^{\frac{<x>-x_k}{<x>}}$ και να δηλώσει το y ως πίνακα. «Το x είναι πίνακας, βλέπουμε και δείκτη στο y... Βάλε έναν πίνακα, να τελειώνουμε!» Αυτό που μας ζητάνε, όμως, είναι να υπολογίσουμε και να τυπώσουμε τα y_k. Μετά δεν μας χρειάζονται πια. Το y θα μπορεί να είναι απλή μεταβλητή τύπου `double`. Άλλωστε, «ζωγραφίζουμε» και τα αποτελέσματά μας έτσι ώστε να βγάζουν δείκτες, για να ευχαριστηθεί και ο χρήστης:

```
ΑΘΡΟΙΣΜΑ = 1772.217  <x> = 17.722
x[  0] = -2.217   y[  0] =  3.081
x[  1] = 58.900   y[  1] =  0.098
x[  2] = 35.850   y[  2] =  0.360
x[  3] = -0.879   y[  3] =  2.857
     . . .
```

Το κράτημα 100 θέσεων `double` στην κύρια μνήμη θα ήταν καθαρή σπατάλη.

Παρατήρηση: ▶

Γλυτώσαμε λοιπόν από το αρχείο! Έτσι, νομίζεις. Πληκτρολογείς 100 αριθμούς για να δοκιμάσεις το πρόγραμμα και βλέπεις ότι έχει λάθος. Μετά, στη δεύτερη δοκιμή, τι κάνεις; Τους ξαναπληκτρολογείς; Δεν είμαστε καλά! Διόρθωσε παιδί μου το πρόγραμμα να τους φυλάξει σε ένα αρχείο! ◀

Αυτό που λέμε στην παρατήρηση θα το πούμε ως γενική συμβουλή:

- ***Όταν πληκτρολογούμε δεδομένα, που πιθανότατα θα ξαναχρειαστούμε, τα φυλάγουμε οπωσδήποτε σε αρχείο.***

9.2 Συνηθισμένες Δουλειές με Πίνακες

Ας δούμε τώρα μερικές πολύ συνηθισμένες δουλειές που γίνονται με πίνακες. Για τα παρακάτω κομμάτια προγράμματος υποθέτουμε ότι έχουμε δηλώσει:

```
const int N(...);
double x[N];
```

Φυσικά, ο τύπος μπορεί να μην είναι `double`, αλλά `int`, `char` κλπ.

Πριν προχωρήσουμε να τονίσουμε το εξής: Κατ' αρχήν, μια μεταβλητή που προορίζεται να παίζει ρόλο δείκτη θα πρέπει να δηλώνεται με τύπο `unsigned int`. Όπως θα δεις όμως, συνήθως, θα τη δηλώνουμε με τύπο `int`. Από αμέλεια; Και από αμέλεια, αλλά όπως θα δεις σε κάποια από τα παραδείγματα, μερικές φορές δίνουμε σε τέτοιες μεταβλητές κάποια αρνητική τιμή=φρουρό.

9.2.1 Εισαγωγή Στοιχείων

Στο παράδειγμα της προηγούμενης παραγράφου είδαμε πώς διαβάζουμε τις τιμές των στοιχείων ενός πίνακα από το πληκτρολόγιο. Το ξαναγράφουμε λίγο αλλαγμένο και χωρίς την άθροιση:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   cout << "Δώσε το " << m << "ο στοιχείο: ";
   cin >> x[m];
} // for
```

Με αυτόν τον τρόπο το πρόγραμμα παίρνει τις τιμές όλων των στοιχείων με τη σειρά, καθοδηγώντας το χρήστη. Παράδειγμα εκτέλεσης:

```
Δώσε το 0ο στοιχείο: 5.5
Δώσε το 1ο στοιχείο: 6.3
Δώσε το 2ο στοιχείο: 4
```

. . .

Ας πούμε τώρα ότι ο χρήστης δεν θέλει να δώσει όλα τα στοιχεία αλλά θέλει να σταματήσει με φρουρό (9999). Τι κάνουμε στην περίπτωση αυτή;

```
m = 0;  cin >> x[m];
while ( x[m] != 9999 && m < N-1 )
{
   m = m + 1;  cin >> x[m];
} // while
```

Όπως βλέπεις, η *m* ξεκινάει από 0 και αυξάνεται κατά 1 όσο έχουμε `m < N-1`. Άρα αποκλείεται να πάρει τιμή μεγαλύτερη από *N*–1. Όταν τελειώσει η εκτέλεση της **while** θα έχουμε: `x[m] == 9999 || m >= N-1`.

- Αν έχουμε `x[m] == 9999` τότε στο x[*m*] έχουμε τον φρουρό, που δεν είναι τιμή προς επεξεργασία. Άρα θα έχουν διαβαστεί *m* τιμές στα x[0] .. x[*m*–1].
- Αν έχουμε *m* `>=` *N*–1 τότε θα έχουμε για την ακρίβεια *m* `==` *N*–1 (αφού η *m* παίρνει πρώτη τιμή 0 και αυξάνεται κατά 1 κάθε φορά). Στην περίπτωση αυτή θα έχουν διαβαστεί ήδη *N* (= *m* + 1) τιμές στα x[0] .. x[*N*–1].

Έτσι, αν θέλουμε να έχουμε το πλήθος των τιμών που διαβάστηκαν στη μεταβλητή *count*, θα πρέπει να βάλουμε:

```
if ( x[m] == 9999 )  count = m;
                else  count = m + 1;
```

Αν θέλουμε να δώσουμε στον χρήστη το δικαίωμα να δίνει και τον δείκτη του στοιχείου θα πρέπει να σκεφτούμε διαφορετικά:

```
cout << "Δώσε τη θέση του στοιχείου: ";  cin >> m;
cout << "Δώσε το " << m << "ο στοιχείο: ";  cin >> x[m];
```

Τώρα όμως πρέπει να σκεφτούμε και άλλα δύο πράγματα:

- Μπορεί να πάρουμε παράνομη τιμή του δείκτη –στην περίπτωσή μας μικρότερη από 0 ή μεγαλύτερη από *N*–1. Θα πρέπει λοιπόν να ελέγχουμε την τιμή του *m* πριν διαβάσουμε το x[*m*].
- Δεν είναι καθόλου σίγουρο ότι ο χρήστης θα δώσει τιμές για όλα τα στοιχεία. Δεν μπορούμε λοιπόν να δουλεύουμε με τη **for**. Θα πρέπει να δουλεύουμε μάλλον με φρουρό (π.χ. -1 ή κάποιον άλλον αρνητικό στο *m*).

```
cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
     << ") (-1 για ΤΕΛΟΣ): ";
cin >> m;
while ( m != -1 )
{
   if ( 0<= m && m <= N-1 )
   {
      cout << "Δώσε το " << m << "ο στοιχείο: ";
      cin >> x[m];
   }
   else
      cout << "*** Λάθος θέση ***" << endl;
   cout << "Δώσε τη θέση του στοιχείου (0.." << (N-1)
        << ") (-1 για ΤΕΛΟΣ): ";
   cin >> m;
} // while
```

Παράδειγμα εκτέλεσης:

```
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 7
Δώσε το 7ο στοιχείο: -15.7
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 3
Δώσε το 3ο στοιχείο: -3
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 11
*** Λάθος θέση ***
Δώσε τη θέση του στοιχείου (0..8) (-1 για ΤΕΛΟΣ): 1
Δώσε το 1ο στοιχείο: 6.3
```

. . .

Παρατήρηση: ▶

Αν γράφεις ένα «πραγματικό» πρόγραμμα θα πρέπει να προσέχεις αυτά τα «7ο στοιχείο» ή «στοιχείο στη θέση 7» που είναι σίγουρο ότι θα προκαλέσουν σύγχυση και λάθη (το πρώτο στοιχείο βρίσκεται στη θέση 0;!). Αν ο χρήστης είναι κάποιος που ξέρει προγραμματισμό μπορείς να αναφερθείς στο «*στοιχείο με δείκτη* 2»· αν δεν ξέρει, θα πρέπει να βρεις τη γλώσσα που καταλαβαίνει και να του μιλήσεις με αυτήν.◀

Μερικές φορές, όταν ο πίνακας είναι μικρός, μπορεί να θελήσεις να διαβάσεις όλα τα στοιχεία του από μια γραμμή. Αυτό είναι απλό: από τον 1ο τρόπο αφαιρούμε τα μηνύματα:

```
for ( m = 0; m <= N-1; m = m+1 )  cin >> x[m];
```

Σε αυτό μπορείς να απαντήσεις –αν π.χ. $N = 5$– με τη γραμμή:

```
5.5   6.3  4.0   -3  5.1<Enter>
```

9.2.2 Εισαγωγή Στοιχείων από Αρχείο

Τα παραπάνω ισχύουν και για εισαγωγή τιμών από αρχείο. Αλλά:

- Πρέπει να προσθέσουμε ελέγχους για τέλος αρχείου.
- Δεν έχουν νόημα τα μηνύματα προς το χρήστη.

Έστω ότι έχουμε $N = 9$, το ρεύμα:

```
ifstream t( "text1.dta" );
```

και το αρχείο text1.dta με περιεχόμενο:

```
-7   -15      8
    14       33
-8  16   114
    375
```

Αν είναι σίγουρο ότι στο αρχείο υπάρχουν N τιμές, μπορείς να διαβάσεις έτσι:

```
for ( m = 0; m <= N-1; m = m+1 )
{
   t >> x[m];
} // for
```

Αλλά, επειδή οι τιμές μπορεί να είναι λιγότερες ή περισσότερες, πιο σίγουρος είναι ο παρακάτω τρόπος (αντικαταστήσαμε τον έλεγχο του φρουρού με τη `!t.eof()`):

```
m = 0;  t >> x[m];
while ( !t.eof() && m < N-1 )
{
   m = m + 1;  t >> x[m];
} // while
if ( t.eof() )  count = m;
          else  count = m+1;
```

Ας πούμε τώρα ότι $N = 9$ και το αρχείο arrval.txt έχει τα εξής:

```
7   -15.7
3    -3
8     3.75
2     4.0
6     0
11    6.3
4     5.1
5   -13
0     5.5
```

Σε κάθε γραμμή, ο πρώτος αριθμός δείχνει τον δείκτη στον πίνακα και η δεύτερη την τιμή του αντίστοιχου στοιχείου. Διαβάζουμε τις τιμές των στοιχείων, με το 2ο τρόπο, ως εξής:

```
ifstream t( "arrval.txt" );
double y;
int    rNum;
```

```
rNum = 0;
t >> m;
while ( !t.eof() )
{
   rNum = rNum + 1;
   if ( 0<= m && m <= N-1 )
      t >> x[m];
   else
   {
      cout << "Λάθος τιμή δείκτη στη γραμμή " << rNum << endl;
      t >> y;
   }
   t >> m;
} // while
t.close();
```

Αυτές οι εντολές θα δώσουν τιμές σε όλα τα στοιχεία του πίνακα εκτός από το x[1]. Και θα μας δώσουν και ένα μήνυμα λάθους:

```
Λάθος τιμή δείκτη στη γραμμή 6
```

διότι εκεί δίνεται τιμή δείκτη 11.

9.2.3 Γράψιμο Στοιχείων

Το γράψιμο είναι πιο απλό. Ας πούμε ότι θέλουμε να γράψουμε τα στοιχεία του x σε μια γραμμή της οθόνης. Δίνουμε:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << "  ";
cout << endl;
```

και παίρνουμε:

```
5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
```

Καταλαβαίνεις βέβαια ότι το `<< "  "` είναι απαραίτητο για να διαχωρίζονται οι τιμές. Να τι θα βγει αν το παραλείψεις:

```
5.56.34-35.1-130-15.73.75
```

Αν έχεις το *t* (*ofstream*) συνδεμένο με κάποιο αρχείο-κείμενο τότε οι:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << "  ";
t << endl;
```

γράφουν τις τιμές σε μια γραμμή του αρχείου.

Αν θέλεις να βάζεις μια τιμή σε κάθε γραμμή θα πρέπει να δίνεις `endl` για κάθε στοιχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  cout << x[m] << endl;
```

και για αρχείο:

```
for ( m = 0; m <= N-1; m = m+1 )  t << x[m] << endl;
```

Δες ακόμη πώς γράψαμε τις τιμές των στοιχείων του x στο παράδ. της §9.2.

Ας πούμε όμως ότι έχουμε:

```
const int N( 50 );
int z[N];
```

και θέλεις να τυπώσεις τον πίνακα z όπως φαίνεται στο Σχ. 9-2. Πώς γίνεται αυτό;

Όπως φαίνεται έχουμε να τυπώσουμε 10 γραμμές, αριθμημένες (από την 1η στήλη) από 0 μέχρι 9. Αυτό γίνεται ως εξής:

```
for ( r = 0; r <= 9; r =r + 1 )
{
   Γράψε τη γραμμή r
} // for (r = ...
```

Τώρα ας δούμε πώς θα γράψουμε τη γραμμή *r*. Ας πάρουμε για παράδειγμα τις γραμμές 0 και 1· τι περιέχουν; Τα:

```
 0   35     10    9     20  872     30  232     40  347
 1   18     11   34     21    0     31  667     41   61
 2   15     12   21     22   23     32  139     42  753
 3   80     13   57     23  -34     33  -45     43   73
 4   10     14  239     24  -32     34   -9     44    6
 5   40     15  909     25   56     35  -89     45  -37
 6   23     16  213     26  787     36   34     46   43
 7  789     17  576     27  146     37  576     47  -99
 8  563     18  903     28  589     38  122     48  344
 9    1     19  239     29  568     39   99     49  572
```

Σχ. 9-2 Εκτύπωση των τιμών των στοιχείων του πίνακα z. Πριν από κάθε τιμή γράφεται ο δείκτης. Π.χ. 0 35 σημαίνει ότι το στοιχείο z[0] έχει τιμη 35.

0, z[0], 10, z[10], 20, z[20], 30, z[30], 40, z[40] (γραμμή 0)
1, z[1], 11, z[11], 21, z[21], 31, z[31], 41, z[41] (γραμμή 1)
Καταλαβαίνεις λοιπόν ότι μπορούμε να γράψουμε τη γραμμή r ως εξής:

```
    cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
         << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
```

(φυσικά θα πρέπει να βάλουμε και κενά για να διαχωρίζονται οι τιμές.)

Να λοιπόν η λύση στο πρόβλημά μας:

```
for ( r = 0; r <= 9; r = r + 1 )
{
   cout << r << z[r] << (r+10) << z[r+10] << (r+20) << z[r+20]
        << (r+30) << z[r+30] << (r+40) << z[r+40] << endl;
} // for (r =…
```

Αλλά μπορούμε να τη γράψουμε πιο κομψά: Η γραμμή δεν βγαίνει με **for**; Για δες αυτήν[1]:

```
    for ( c = 0; c <= 40; c = c + 10 )
        cout << (r + c) << z[r+c];
```

Να ολόκληρο το κομμάτι μαζί με τον καθορισμό πλάτους πεδίου για να ξεχωρίζουν οι τιμές μεταξύ τους:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 40; c = c + 10)
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << z[r+c];
   } // for (c =...
   cout << endl;
} // for (r =...
```

9.2.4 Απλοί Υπολογισμοί

Οι υπολογισμοί αθροίσματος και γινομένου στοιχείων πίνακα γίνονται όπως ξέρουμε:

```
sum = 0;
for ( m = 0; m <= N-1; m = m + 1 )
   sum = sum + x[m];// άθροισμα
```

[1] Άλλοι προτιμούν να γράψουν το εξής:

```
for (r = 0; r <= 9; r = r + 1)
{
   for (c = 0; c <= 4; c = c + 1)
   {
      cout.width(7);  cout << (r + c*10);
      cout.width(5);  cout << z[r+c*10];
   } // for (c =...
   cout << endl;
} // for (r =...
```

Έτσι έχεις το πλεονέκτημα το *c* να έχει πάντοτε ως τιμή τον αριθμό της στήλης (0 .. 4).

και

```
product = 1;
for ( m = 0; m <= N-1; m = m + 1 )
   product = product * x[m];// γινόμενο
```

Για να βρούμε το μέγιστο (ελάχιστο) δεν χρειάζεται να καταφύγουμε στην τεχνική της απίθανα μικρής (μεγάλης) αρχικής τιμής: Θεωρούμε αρχικώς ότι μέγιστη (ελάχιστη) είναι η τιμή του στοιχείου με δείκτη 0:

```
maxNdx = 0;  xMax = x[maxNdx];
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > xMax )
   {  maxNdx = m;  xMax = x[m];  }
} // for (m …
```

Στη *xMax* κρατάμε τη μέγιστη από τις τιμές των στοιχείων του πίνακα και στη *maxNdx* τον δείκτη του στοιχείου στον πίνακα. Πρόσεξε όμως το εξής: το βασικό είναι να βρούμε τον *δείκτη* του μεγίστου (*maxNdx*). Αν τον ξέρουμε, αφού έχουμε όλες τις τιμές στη μνήμη, παίρνουμε και τη μέγιστη τιμή:

```
maxNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] > x[maxNdx] )  maxNdx = m;
} // for (m …
xMax = x[maxNdx];
```

Παρομοίως δουλεύουμε και για το ελάχιστο:

```
minNdx = 0;
for ( m = 1; m <= N-1; m = m + 1 )
{
   if ( x[m] < x[minNdx] )  minNdx = m;
} // for (m …
xmin = x[minNdx];
```

9.3 Παράμετρος – Πίνακας

Κάτι που θα σου είναι πολύ χρήσιμο είναι το να γράφεις συναρτήσεις με παραμέτρους - πίνακες.

Ας δούμε πώς γίνεται αυτό με ένα παράδειγμα. Η:

```
double vectorSum( double x[], int n )
{
   int m;
   double sum( 0 );

   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

υπολογίζει και επιστρέφει το άθροισμα των τιμών των στοιχείων ενός πίνακα με στοιχεία τύπου `double`.

Θέλήσαμε να κάνουμε τη συνάρτησή μας γενική, ώστε να δουλεύει με οποιονδήποτε πίνακα με στοιχεία τύπου `double`. Βάλαμε τον x ως παράμετρο στη συνάρτηση *αλλά δεν βάλαμε πλήθος στοιχείων*. Το πλήθος *n* των στοιχείων του πίνακα το περνάμε με άλλη παράμετρο.

Με το ίδιο τρόπο μπορούμε να γράψουμε και μια συνάρτηση που θα επιστρέφει τον δείκτη του στοιχείου με τη μέγιστη από τις τιμές ενός πίνακα.

```
int maxNdx( int x[], int n )
{
   int m;
```

```
    int mxp( 0 );

    for ( m = 1; m <= n-1; m = m + 1 )
    {
       if (x[m] > x[mxp])  mxp = m;
    } // for (m ...
    return mxp;
} // maxNdx
```

Ας πούμε ότι σε ένα πρόγραμμα έχουμε:

```
const int N( 9 ), Nd2( N/2 );

double x[N] = { 5.5, 6.3, 4, -3, 5.1, -13, 0, -15.7, 3.75 },
       u[Nd2] = { 5.5, 4, 5.1, 0 };
int ix[N] = { 5, 6, 4, -3, 1, -13, 0, -15, 3 },
    iu[Nd2] = { 5, 4, 1, 0 };
```

και δίνουμε:

```
cout << " Σx: " << vectorSum( x, N ) << endl;
cout << " Σu: " << vectorSum( u, Nd2 ) << endl;

cout << " max(ix): " << ix[maxNdx(ix, N)]
     << " στη θεση: " << maxNdx( ix, N ) << endl;
cout << " max(iu): " << iu[maxNdx(iu, Nd2)]
     << " στη θεση: " << maxNdx( iu, Nd2 ) << endl;
```

Αποτέλεσμα:

```
Σx: -7.05
Σu: 14.6
max(ix): 6 στη θεση: 1
max(iu): 5 στη θεση: 0
```

Πρόσεξε τις κλήσεις των συναρτήσεων: Παίρνουμε το άθροισμα των τιμών των στοιχείων

- του x με τη `vectorSum(x, N)` και
- του u με τη `vectorSum(u, Nd2)`.

Ακόμη:

- Η `maxNdx(ix, N)` μας δίνει τον *δείκτη* του μέγιστου στοιχείου του ix ενώ παίρνουμε την *τιμή* αυτού του στοιχείου με την `ix[maxNdx(ix, N)]`.
- Για τον iu δείκτης μέγιστου στοιχείου: `maxNdx(iu, Nd2)` και *τιμή* μέγιστου στοιχείου: `iu[maxNdx(iu, Nd2)]`.

Αλλά, δυστυχώς, η *vectorSum* δέχεται μόνο πίνακες με στοιχεία τύπου `double`. Αν θέλουμε το άθροισμα των στοιχείων του ix θα πρέπει να γράψουμε άλλη συνάρτηση. Παρομοίως, η *maxNdx* δέχεται πίνακα τύπου `int` μόνον. Αργότερα θα δούμε ότι μπορούμε να βάλουμε τον μεταγλωττιστή να γράφει τις συναρτήσεις που μας ενδιαφέρουν.

Και γιατί βάλαμε τον τύπο της *maxNdx* `int` και όχι `unsigned int`; Αυτό θα ήταν το σωστό, αλλά διάβασε αυτά που ακολουθούν.

Μερικές φορές μας ενδιαφέρει να κάνουμε μια επεξεργασία σε ένα κομμάτι του πίνακα μόνον· ας πούμε να βρούμε το άθροισμα των πρώτων πέντε στοιχείων του x. Θα μπορούσαμε να καλέσουμε τη *vectorSum* ως εξής:

```
q = vectorSum( x, 5 );
```

Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη *vectorSum* και για την περίπτωση αυτή.

Πάντως αυτές οι χρήσεις έχουν ένα πρόβλημα: μπερδεύουν τη δομή του πίνακα με την περιοχή επεξεργασίας, Αυτό μπορεί να οδηγήσει σε μερικά πολύ «δύσκολα» προγραμματιστικά λάθη. Μέχρι να γίνεις μεγάλος/η προγραμματιστής/τρια καλύτερα να διαχωρίζεις αυτά τα στοιχεία στη συνάρτηση. Π.χ.:

```
double vectorSum( double x[], int n, int from, int upto )
```

```
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Φυσικά, θα πρέπει να έχουμε:

$$0 \le from \le upto \le n-1$$

Και αν δεν ισχύει η συνθήκη τι κάνουμε; Κατ' αρχήν θα πρέπει να καλέσουμε την *exit()*. Επειδή όμως δεν είναι σπάνιο να καλούμε μια συνάρτηση σαν αυτήν με $n == 0$ ή με $upto < from$ και να περιμένουμε από τη συνάρτηση τιμή 0, θα γράψουμε:

```
   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 ) sum = sum + x[m];
   }
```

Παρομοίως γράφουμε:

```
int maxNdx( int x[], int n, int from, int upto )
{
   int m;
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      mxp = from;
      for ( m = from+1; m <= upto; m = m + 1 )
      {
         if (x[m] > x[mxp])  mxp = m;
      } // for (m ...
   }
   return mxp;
} // maxNdx
```

Όπως βλέπεις και εδώ αποφεύγουμε να καλέσουμε την *exit* αλλά εδώ έχουμε τη δυνατότητα –όταν υπάρχει πρόβλημα με την περιοχή επεξεργασίας– να επιστρέψουμε μια τιμή ("`-1`") που δεν μπορεί να είναι δείκτης στοιχείου πίνακα.[2]

Οι παράμετροι πίνακες έχουν μια σημαντική διαφορά από τις παραμέτρους τιμής: *Η τυπική παράμετρος-πίνακας δεν είναι αντίγραφο της πραγματικής παραμέτρου· είναι το ίδιο αντικείμενο*. Γι' αυτό, προσοχή!

- ***Αν αλλάξεις τιμές στοιχείων παράμετρου-πίνακα μέσα στη συνάρτηση η αλλαγή περναει στη συνάρτηση που έκανε την κλήση.***

Παράδειγμα

Αλλάζουμε τη *vectorSum* ως εξής:

```
double vectorSum( double x[], int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )  sum = sum + x[m];

   for ( m = 0; m <= n-1; m = m + 1 )  x[m] = 0;

   return sum;
```

[2] Καταλαβαίνεις τώρα γιατί προτιμήσαμε να βάλουμε τύπο αποτελέσματος της συνάρτησης `int` και όχι `unsigned int`.

```
} // vectorSum
```

Δηλαδή, αφού υπολογίσουμε το άθροισμα των στοιχείων, αλλάζουμε τις τιμές τους σε "**0**". Καλούμε τη συνάρτηση και γράφουμε τις τιμές των στοιχείων πριν και μετά την κλήση:

```
cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;

cout << " Σx: " << vectorSum( x, N, 0, N-1 ) << endl;

cout << " x:  ";
for (m = 0; m <= N-1; m = m + 1)  cout << x[m] << "  ";
cout << endl;
```

Αποτέλεσμα:

```
x:  5.5  6.3  4  -3  5.1  -13  0  -15.7  3.75
Σx: -7.05
x:  0  0  0  0  0  0  0  0  0
```

Η C++ σου δίνει ένα εργαλείο για να προστατευθείς από αλλαγές στοιχείων του πίνακα κατά λάθος. Μπορείς να δηλώνεις τις παραμέτρους σου **const** (σταθερές). Αν στο παραπάνω παράδειγμα δηλώσεις:

```
double vectorSum( const double x[], int n, int from, int upto )
{ . . . }
int maxNdx( const int x[], int n, int from, int upto )
{ . . . }
```

τότε αν ο *μεταγλωττιστής* βρει μέσα στο σώμα της συνάρτησης εντολή που να αλλάζει τιμή στοιχείου του x θα βγάλει λάθος: «Cannot modify a const object» (δεν μπορείς να αλλάξεις ένα αντικείμενο **const**).

Παρατήρηση: ▶

Όταν γράφουμε μια συνάρτηση αναδρομικώς μας *είναι απαραίτητη η περιοχή επεξεργασίας* η οποία περιορίζεται σε κάθε αναδρομική κλήση. Δίνουμε στη συνέχεια τις δύο συναρτήσεις σε αναδρομική μορφή:

```
double rVectorSum( const double x[], int n, int from, int upto)
{
   double sum( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      sum = x[upto] + rVectorSum( x, n, from, upto-1 );
   }
   return sum;
} // rVectorSum
```

Σε κάθε αναδρομική κλήση μειώνεται κατά 1 η τιμή της *upto* μέχρι που να γίνει κλήση με *upto* < *from* και η *sum* μένει με το 0 παίρνει αρχικώς.

```
int rMaxNdx( const int x[], int n, int from, int upto )
{
   int mxp;

   if ( from < 0 || upto < from || n <= upto )
      mxp = -1;
   else // 0 <= from <= upto <= n-1
   {
      if ( from == upto )
         mxp = from;
      else
      {
         mxp = rMaxNdx( x, n, from+1, upto );
         if ( x[from] > x[mxp])  mxp = from;
      }
   }
```

```
      return mxp;
   } // rMaxNdx
```

Εδώ ο περιορισμός της περιοχής επεξεργασίας γίνεται με αύξηση της τιμής της *from*. ◀

9.4 Δύο Παραδείγματα με Αριθμούς

Στην παράγραφο αυτή θα δούμε δυο αριθμητικά παραδείγματα, από τα πιο χαρακτηριστικά για πίνακες. Είναι πολύ πιθανό να σου φανούν χρήσιμα σε προγράμματα που θα γράψεις για άλλα ενδιαφέροντα ή άλλες υποχρεώσεις σου.

Πριν προχωρήσουμε όμως να σου τονίσουμε κάτι, αν δεν το έχεις καταλάβει ήδη. Στα προγράμματα με πίνακες τα στοιχεία εισόδου είναι συνήθως πολλά. Μέχρι να κάνεις ένα πρόγραμμα να δουλέψει –μέχρι να διορθώσεις τα διάφορα σημαντικά λάθη– θα χρειαστεί συνήθως να πληκτρολογήσεις αρκετές φορές τα ίδια στοιχεία, πράγμα ενοχλητικό και αντιπαραγωγικό! Μια καλή λύση είναι η εξής: Γράψε τα στοιχεία με τα οποία δοκιμάζεις το πρόγραμμά σου σε ένα αρχείο text. Γράψε το πρόγραμμά σου έτσι που να μην διαβάζει από το πληκτρολόγιο, αλλά από το αρχείο.

Έτσι είναι γραμμένα τα παραδείγματα που ακολουθούν.

Παράδειγμα 1 - Τιμή Πολυωνύμου (αλγόριθμος του Horner)

Ένα πολύ συνηθισμένο πρόβλημα, σε αριθμητικά προγράμματα, είναι ο υπολογισμός της τιμής πολυωνύμου:

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m = \sum_{k=0}^{m} a_k x^k$$

όταν δίνονται οι συντελεστές (a_k, $k = 0..m$) και η τιμή της x.

Θέλουμε μια συνάρτηση που θα παίρνει ως παραμέτρους, τους συντελεστές και το x και θα επιστρέφει ως τιμή το $p(x)$.

Οι συντελεστές θα πρέπει να αποθηκευτούν σε ένα μονοδιάστατο πίνακα τύπου `double` με $m+1$ στοιχεία ($0 .. m$). Η συνάρτηση που γράφουμε είναι αρκετά γενική:

```
   double p1( const double a[], int m, double x )
```

Για να δοκιμάσουμε τη συνάρτηση που θα γράψουμε, ετοιμάζουμε το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

   double p1( const double a[], int m, double x );

int main()
{
   const int N( 20 );

   double   a[N];    // συντελεστές πολυωνύμου
   int      b;       // βαθμός πολυωνύμου
   double   x, y;
   int      k, pa;
   ifstream syntFl;  // από το αρχείο Text με τους συντελεστές

   syntFl.open( "syntFl.txt" );
   syntFl >> b;
   if ( syntFl.eof() )
   {
      syntFl.close();
      cout << "αρχειο συντελεστών άδειο" << endl;
   }
   else if ( b < 0 || b > N-1 )
```

```
   {
      syntFl.close();
      cout << "λάθος βαθμός πολυωνύμου" << endl;
   }
   else   // θέλουμε να διαβάσουμε b+1 συντελεστές
   {      // στις θεσεις a[0]..a[b]
      k = 0;  syntFl >> a[k];
      while ( !syntFl.eof() && k <= b-1 )
      {
         k = k + 1;  syntFl >> a[k];
      } // while
      if ( syntFl.eof() )  pa = k;
                     else  pa = k + 1;
      syntFl.close();
   // διαβάσαμε pa συνετελεστές
      if ( pa < b + 1 )
         cout << "Διάβασα " << pa << " τιμές" << endl;
      else    // όλα εντάξει
      {
         cout << " ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :" << endl;
         for ( k = 0; k <= b; k = k+1 )  cout << a[k] << "  ";
         cout << endl;

         cout << "ΔΩΣΕ  x = ";   cin >> x;
         while ( x != 0 )
         {
            cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
                 << p1( a, b, x ) << endl;
            cout << "ΔΩΣΕ  x = ";   cin >> x;
         } // while
         cout << " ΑΠΟΤΕΛΕΣΜΑ:  p(" << x << ")= "
              << p1( a, b, x ) << endl;
      } // if (pa ...
   } // !syntFl.eof ...
} // main
```

Το syntfl.txt είναι ένα αρχείο text, όπου γράφουμε, με τον κειμενογράφο μας, στην πρώτη γραμμή το βαθμό του πολυωνύμου και στις επόμενες τους συντελεστές. Βλέπεις ότι διαβάζουμε τα στοιχεία του a όπως είδαμε στην §9.1.1 και τους γράφουμε όπως είδαμε στην §9.1.2. Πριν από αυτό, διαβάζουμε και ελέγχουμε την τιμή του βαθμού του πολυωνύμου· θα πρέπει να είναι: $0 \le b \le N - 1$.

Ας πάρουμε για παράδειγμα το παρακάτω πολυώνυμο, 10ου βαθμού, που αποτελείται από 11 όρους (έχει 11 συντελεστές):

$p(x) = -10 + 9.5x + 7.2x^2 + 6.8x^3 - 6.1x^4 + 5.3x^5 - 4.6x^6 + 3.5x^7 + 2.7x^8 - x^9 + 0.8x^{10}$

Στο αρχείο μας θα γράψουμε:

```
10
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
```

Ελέγχουμε τη **while** με φρουρό το "**0**" στη x. Πάντως δεν ξεχνούμε να γράψουμε και το $p(0)$ στο τέλος.

Και τώρα, να γράψουμε τη συνάρτηση, δηλαδή εντολές που υπολογίζουν το άθροισμα που γράψαμε πιο πάνω. Εύκολο:

```
      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
      {
         Υπολόγισε το x^k
         px = px + a[k]*x^k
      }
```

Πως υπολογίζουμε το x^k; Ε, αυτό το ξέρουμε: **pow(x, k)**. Η συνάρτηση που θέλαμε, γράφτηκε εύκολα:

```
   double p1( const double a[], int m, double x )
   {
```

```
      double px;
      int k;

      px = a[0];
      for ( k = 1; k <= m; k = k+1 )
         px = px + a[k]*pow(x, k);
      return px;
   }  // p1
```

Εισάγουμε τη συνάρτηση στο πρόγραμμά μας και να ένα παράδειγμα εκτέλεσης:

```
 ΣΥΝΤΕΛΕΣΤΕΣ ΤΟΥ ΠΟΛΥΩΝΥΜΟΥ :
-10  9.5  7.2  6.8  -6.1  5.3  -4.6  3.5  2.7  -1  0.8
ΔΩΣΕ  x = 0.3562
 ΑΠΟΤΕΛΕΣΜΑ:  p(0.3562)= -5.46928
ΔΩΣΕ  x = -0.45
 ΑΠΟΤΕΛΕΣΜΑ:  p(-0.45)= -13.8303
ΔΩΣΕ  x = 0
 ΑΠΟΤΕΛΕΣΜΑ:  p(0)= -10
```

Και τώρα να *μετρήσουμε*. Ο αλγόριθμός μας κάνει:

m υψώσεις σε δύναμη (pow),

m πολλαπλασιασμούς *a*[*k*] * ...

m προσθέσεις *px* + *a*[*k*] * ...

Δεν μετράμε εκχωρήσεις και τις πράξεις για τον έλεγχο της **for**.

Τι έγινε όμως εδώ; Αγνοήσαμε τελείως το γεγονός ότι: $x^k = x^{k-1} \cdot x$ και για κάθε όρο υπολογίζαμε το x^k από την αρχή. Ασυγχώρητα σπάταλο πρόγραμμα. Η *p2()* διορθώνει αυτήν την αβλεψία:

```
   double p2( const double a[], int m, double x )
   {
      double px, xPow;
      int    k;

      px = a[0];  xPow = 1;
      for ( k = 1; k <= m; k = k+1 ) // εδώ έχουμε xPow = x^(k-1)
      {
         xPow = xPow * x;            // xPow = x^k
         px = px + a[k]*xPow;
      }
      return  px;
   }  // p2
```

Εδώ υπολογίζουμε το x^k με ένα πολλαπλασιασμό από το x^{k-1} και αποφεύγουμε την χρονοβόρα ύψωση σε δύναμη.

Βάλε την *p2()* στο πρόγραμμά σου και, όπου υπάρχει "`p1`" άλλαξε το σε "`p2`". Δοκίμασέ το και θα πάρεις τα ίδια αποτελέσματα που είδαμε παραπάνω. Αλλά τώρα έχουμε:

$2m$ πολλαπλασιασμούς (`xPow * x` και `a[k] * xPow`),

m προσθέσεις (`px + a[k] ...`),

Εδώ δεν έχουμε υψώσεις σε δύναμη και αυτό είναι σημαντικό κέρδος.

Όμως μπορούμε να έχουμε ένα καλύτερο πρόγραμμα! Πώς; Ας δώσουμε ένα παράδειγμα, με το πολυώνυμο τρίτου βαθμού:

$$p(x) = ax^3 + bx^2 + cx + d$$

Υπολογίζοντας την τιμή του με την *p2()*, θα κάνουμε 6 πολλαπλασιασμούς και 3 προσθέσεις.

Ένας άλλος τρόπος να γράψουμε το $p(x)$ είναι ο εξής:

$$p(x) = ((ax + b)x + c)x + d$$

που μας υποδεικνύει έναν άλλο τρόπο υπολογισμού:

$p(x) \leftarrow a;$

$p(x) \leftarrow p(x)x + b;$

$p(x) \leftarrow p(x)x + c;$
$p(x) \leftarrow p(x)x + d;$

Μετράμε: 3 πολλαπλασιασμοί και 3 προσθέσεις. Θρίαμβος! Αυτός ο νέος τρόπος υπολογισμού, που λέγεται **αλγόριθμος του Horner** ή μέθοδος των **φωλιασμένων πολλαπλασιασμών** (nested multiplication) δίνεται στη συνάρτηση ph:

```
// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x )
{
   double px;
   int    k;

   px = a[m];
   for ( k = m-1; k >= 0; k = k-1 )  px = px*x + a[k];
   return  px;
}  // ph
```

αλλά τώρα, κάνοντας μόνον:

m πολλαπλασιασμούς (`px*x`),
m προσθέσεις (`px*x + a[k]`),

Τι καταφέραμε λοιπόν; Η περιπλοκότητα αυτού του αλγόριθμου είναι τάξης *N*, όπως και του πρώτου. Αλλά, το ότι ελαττώσαμε τον αριθμό των πολλαπλασιασμών –αφού πρώτα-πρώτα διώξαμε αυτούς που χρειάζονται για την ύψωση σε δύναμη– μας δίνει διπλό κέρδος:

- *λιγότερα λάθη στρογγύλευσης*, δηλ. μεγαλύτερη ακρίβεια του αποτελέσματος· τα λάθη στρογγύλευσης είναι από τα σοβαρότερα προβλήματα με τον τύπο `double`,
- εξοικονόμηση χρόνου, γιατί οι πράξεις στον τύπο `double` είναι πιο αργές.

Συνεπώς ο αλγόριθμος του Horner είναι πολύ καλύτερος από τους προηγούμενους. Για υπολογισμό 500 τιμών του παραπάνω πολυωνύμου οι λόγοι των χρόνων των τριών προγραμμάτων είναι: 210:71:42. Όπως βλέπεις, η μεγάλη διαφορά προέρχεται από τον υπολογισμό της δύναμης, περίπου 3:1. Από τον υποδιπλασιασμό του πλήθους των πολλαπλασιασμών είχαμε περίπου υποδιπλασιασμό του χρόνου, περίπου 7:4.

Ποιό είναι το δίδαγμα από τα παραπάνω; Κατ' αρχήν:

♦ ***Υπάρχει πάντα μια καλύτερη λύση!***

Πάντα; Ναι, εκτός αν μπορείς να αποδείξεις το αντίθετο. Και θα ψάχνουμε πάντα για την καλύτερη λύση; Δεν θα πάρουμε υπ' όψη μας ότι την *p1()* την είχαμε έτοιμη στο κεφάλι μας πριν μας τη ζητήσουν; Και βέβαια!

Ας πούμε τα πράγματα με οικονομικούς όρους: Η *p1()* είχε πολύ μικρό **κόστος ανάπτυξης**, ενώ η *ph()* είχε μικρό **κόστος εκμετάλλευσης**[3]. Δεν θα πρέπει να αγνοήσουμε την *p2()*, όπου επιτύχαμε ικανοποιητικό κόστος εκμετάλλευσης χωρίς μεγάλο κόστος ανάπτυξης.

Αν η συνάρτηση, που έχουμε να γράψουμε, πρόκειται να χρησιμοποιηθεί σε κάποιο πρόγραμμα, που υπολογίζει χιλιάδες τιμές και θα χρησιμοποιείται πολύ καιρό, θα πρέπει να κατεβάσουμε το κόστος εκμετάλλευσης με κάθε θυσία (στην περίπτωσή μας: υψηλό κόστος ανάπτυξης). Αν πρόκειται να χρησιμοποιηθεί μια φορά, τότε δεν είναι και τρομερό να χρησιμοποιήσουμε κάτι που το γράφουμε πολύ γρήγορα, ακόμα κι αν δεν έχει την "ταχύτητα του φωτός".

Ο καλός προγραμματιστής ξέρει να σταθμίσει αυτούς τους παράγοντες και να δώσει τη *συνολικώς* καλύτερη λύση. Βέβαια, ο καλός προγραμματιστής έχει τις απαραίτητες γνώσεις για να γράψει κατ' ευθείαν την *ph()!*[4]

---

[3] Ένας άλλος παράγοντας, το **κόστος συντήρησης**, δεν παίζει σημαντικό ρόλο στο παράδειγμά μας.

[4] Στην πραγματικότητα την έχει ήδη έτοιμη σε κάποια από τις βιβλιοθήκες προγραμμάτων που έχει.

Παράδειγμα 2 - Στατιστικές

Αν μας δοθούν N αριθμοί x_k, $k = 0,\ldots N-1$, έχουμε τη μέση τιμή τους:

$$\langle \mathbf{x} \rangle = \frac{1}{N}\sum_{k=0}^{N-1} x_k$$

και την τυπική απόκλισή τους: $\sigma = \sqrt{\mathrm{Var}(\mathbf{x})}$, όπου:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2 \qquad (A)$$

που υπολογίζεται και ως εξής:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{N}\sum_{k=0}^{N-1} x_k^2 - \langle \mathbf{x} \rangle^2 \qquad (B)$$

Θέλουμε δυο προγράμματα τα οποία θα διαβάζουν (το καθένα) από το αρχείο text stat.dta τους N αριθμούς x_k τύπου `double` και θα βρίσκουν και θα τυπώνουν το $\langle \mathbf{x} \rangle$ και σ. Το πρώτο από τα προγράμματα θα χρησιμοποιεί τον τύπο (A) για το σ, ενώ το δεύτερο τον τύπο (B). Και τα δύο προγράμματα θα πρέπει να κάνουν τη μέγιστη δυνατή οικονομία σε μνήμη και υπολογιστικό χρόνο. Ποιό από τα δυό προγράμματα είναι καλύτερο και γιατί;

1η λύση: Έχουμε να γράψουμε δύο συναρτήσεις μια για τη μέση τιμή, ας την πούμε *vectorAvrg*, και μια για την τυπική απόκλιση, ας την πούμε stdDev. Και οι δύο θα βγουν από αλλαγές που θα κάνουμε στην *vectorSum* που γράψαμε πιο πάνω. Πρώτα η:

```
double vectorAvrg( const double x[], int n, int from, int upto)
{
   int m;
   double fv( 0 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      for ( m = from; m <= upto; m = m + 1 )  fv = fv + x[m];
      fv = fv/(upto-from+1);
   }
   return fv;
} // vectorAvrg
```

και μετά η:

```
double stdDev( const double x[], int n, int from, int upto )
{
   int m;
   double fv( 0 ), vAvg;

   if ( 0 <= from && from <= upto && upto < n )
   {
      vAvg = vectorAvrg( x, n, from, upto );
      for ( m = from; m <= upto; m = m + 1 )
         fv = fv + pow( x[m]-vAvg, 2 );
      fv = sqrt( fv/(upto-from+1) );
   }
   return fv;
} // stdDev
```

Στον υπολογισμό του $\frac{1}{N}\sum_{k=0}^{N-1}\left(x_k - \langle \mathbf{x} \rangle\right)^2$ θα μπορούσαμε να είχαμε γράψει:

```
   for ( m = 0; m <= n-1; m = m + 1 )
      sum = sum + pow( x[m]-vectorAvrg(x, n, from, upto), 2 );
```

Αλλά έτσι, ζητούμε να κληθεί n φορές η *vectorAvrg* για να υπολογίσει τη μέση τιμή. Όπως τη γράψαμε τώρα, γίνεται μόνο μια κλήση, όταν δίνουμε αρχική τιμή στη *vAvg*. Βέβαια, ένας καλός μεταγλωττιστής θα κάνει αυτήν τη μετατροπή αυτομάτως.

Η `main` δεν έχει δυσκολίες:

```
#include <iostream>
```

```
#include <fstream>
#include <cmath>
using namespace std;

   double vectorAvrg(const double x[], int n, int from, int upto);
   double stdDev( const double x[], int n, int from, int upto );

int main()
{
   const int Nmax = 100;

   double x[Nmax], y;
   int n, k;
   ifstream  t("stat.dta");

   k = 0;  t >> x[k];
   while ( !t.eof() && k <= Nmax-2 )
   {  k = k + 1;  t >> x[k];  }  // while
   if ( t.eof() )  n = k;
             else  n = k + 1;
   t.close();

   cout << " <x> = " << vectorAvrg( x, Nmax, 0, n-1 ) << endl;
   cout << " σ = " << stdDev( x, Nmax, 0, n-1 ) << endl;
} // main
```

2η λύση: Η πρώτη σκέψη είναι να ξαναγράψουμε την *stdDev* και με αυτόν τον τρόπο να κάνουμε $N - 1$ λιγότερες αφαιρέσεις. Αλλά ας το ξανασκεφτούμε. Το πρόβλημά μας λύνεται αν υπολογίσουμε τα $\sum_{k=0}^{N-1} x_k$ και $\sum_{k=0}^{N-1} x_k^2$. Αυτά όμως μπορούμε να τα υπολογίζουμε όταν διαβάζουμε το αρχείο και *δεν χρειαζόμαστε πίνακα*! Βέβαια στην περίπτωση αυτή χάνουμε τις ωραίες μας συναρτήσεις.

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main()
{
   double x, vAvg, sigma, sx, sx2;
   int n, k;
   ifstream  t( "stat.dta" );

   sx = 0;  sx2 = 0;
   k = 0;  t >> x;
   while ( !t.eof() )
   {
      k = k + 1;
      sx = sx + x;   sx2 = sx2 + x*x;
      t >> x;
   }  // while
   n = k;
   t.close();
   cout << "Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      vAvg = sx/n;
      sigma = sqrt( sx2/n - vAvg*vAvg );
      cout << " <x> = " << vAvg << endl;
      cout << " σ = " << sigma << endl;
   } // if
} // main
```

Πρόσεξε ότι εδώ διαβάζουμε όσους αριθμούς υπάρχουν, αφού δεν έχουμε πίνακα.

k	v[k]	k	v[k]	k	v[k]	k	v[k]	k	v[k]
1	35	11	9	21	872	31	232	41	347
2	18	12	34	22	0	32	667	42	61
3	15	13	21	23	23	33	139	43	753
4	80	14	57	24	-34	34	-45	44	73
5	10	15	239	25	-32	35	-9	45	6
6	40	16	909	26	56	36	-89	46	-37
7	23	17	213	27	787	37	34	47	43
8	789	18	576	28	146	38	576	48	-99
9	563	19	903	29	589	39	122	49	344
10	1	20	239	30	568	40	99	50	572

Σχ. 9-3 Ο πίνακας που θα χρησιμοποιούμε στις δοκιμές των προγραμμάτων μας. Είναι ίδιος με αυτόν του Σχ. 9-2 με τη διαφορά ότι `v[0] == INT_MIN` και `v[51] == INT_MAX`.

Η 2η λύση είναι και ταχύτερη και οικονομικότερη σε χρήση μνήμης (αφού δεν χρησιμοποιεί πίνακα).

Πάντως οι δύο συναρτήσεις που γράψαμε για την πρώτη λύση είναι χρήσιμες γενικότερα.

9.5 Και Άλλες Συνηθισμένες Δουλειές με Πίνακες

Στην επεξεργασία στοιχείων με τη βοήθεια του ΗΥ συχνά εφαρμόζουμε τις παρακάτω επεξεργασίες πινάκων:

- **Αναζήτηση** ή **ψάξιμο** (searching) για τον εντοπισμό μιας τιμής μέσα σε έναν πίνακα.
- **Ταξινόμηση** (sorting), σε αύξουσα ή φθίνουσα διάταξη, των τιμών που είναι αποθηκευμένες στον πίνακα.
- **Συγχώνευση** (merging) δυό ταξινομημένων πινάκων σε έναν.

Για τις επεξεργασίες αυτές έχουν επινοηθεί αρκετοί αλγόριθμοι. Μερικοί από αυτούς είναι εξαιρετικά ενδιαφέροντες είτε διότι είναι γρήγοροι είτε διότι είναι οικονομικοί σε μνήμη είτε διότι μπορούν να εφαρμοστούν και σε (σειριακά) αρχεία είτε τέλος διότι είναι *όμορφοι*(!)[5]. Στις επόμενες παραγράφους θα δούμε μερικούς αλγορίθμους και τα αντίστοιχα προγράμματά τους για τέτοιες επεξεργασίες. Σίγουρα οι αλγόριθμοι αυτοί δεν είναι οι καλύτεροι, αλλά είναι αρκετά *απλοί* και *κατανοητοί*.

Για τα παραδείγματά μας θα χρησιμοποιήσουμε έναν πίνακα με 50 στοιχεία τύπου `int`. Επειδή σε ορισμένες περιπτώσεις θα μας χρειαστούν και φρουροί θα δηλώσουμε:

```
const int N( 50 );

int v[N+2];
```

και θα αποθηκεύουμε τις τιμές που θέλουμε στις θέσεις από `v[1]` μέχρι `v[N]`. Στη θέση `v[0]` θα έχουμε το −∞ (*INT_MIN*) και στη θέση `v[N+1]` το +∞ (*INT_MAX*). Οι τιμές των στοιχείων (Σχ. 9-3) βρίσκονται στο αρχείο text, με όνομα στο δίσκο pin.txt και θα τις διαβάζουμε, χωρίς ελέγχους, ως εξής:

```
ifstream a;
   :
a.open( "pin.txt" );
for ( k = 1; k <= N; k = k+1 )       // Διάβασε τον πίνακα
   a >> v[k];
a.close();
v[0] = INT_MIN;  v[N+1] = INT_MAX;   // βάλε τους φρουρούς
```

[5] Αισθητική των αλγορίθμων; Μα σίγουρα θα έχει τύχει να δεις μερικές όμορφες μαθηματικές αποδείξεις. Παρόμοια αισθητική υπάρχει κι' εδώ.

Μπορείς να δεις το περιεχόμενο του πίνακα όπως φαίνεται στο Σχ. 9-3 αν στις εντολές που δώσαμε στο τέλος της §9.2.3 αλλάξεις τη **for** που διατρέχει τις γραμμές:

```
for ( c = 0; c <= 4; c = c+1 )  cout << "     k  v[k]";
cout << endl;
for ( r = 1; r <= 10; r = r+1 )   // τύπωσε τη r-οστή γραμμή
{
   for ( c = 0; c <= 40; c = c+10 )
   {
      cout.width(7);  cout << (r + c);
      cout.width(5);  cout << v[r+c];
   } // for ( c = ...
   cout << endl;                 // ...και πήγαινε στη επόμενη
} // for ( r = ...
```

9.5.1 Αναζήτηση στα Στοιχεία Πίνακα

Όταν λέμε «αναζήτηση» εννοούμε τη διαδικασία που δίνει απάντηση στο εξής πρόβλημα:

Έχουμε έναν πίνακα `T v[N]` *και μια τιμή x (τύπου T). Υπάρχει στοιχείο του* `v` *που να έχει τιμή ίση με x και αν ναι ποια η τιμή του δείκτη για το στοιχείο αυτό;*

Παρατήρηση: ▶

Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου στο πρόγραμμά μας. Στην περίπτωση αυτή η αναζήτηση δίνει απάντηση στο ερώτημα «*ανήκει η τιμή x στο σύνολο (που υλοποιείται με τον πίνακα) v;*»◀

Εμείς θα δουλέψουμε με τον πίνακα τύπου `int` που είδαμε πιο πριν. Θέλουμε λοιπόν μια:

```
int linSearch( int v[], int n, int from, int upto, int x )
```

που θα ψάχνει στα στοιχεία v`[`*from*`]`, v`[`*from*+1`]`,..., v`[`*upto*`]` να βρει την τιμή *x*. Αν τη βρει, θα επιστρέφει τον δείκτη του στοιχείου ως τιμή της συνάρτησης, αλλιώς θα επιστρέφει τιμή –1. Αν δηλαδή δώσουμε:

```
thesi = linSearch( v, N+2, n1, n2, x );
```

και αν $0 < n1 \le n2 < N + 2$ θα πρέπει:

$$(n1 \le \text{linSearch}(v, N{+}2, n1, n2, x) \le n2 \;\&\&\; v[\text{linSearch}(v, N{+}2, n1, n2, x)] == x)$$
$$|| \;(\text{linSearch}(v, N{+}2, n1, n2, x) == -1 \;\&\&\; (\forall j: n1..n2 \bullet v[j] \;!= x))$$

Τα πράγματα είναι πολύ απλά:

Ξεκίνα από την αρχή (*from*)
Όσο (δεν τελείωσε ο πίνακας) και (δεν το βρήκες) κάνε τα εξής:
{ Προχώρησε στο επόμενο στοιχείο του πίνακα

Μεταφράζουμε:

Ξεκίνα από την αρχή (*from*)	→ `k = from`
δεν τελείωσε ο πίνακας	→ `k <= upto`
δεν το βρήκες	→ `v[k] != x`
Προχώρησε στο επόμενο στοιχείο του πίνακα	→ `k = k+1`

Δηλαδή:

```
k = from;
while ( k < upto && v[k] != x )  k = k+1;
```

Δεν τελειώσαμε όμως: Η εκτέλεση της **while** θα τελειώσει:

- είτε διότι `!(k < upto)` ή αλλιώς `k >= upto` (για την ακρίβεια `k == upto` αφού η τιμή της *k* αυξάνεται κατά 1), φτάσαμε δηλαδή στο τέλος της περιοχής αναζήτησης,
- είτε διότι `!(v[k] != x)` ή αλλιώς `v[k] == x`, δηλαδή βρήκαμε την τιμή που ψάχνουμε στο στοιχείο v`[`*k*`]`.

Χρειάζεται λοιπόν και άλλος ένας έλεγχος, ακριβώς μετά τη **while**:

```
if ( v[k] == x )  fv = k;
```

```
               else  fv = -1;
```

Να λοιπόν ολόκληρη η *linSearch*:

```
   // linSearch -- Ψάχνει στα στοιχεία v[from],... v[upto] να
   //              βρει την τιμή x. Αν τη βρει
   //                 επιστρέφει ως τιμή τον δείκτη του στοιχείου
   //              αλλιώς
   //                 επιστρέφει τιμή -1
   int linSearch( const int v[], int n, int from, int upto, int x)
   {
      int k, fv( -1 );

      if ( 0 <= from && from <= upto && upto < n )
      {
         k = from;
         while ( k < upto && v[k] != x )  k = k+1;
         if ( v[k] == x )  fv = k;
                     else  fv = -1;
      // (from <= fv <= upto && v[fv] == x) ||
      //                (fv == -1 && ( j:from..upto " v[j] != x))
      }
      return fv;
   } // linSearch
```

Αυτή είναι η μέθοδος **γραμμικής αναζήτησης** (linear search). Στη συνέχεια βλέπεις και ένα πρόγραμμα που τη δοκιμάζει:

```
#include <iostream>
#include <fstream>
#include <climits>
#include <cstdlib>
using namespace std;

 int linSearch( const int v[], int n, int from, int upto, int x );

int main()
{
   const int N( 50 );

   int  v[N+2];      // ο πίνακας όπου ψάχνουμε
   int  x;           // για την τιμή της ψάχνουμε
   int  ndx;         // ο δείκτης της, αν τη βρούμε
   int  k;
   ifstream a;

// Διάβασε τον πίνακα και βάλε τους φρουρούς
// όπως παραπάνω

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = linSearch( v, N+2, 1, N, x );
      if ( ndx > 0 )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
         cout << " ΔΕΝ ΥΠΑΡΧΕΙ" << endl;
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Παράδειγμα εκτέλεσης:

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 23
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 7
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): -121
 ΔΕΝ ΥΠΑΡΧΕΙ
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 6
 ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 45
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατηρήσεις ▶

1. Όταν δώσαμε τιμή της $x = 23$ πήραμε απάντηση 7. Αλλά, αν δεις τον πίνακα στο Σχ. 9-3, το 23 υπάρχει και στη θέση 23. Η διαδικασία αυτή θα σου δώσει λοιπόν *μόνο την πρώτη εμφάνιση της τιμής*. Ο πίνακας του παραδείγματος υλοποιεί ένα **πολυσύνολο** (multiset, bag). Για ένα πολυσύνολο B η ερώτηση "$x \in B$;" δεν έχει και τόσο νόημα· η πιο σωστή ερώτηση είναι «*πόσες φορές υπάρχει η τιμή x στο πολυσύνολο B;*» (άσκ. 9-10)

2. Ειδικώς για τη συνάρτηση αυτή, η βίαιη αντίδραση (`exit(1)`) όταν είμαστε εκτός προδιαγραφών δεν είναι καλή. Πολλές φορές καλούμε μια τέτοια συνάρτηση με *μηδενική περιοχή αναζήτησης* (*from* > *upto*) και αυτό που θέλουμε είναι η απάντηση: *δεν υπάρχει η τιμή*. Γι' αυτό, όταν δεν ισχύει η συνθήκη της **if** η *fv* μένει με την τιμή "`-1`" που σημαίνει ακριβώς αυτό. ◀

Που γίνεται η πολλή δουλειά στην *linSearch()*; Προφανώς στη **while**. Για κάθε επανάληψη γίνονται 2 συγκρίσεις, 1 λογική "`&&`", 1 πρόσθεση και 1 εκχώρηση (`k = k+1`). Αν δεν υπάρχει η τιμή που ψάχνουμε –η χειρότερη περίπτωση για τον αλγοριθμό μας– όλα πολλαπλασιάζονται επί N (για την ακρίβεια επί *upto*−*from*+1).

Αν καταφέρουμε να διώξουμε τη μια σύγκριση διώχνουμε αυτόματα και τη λογική πράξη και ο αλγόριθμός μας επιταχύνεται σημαντικά. Ας δοκιμάσουμε. Και πρώτα-πρώτα: Ποια θα διώξουμε; Μάλλον δεν μπορούμε να διώξουμε την "`v[k] != x`". Γι' αυτήν γράφτηκε ολόκληρη η διαδικασία! Αν διώξουμε την "`k < upto`" πώς θα σταματήσουμε στο τέλος του πίνακα; Θα μπορούσαμε να σταματήσουμε με την άλλη συνθήκη. Αλλά αν η x δεν υπάρχει στον πίνακα; Θα τη βάλουμε εμείς με το ζόρι!

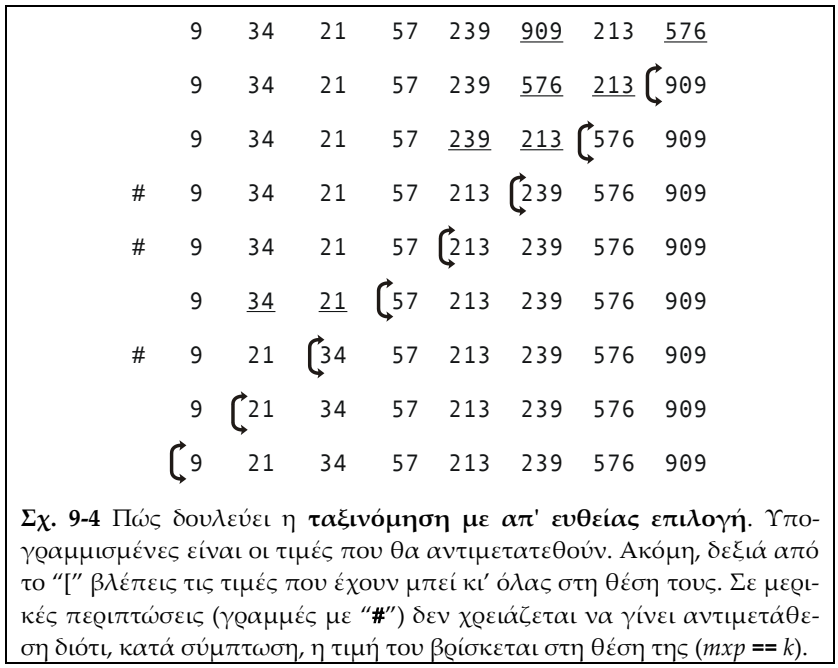
Τώρα μπερδεύτηκες, ε; Λοιπόν, πριν αρχίσουμε την αναζήτηση, θα βάλουμε την x στη θέση `v[`*upto*`+1]`. Όταν η "`while (v[k] != x) k = k+1`" σταματήσει, ελέγχουμε την τιμή της k: Αν $k \le upto$ τότε πραγματικά η x υπάρχει στον πίνακα, αλλιώς, αν $k > upto$ τότε δεν υπάρχει· σταμάτησε από τη x που βάλαμε εμείς στην `v[`*upto*`+1]`. Έξυπνο, έτσι; Αλλά κάτι σου θυμίζει! Τι άλλο; Την **τιμή - φρουρό**. Η τεχνική του φρουρού λύνει πολλά προβλήματα και απλουστεύει τα προγράμματά μας:

```
        save = v[upto+1];    // φύλαξε το v[upto+1]
        v[upto+1] = x;       // φρουρός
        k = from;
        while ( v[k] != x )  k = k+1;
        if ( k <= upto )  fv = k;
                    else  fv = -1;
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Πρόσεξε ότι πριν βάλουμε το φρουρό (x) στη θέση `v[`*upto* + 1`]` φυλάγουμε την τιμή που υπάρχει εκεί σε μια μεταβλητή (*save*) και αποκαθιστούμε την αρχική κατάσταση μετά το τέλος της αναζήτησης. Αν η *upto* έχει τιμή N δεν υπάρχει πρόβλημα αφού έχουμε προβλέψει μια ακόμη θέση (όπου έχουμε τον φρουρό "+∞").

Αλλά, δυστυχώς, ο μεταγλωττιστής έχει αντιρρήσεις: "`Cannot modify a const object in function linSearch`". Τι συμβαίνει; Φταίει το "`const`" που βάλαμε στην παράμετρο-πίνακα. Αργότερα θα δούμε πώς μπορούμε να το διορθώσουμε. Προς το παρόν θα καταφύγουμε σε «ριζικές θεραπείες»: θα βγάλουμε το "`const`"!

```
int linSearch( int v[], int n, int from, int upto, int x )
{
   int save;
   int k, fv( -1 );

   if ( 0 <= from && from <= upto && upto < n )
   {
      save = v[upto+1];    // φύλαξε το v[upto+1]
      v[upto+1] = x;       // φρουρός
      k = from;
      while ( v[k] != x )  k = k+1;
      if ( k <= upto )  fv = k;
                  else  fv = -1;
```



Σχ. 9-4 Πώς δουλεύει η **ταξινόμηση με απ' ευθείας επιλογή**. Υπογραμμισμένες είναι οι τιμές που θα αντιμετατεθούν. Ακόμη, δεξιά από το "[" βλέπεις τις τιμές που έχουν μπεί κι' όλας στη θέση τους. Σε μερικές περιπτώσεις (γραμμές με "**#**") δεν χρειάζεται να γίνει αντιμετάθεση διότι, κατά σύμπτωση, η τιμή του βρίσκεται στη θέση της (*mxp* == *k*).

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
    // (from <= fv <= upto && v[fv] == x) ||
    //         (fv == -1 && (για κάθε j:from..upto " v[j] != x))
    }
    return fv;
} // linSearch
```

9.5.2 Ταξινόμηση Στοιχείων Πίνακα

Τώρα θέλουμε να ταξινομήσουμε[6] τα στοιχεία του σε αύξουσα διάταξη, δηλ. να έχουμε (στα v[0] και v[*N*+1] έχουμε τα "−∞" και "+∞"):

$$\forall j: 1..n \bullet v[j] \le v[j+1]$$

Ενας απλός τρόπος είναι ο εξής:

Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*]
Βρες το μέγιστο από τα στοιχεία v[1]...v[*N*−1] και
αντιμετάθεσε την τιμή του με αυτήν του v[*N*−1]
:
Βρες το μέγιστο από τα στοιχεία v[1]...v[2] και
αντιμετάθεσε την τιμή του με αυτήν του v[2]

Αυτή η διαδικασία περιγράφεται και ως εξής:

```
for (k = N; k >= 2; k = k-1)
{
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k] και
   αντιμετάθεσε την τιμή του με αυτήν του v[k]
}
```

Τελειώσαμε! Η "εντολή":

```
   Βρες το μέγιστο από τα στοιχεία v[1]...v[k]
```

μας είναι γνωστή. Χρησιμοποιώντας τη *maxNdx* που είδαμε στην §9.3 μπορούμε να τη γράψουμε ως εξής:

---

[6] Όπως θα δούμε στη συνέχεια, ο πιο συνηθισμένος λόγος για να ταξινομήσουμε τα στοιχεία ενός πίνακα είναι η ευκολία στο ψάξιμό τους είτε με υπολογιστή είτε με το χέρι. Σκέψου, πώς θα έψαχνες τον τηλεφωνικό κατάλογο αν δεν ήταν ταξινομημένος!

```
        mxp = maxNdx( v, N+2, 1, k );
```

Όσο για την "εντολή":

αντιμετάθεσε την τιμή του με αυτήν του v[*k*]

ξέρουμε να τη γράψουμε:

```
        sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
```

Αυτή η μέθοδος λέγεται **ταξινόμηση με απ' ευθείας επιλογή** (straight selection sort). Στο Σχ. 9-4 βλέπεις πως ταξινομείται με τη διαδικασία αυτή ένας πίνακας με 8 στοιχεία.

Το παρακάτω πρόγραμμα:

- διαβάζει τα στοιχεία του v από το pin.txt,
- τα ταξινομεί με τη μέθοδο που είδαμε,
- γράφει τον ταξινομημένο πίνακα στην οθόνη και
- τον φυλάγει στο αρχείο pinsrt.txt.

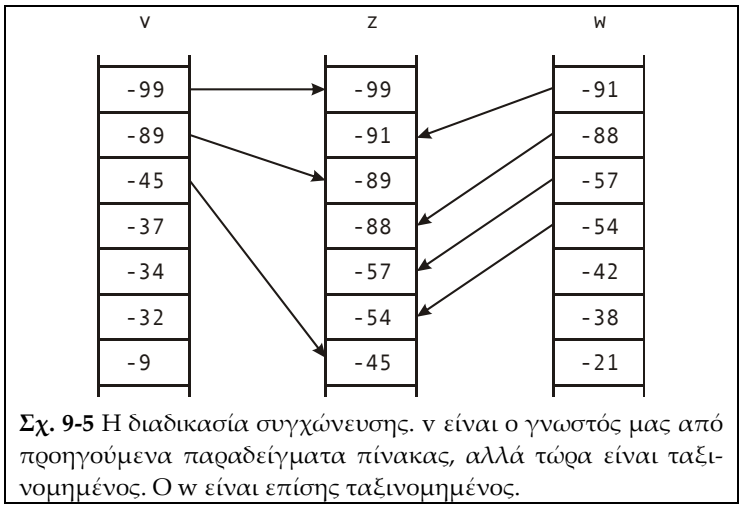
Φυσικά, χρησιμοποιεί τη *maxNdx* που ξέρουμε.

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

   int maxNdx( int x[], int n, int from, int upto );

int main()
{
   const int N( 50 );

   int  v[N+2], sv;
   int  k, mxp, r, c;
   fstream a;

   a.open( "pin.txt", ios_base::in );
// Διάβασε τον πίνακα και βάλε τους φρουρούς όπως παραπάνω

// ταξινόμηση
   for ( k = N; k >= 2; k = k-1 )
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
   } // for

// Δείξε τα στοιχεία του ταξινομημένου πίνακα όπως παραπάνω
// φύλαξε τον πίνακα
   a.open( "pinsrt.txt", ios_base::out );
   for ( k = 1; k <= N; k = k+1 )  a << v[k] << endl;
   a.close();
} // main
```

Να τι θα δούμε στην οθόνη:

```
ΣΤΟΙΧΕΙΑ ΤΟΥ ΠΙΝΑΚΑ v ΜΕΤΑ ΤΗΝ ΤΑΞΙΝΟΜΗΣΗ
     k  v[k]     k  v[k]     k  v[k]     k  v[k]     k  v[k]
     1  -99     11    9     21   40     31  146     41  576
     2  -89     12   10     22   43     32  213     42  576
     3  -45     13   15     23   56     33  232     43  589
     4  -37     14   18     24   57     34  239     44  667
     5  -34     15   21     25   61     35  239     45  753
     6  -32     16   23     26   73     36  344     46  787
     7   -9     17   23     27   80     37  347     47  789
     8    0     18   34     28   99     38  563     48  872
     9    1     19   34     29  122     39  568     49  903
    10    6     20   35     30  139     40  572     50  909
```

Ας «μετρήσουμε» λιγάκι αυτή τη μέθοδο. Όπως φαίνεται από τη **for**, θα έχουμε *N*−1 κλήσεις της *maxNdx* και άλλες τόσες αντιμεταθέσεις τιμών του v. Στη *maxNdx* (*from* = 1, *upto*



Σχ. 9-5 Η διαδικασία συγχώνευσης. v είναι ο γνωστός μας από προηγούμενα παραδείγματα πίνακας, αλλά τώρα είναι ταξινομημένος. Ο w είναι επίσης ταξινομημένος.

= *k*), υπάρχει μια άλλη **for** που, για κάθε τιμή της *k*, εκτελεί, *k* – 1 φορές τη σύγκριση v**[***m***]** > v**[***mxp***]**. Αυτή η σύγκριση θα εκτελεσθεί συνολικώς:

$$(N-1) + (N-2) + \ldots + 2 + 1 = \tfrac{1}{2}N(N-1) = \tfrac{1}{2}N^2 - \tfrac{1}{2}N$$

φορές. Για μεγάλα *N*, ο όρος ½*N* γίνεται αμελητέος και ο όρος που κυριαρχεί είναι ο ½N^2. Λέμε ότι ο χρόνος εκτέλεσης του αλγόριθμου αυξάνεται σαν το N^2. Πάντως υπάρχουν και άλλες πράξεις, που πρέπει να εξετασθούν πιο προσεκτικά. Π.χ. πόσες φορές εκτελείται η `mxp = m`; Για πιο πολύπλοκες δομές στοιχείων θα πρέπει να μετρήσουμε τον υπολογιστικό χρόνο για την εκτέλεση της v**[***m***]** > v**[***mxp***]** και ακόμη της αντιμετάθεσης (εκτελείται *N*–1 φορές).

Αυτή η μέθοδος είναι πολύ απλή στη σύλληψη και για μικρές τιμές της *N* είναι ικανοποιητική.

9.5.3 Συγχώνευση Πινάκων

Έστω ότι έχουμε δυό μονοδιάστατους πίνακες v και w, με στοιχεία του ίδιου τύπου, ας πούμε `int`, με *NV* και *NW* στοιχεία αντιστοίχως. Οι πίνακες αυτοί είναι ταξινομημένοι σε αύξουσα τάξη. Θέλουμε να τους συγχωνεύσουμε σε έναν άλλο πίνακα z, με *NZ* (≥ *NV* + *NW*) στοιχεία του ίδιου τύπου, έτσι ώστε ο z να είναι ταξινομημένος επίσης κατ' αύξουσα διάταξη.

Μια πρώτη σκέψη (και η χειρότερη) που θα μπορούσαμε να κάνουμε είναι η εξής: Να βάλουμε στις πρώτες *NV* θέσεις του πίνακα z τα στοιχεία του v, στις επόμενες *NW* θέσεις τα στοιχεία του w και στη συνέχεια να ταξινομήσουμε τον z όπως είδαμε στην προηγούμενη παράγραφο. Με αυτόν τον τρόπο, δεν εκμεταλλευόμαστε το γεγονός ότι οι δύο πίνακες είναι ήδη ταξινομημένοι. Ο υπολογιστικός χρόνος θα είναι ανάλογος του $(NV+NW)^2$, ενώ ήδη έχει καταναλωθεί χρόνος για να ταξινομηθούν οι v και w. Θα πρέπει να ψάξουμε για κάτι καλύτερο.

Για δες την παρακάτω απλή ιδέα:

```
Πάρε την πρώτη τιμή από τον v
Πάρε την πρώτη τιμή από τον w
Ετοιμάσου να γράψεις στην πρώτη θέση του z
while ( δεν έφτασες στο τέλος ούτε του v ούτε του w )
{
  if ( τιμή από τον v < τιμή από τον w )
  {
    Γράψε στον z την τιμή από τον v
    Αντικατάστησε την τιμή από τον v με την επόμενή της
  }
  else   // if τιμή από τον v >= τιμή από τον w
  {
```

```
    Γράψε στον z την τιμή από τον w
    Αντικατάστησε την τιμή από τον wμε την επόμενή της
  }
  Ετοιμάσου να γράψεις στην επόμενη θέση του z
}
```

Στο Σχ. 9-5, βλέπεις πως δουλεύει αυτή η μέθοδος.

Αλλά δεν τελειώσαμε ακόμη: Η εκτέλεση της **while** θα τελειώσει όταν φτάσουμε στο τέλος είτε του v είτε του w. Στη συνέχεια θα πρέπει να αντιγράψουμε στον z τα στοιχεία του άλλου πίνακα, που δεν τελείωσε. Και αν τελειώσουν και οι δυο μαζί; Δεν γίνεται! *Σε κάθε εκτέλεση της περιοχής επανάληψης παίρνουμε ένα στοιχείο από έναν πίνακα*. Θα πρέπει να συμπληρώσουμε τον αλγόριθμό μας με το κομμάτι:

```
Πρόσθεσε στον z τα στοιχεία που περίσσεψαν από τον πίνακα που δεν εξαντλήθηκε
```

Θα γράψουμε ένα πρόγραμμα που θα διαβάζει, όπως ξέρουμε, τις τιμές των στοιχείων του πίνακα v, από το αρχείο pinsrt.txt, όπου τα γράψαμε με το πρόγραμμα της προηγούμενης παραγράφου, μετά την ταξινόμηση. Θα διαβάζει ακόμη, με τον ίδιο τρόπο, τις τιμές των (20) στοιχείων του πίνακα w που βρίσκονται στο αρχείο pinw.dta, ήδη ταξινομημένα. Στη συνέχεια θα συγχωνεύει τα στοιχεία των δύο πινάκων στον z.

Τρεις μεταβλητές *dv*, *dw*, *dz* θα παίζουν ρόλο δεικτών στους v, w, z αντιστοίχως. Οι *dv*, *dw* θα δείχνουν τα στοιχεία των v και w που συγκρίνουμε, ενώ ο *dz* δείχνει το στοιχείο του z όπου θα γίνει η αντιγραφή. Και οι τρεις θα αρχίζουν από 1:

```
   dv = 1;  dw = 1;  dz = 1;
```

Η while που δώσαμε παραπάνω γράφεται:

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
```

Αν, στο τέλος, τελείωσαν τα στοιχεία του v (`dv > NV`) και έχουν περισσέψει στοιχεία από τον w, τα αντιγράφουμε στον z ως εξής:

```
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
```

Πρόσεξε ότι οι αρχικές τιμές των δεικτών *dz* και *dw* γι' αυτήν τη while είναι ακριβώς οι τελικές τιμές τους από την προηγούμενη. Παρόμοια προσθέτουμε στον z και τα στοιχεία του v, αν έχει εξαντληθεί ο w. Μετά από τις πιό πάνω παρατηρήσεις μπορούμε να δώσουμε ολόκληρο το πρόγραμμα.

```
#include <fstream>
using namespace std;

int main()
{
   const int N( 50 ), NV( N ), NW( 20 ), NZ( 70 );

   int  v[N+2], w[NW+2], z[NZ+2];
   int  k;
   int  dv, dw, dz;
   fstream a;

   a.open( "pinsrt.txt", ios_base::in );
   for (k = 1; k <= NV; k = k+1)  a >> v[k];
   a.close();
   a.open( "pinw.txt", ios_base::in);
   for (k = 1; k <= NW; k = k+1)  a >> w[k];
   a.close();

// ΣΥΓΧΩΝΕΥΣΗ
   dv = 1;  dw = 1;  dz = 1;
```

```
   while ( dv <= NV && dw <= NW )
   {
      if (v[dv] < w[dw])  {  z[dz] = v[dv];  dv = dv + 1;  }
                    else  {  z[dz] = w[dw];  dw = dw + 1;  }
      dz = dz + 1;
   } // while
   if ( dv > NV )
      while ( dw <= NW )
      {  z[dz] = w[dw];  dw = dw + 1;  dz = dz + 1;  }
   else
      while ( dv <= NV )
      {  z[dz] = v[dv];  dv = dv + 1;  dz = dz + 1;  }

   a.open( "pinz.txt", ios_base::out );
   for ( k = 1; k <= NZ; k = k+1 )  a << z[k] << endl;
   a.close();
} // main
```

Μπορείς εύκολα να δείς ότι ο χρόνος εκτέλεσης αυτού του αλγόριθμου είναι ανάλογος του *NV*+*NW*.

9.6 Ταχύτερα – Οικονομικότερα – Καλύτερα

Τώρα ας γυρίσουμε για να ξαναδούμε τα προηγούμενα προβλήματά μας.

Πρώτα η αναζήτηση. Αλλάζουμε λίγο το πρόβλημά μας: Έστω π.χ. ότι έχουμε έναν μονοδιάστατο πίνακα, v, με *N* στοιχεία τύπου **int**, *ταξινομημένα κατ' αύξουσα τάξη* και μια τιμή *x* (επίσης τύπου **int**). Θέλουμε έναν αλγόριθμο που θα ψάχνει στα στοιχεία v[*from*], v[*from*+1], ..., v[*upto*] να βρει την τιμή *x*.

Οι δυο παραλλαγές γραμμικής αναζήτησης που είδαμε δουλεύουν μια χαρά. Αλλά παίρνοντας υπόψη μας την ταξινόμηση μπορούμε να τα καταφέρουμε κάπως καλύτερα. Και μάλιστα στη χειρότερη περίπτωση: όταν η τιμή που ψάχνουμε δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν, οι αλγόριθμοι που είδαμε σαρώνουν ολόκληρον τον πίνακα πριν αποφανθούν.

Αν ο πίνακας είναι ταξινομημένος, δεν χρειάζεται να τον ψάξουμε ολόκληρον. Ας ξεκινήσουμε από τον τηλεφωνικό κατάλογο. Ψάχνουμε το τηλέφωνο του "**ΠΑΠΑΔΗΜΑ**", αλλά μετά το όνομα "**ΠΑΠΑΔΑΝΙΗΛ**" βρίσκουμε το όνομα "**ΠΑΠΑΔΙΑΚΟΣ**". Φυσικά, σταματάμε το ψάξιμο!

Έστω, λοιπόν, ότι v[*from*] ≤ *x* ≤ v[*upto*]. Θα ψάχνουμε μέχρι να βρούμε κάποιο v[*k*] ≥ *x*:

- Αν έχουμε v[*k*] == *x* τότε βρήκαμε την τιμή.
- Αλλιώς, αν v[*k*] > *x*, δεν έχει νόημα να συνεχίσουμε την αναζήτηση αφού όλα τα επόμενα στοιχεία έχουν τιμές μεγαλύτερες από το v[*k*], άρα και από το *x*.

```
      k = from;
      while ( v[k] < x )  // I: ∀j:from-1..k-1 • v[j] < x
         k = k+1;
   // (∀j:0..k-1 • v[j] < x) && (x ≤ v[k])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Για να τερματίζεται η **while** και στην περίπτωση που *x* > v[*upto*] θα πρέπει να έχουμε φρουρό (+∞) στο v[*upto*+1]. Ο φρουρός (−∞) στο v[*from*−1] χρειάζεται για να έχει νόημα η αναλλοίωτή μας όταν *m* == *from*, αλλά δεν είναι απαραίτητη η φυσική του παρουσία.

Αν θέλεις, μπορείς να ψάξεις και από το τέλος προς την αρχή:

```
      k = upto;
      while ( x < v[k] )  // I: ∀j:k+1..upto+1 • x < v[j]
         k = k-1;
   // (v[k] ≤ x) && (∀j:k+1..upto+1 • x < v[j])
      if ( v[k] == x )  fv = k;
                  else  fv = -1;
```

Στην περίπτωση αυτή, θα πρέπει να έχουμε φρουρό (−∞) στο v[*from*−1] για να τερματιζεται η **while** και στην περίπτωση που *x* < v[*from*]. Εδώ, δεν είναι απαραίτητη η φυσική του παρουσία φρουρού (+∞) στο v[*upto*+1], αλλά πρέπει να υποθέσουμε ότι υπάρχει για να έχει νόημα η αναλλοίωτή μας όταν *m* == *upto*.

Τώρα ας δούμε κάτι καλύτερο.

Έψαξες ποτέ σου τον τηλεφωνικό κατάλογο γραμμικώς; Μάλλον απίθανο. Συνήθως, αν ψάχνεις για τον ΠΑΠΑΔΗΜΑ, ανοίγεις τον κατάλογο εκεί που μαντεύεις ότι είναι το 'Π'. Αν πέσεις στο 'Σ', γυρνάς πιο πίσω. Αν τώρα πέσεις στο 'Ο' ξαναδοκιμάζεις πιο μπροστά κ.ο.κ. Ας προσπαθήσουμε να κάνουμε κάτι παρόμοιο και στον πίνακά μας.

Αρχίζουμε ελέγχοντας το στοιχείο που βρίσκεται στο μέσο του πίνακα, έχει δηλαδή δείκτη *middle* = (*from*+*upto*)/2. Αν v[*middle*] == *x*, βρήκαμε την τιμή που ψάχνουμε, τελειώσαμε. Αν v[*middle*] < *x* τότε, αφού ο πίνακας είναι ταξινομημένος, όλα τα στοιχεία v[*from*],... v[*middle*], είναι σίγουρα < *x*. Θα πρέπει λοιπόν να ψάξουμε στα: v[*middle*+1],... v[*upto*]. Αν βρίσκαμε v[*middle*] > *x* θα έπρεπε να συνεχίζαμε το ψάξιμο στην περιοχή v[*from*],... v[*middle*−1]. Και πώς θα συνεχιστεί το ψάξιμο; Μα με τον ίδιο τρόπο! Αλλά στον μισό πίνακα.

Δες το παράδειγμα στο Σχ. 9-4, όπου σε ένα ταξινομημένο πίνακα 8 στοιχείων ψάχνουμε για την τιμή 100.

Βάζουμε αρχικά, "`from = 1; upto = 8;`" και υπολογίζουμε το *middle* = 4: v[*middle*] = 57. Το 100 –αν υπάρχει– θα βρίσκεται μετά το 57. Συνεχίζουμε με τον ίδιο τρόπο, αφού αλλάξουμε το "`from = middle + 1;`" (= 5). Υπολογίζουμε και πάλι το *middle* = 6. Τώρα έχουμε: v[*middle*] = 239. Το 100 αν υπάρχει, θα υπάρχει πριν από το 239. Τώρα, πριν συνεχίσουμε, αλλάζουμε το "`upto = middle - 1;`" (= 5). Η περιοχή που ψάχνουμε περιορίστηκε σε ένα στοιχείο. Αλλά ας συνεχίσουμε: *middle* = 5, v[*middle*] = 213. Αν υπάρχει το 100, θα υπάρχει πριν από αυτό. Αλλάζουμε λοιπόν το "`upto = middle - 1;`" (= 4). Και εδώ έχουμε το «περίεργο φαινόμενο»: το άνω όριο της περιοχής αναζήτησης (*upto* = 4) να είναι μικρότερη από το κάτω όριο (*from* = 5). Δεν υπάρχει πια περιοχή αναζήτησης. Καιρός να σταματησουμε διότι το 100 δεν υπάρχει στον πίνακά μας.

Τελειώνουμε λοιπόν το ψάξιμο:

- όταν βρούμε το *x* (v[*middle*] == *x*) ή
- όταν η περιοχή που ψάχνουμε δεν υπάρχει πια (*from* > *upto*).

```
middle = (from + upto) / 2;
while ( from <= upto && v[middle] != x )
{
   if (v[middle] < x)  from = middle + 1;
   else if (v[middle] > x)  upto = middle - 1;
   middle = (from + upto) / 2;
}  // while
```

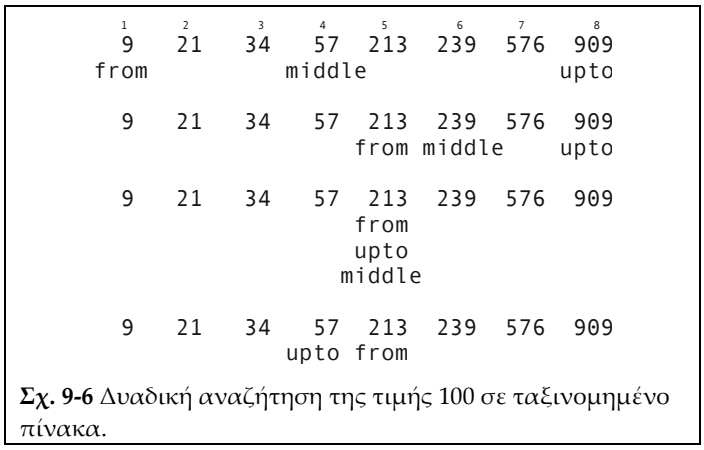
Μετά το τέλος της επαναληπτικής διαδικασίας, θα πρέπει να ελέγξουμε για ποιο λόγο σταμάτησε:

```
if (v[middle] == x)  η x υπάρχει στο Meso
```

Αυτός είναι ο αλγόριθμος **δυαδικής αναζήτησης** (binary search).

Όπως είδες και από το παράδειγμα, οι περισσότερες επαναλήψεις θα γίνουν στην περίπτωση που η τιμή *x* δεν υπάρχει στον πίνακα. Στην περίπτωση αυτήν το ψάξιμο πρέπει να συνεχιστεί ωσότου συναντηθούν οι δείκτες *from* και *upto* (που τους λέμε αντιστοίχως *κάτω δείκτη* και *άνω δείκτη* της περιοχής που ψάχνουμε). Ας πούμε ότι, αρχικά, η περιοχή που ψάχνουμε έχει *N* στοιχεία (*from* = 1 και *upto* = *N*). Όταν οι δείκτες συναντηθούν η περιοχή θα έχει λιγότερα από 1 στοιχεία (*upto* < *from*). Το μήκος της περιοχής μικραίνει με υποδιπλασιασμό (περίπου, γιατί έχουμε ακέραιη διαίρεση). Μετά από *l* επαναλήψεις, η περιοχή που ψάχνουμε θα έχει περίπου:

$$N / 2^l \text{ στοιχεία.}$$



Σχ. 9-6 Δυαδική αναζήτηση της τιμής 100 σε ταξινομημένο πίνακα.

Αυτός ο αριθμός γίνεται μικρότερος από 1 όταν το 2^l γίνει μεγαλύτερο από N, ή:

$$l > log_2 N$$

Έχουμε λοιπόν έναν αλγόριθμο με *λογαριθμική* περιπλοκότητα χρόνου, αντί για την γραμμική (ανάλογη του N) που έχουν οι προηγούμενοι. Τι σημαίνει αυτό; Αν π.χ. πάρουμε $N = 1024$, τότε στη χειρότερη περίπτωση θα έχουμε 10 επαναλήψεις ($log_2 1024 = 10$) της **while**, ενώ με το γραμμική αναζήτηση θα έχουμε 1024!

Η συνάρτηση *binSearch()*, που υλοποιεί τη μέθοδο δυαδικής αναζήτησης, θα είναι διαφορετική από τη *linSearch()* ως προς το εξής: Θα επιστρέφει πάντοτε μια τιμή δείκτη!

Έστω ότι έχουμε:

- πίνακα που δεν είναι «γεμάτος» αλλά έχει τιμές, ταξινομημένες κατ' αύξουσα τάξη, στις θέσεις από `v[1]` μέχρι `v[`*last*`]`, όπου *last* < *N* και
- μια τιμή *x* που πρέπει να εισαχθεί στον πίνακα ώστε να παραμείνει ταξινομημένος.

Η *binSearch()* θα μας δίνει τη θέση *k* στην οποία θα πρέπει να εισαχθεί η *x* διότι όταν σταματησει η **while** θα έχουμε $v[k-1] < x \le v[k]$.

Δες ένα χαρακτηριστικό παράδειγμα χρήσης της *binSearch*:

```
   ndx = binSearch( v, last, x );
   if ( v[ndx] == x )
      cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   else
   {
      for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
      v[ndx] = x;
      last = last + 1;
      cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
   }
```

Από την τιμή που μας επιστρέφει η συνάρτηση –και φυλάγουμε στην *ndx*– δεν ξέρουμε αν βρέθηκε η *x* στιον πίνακα· χρειάζεται και ο έλεγχος

```
   if ( v[ndx] == x )
```

Πρόσεξε πώς γίνεται η εισαγωγή της νέας τιμής στον πίνακα: Με τη **for** «ανοίγουμε χώρο» για να τη εισαγάγουμε.

Να πώς θα είναι η *binSearch*:

```
// binSearch -- Δυαδική αναζήτηση στα στοιχεία v[1],...v[last]
//              για να βρεθεί θέση fv τέτοια ώστε:
//                       v[fv-1] < x <= v[fv]
//              Επιστρέφει την fv.
//              Ο v πρέπει να είναι ταξινομημένος κατ' αύξουσα
//              τάξη με φρουρούς στα άκρα
//              v[0] == -inf,  v[last+1] == +inf
unsigned int binSearch( const int v[], int last, int x )
{
   unsigned int l( 1 ), r( last+1 );
   unsigned int middle;
```

```
   while ( l < r ) // I: (∀k:[0..l) • v[k] < x) &&
   {               //    (∀k:[r..last+1) • v[k] >= x)
      middle = (l + r) / 2;
      if ( v[middle] < x )  l = middle + 1;
                    else  r = middle;
   }  // while
// v[l-1] < x <= v[l]
   return l;
}  // binSearch
```

Στη συνέχεια δίνουμε μια (όχι και πολύ αυστηρή) απόδειξη ορθότητας της *binSearch*.

Γιατί δεν βάλαμε και εδώ παραμέτρους (*from*, *upto*) που θα μας δίνουν δυνατότητα αναζήτησης σε τμήμα του πίνακα; Αυτή η δυνατότητα έρχεται σε σύγκρουση με τη δυνατότητα να μας δίνεται η θέση εισαγωγής. Δες ένα παράδειγμα. Ας πούμε ότι έχουμε:

0	1	2	3	4	6	7	8	9	10	11	12	13	14	15	16	17
-∞	2	7	11	15	17	19	23	29	31	37	41	44	53	+∞		

και αναζητούμε το 12 με *from* = 7 και *upto* = 12. Αφού δεν υπάρχει θα μας υποδειχθεί η εισαγωγή στη θέση *from* = 7 αλλά αυτό είναι λάθος. Παρομοίως, θα μας υποδειχθεί η εισαγωγή του 61 στη θέση *upto*+1 = 13. Οι υποδείξεις για τη θέση εισαγωγής είναι σωστές μόνον αν v[*from*-1] < *x* < v[*upto*+1].

Δοκιμάζουμε τη *binSearch* με ένα πρόγραμμα που έχει τις εντολές αναζήτησης – εισαγωγής (που είδαμε παραπάνω) σε έναν ταξινομημένο πίνακα που διαβάζουμε από το pinsrt.txt:

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;

unsigned int binSearch( const int v[], int n, int x );

int main()
{
   const unsigned int N( 60 );

   int  v[N];               // ο πίνακας που ψάχνουμε
   unsigned int last(N-10); // με τιμές στις θέσεις v[1]...v[last]
   int  x;                  // για την τιμή της ψάχνουμε
   unsigned int ndx;        // εδώ θα είναι αν τη βρούμε
   int  k;
   ifstream a( "pinsrt.txt" );

   for ( k = 1; k <= last; k = k+1 )
      a >> v[k];                              // Διάβασε τον πίνακα
   a.close();
   v[0] = INT_MIN;  v[last+1] = INT_MAX;      // βάλε τους φρουρούς

   cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   while ( x != 9999 )
   {
      ndx = binSearch( v, last, x );
      if ( v[ndx] == x )
         cout << " ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      else
      {
         for ( k = last+1; k >= ndx; --k )  v[k+1] = v[k];
         v[ndx] = x;
         last = last + 1;
         cout << " ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: " << ndx << endl;
      }
      cout << "ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): ";  cin >> x;
   } // while
} // main
```

Τώρα, κρατούμε για τον πίνακα 60 θέσεις αλλά τον φορτώνουμε στις θέσεις 1..50. Στις θέσεις 0 και 51 μπαίνουν οι φρουροί.

```
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΕΙΣΑΧΘΗΚΕ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 40
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 21
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 43
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 23
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 41
ΤΗ ΒΡΗΚΑ ΣΤΗ ΘΕΣΗ: 22
ΔΩΣΕ ΤΗΝ ΤΙΜΗ Χ (9999 για ΤΕΛΟΣ): 9999
```

Παρατήρηση: ▶

Εδώ θα πρέπει να σταματήσουμε για λίγο και να σκεφτούμε αυτά που είδαμε. Η δυαδική αναζήτηση είναι πολύ εντυπωσιακή στην ταχύτητά της αλλά δεν είναι θαύμα. Είναι γρήγορη, διότι ο πίνακας μας έχει υποστεί μια προεργασία, *έχει ταξινομηθεί*. Ας πούμε λοιπόν ότι θέλεις να κάνεις ένα σύστημα, όπου θα πρέπει θα ψάχνεις έναν πίνακα και να δίνεις την απάντησή σου *γρήγορα*. Θα πρέπει να χρησιμοποιήσεις μια γρήγορη μέθοδο, όπως είναι η δυαδική αναζήτηση. Ενώ όμως απαιτείται ταχύτητα στην αναζήτηση, πολύ συχνά, έχεις αρκετή άνεση χρόνου για να προετοιμάσεις κατάλληλα το σύστημά σου: να ταξινομήσεις τον πίνακά σου ή να ενημερώνεις τον πίνακά σου ώστε να είναι ταξινομημένος – διαδικασίες σχετικά χρονοβόρες. Με αυτήν τη φιλοσοφία σχεδιάζονται πολλά πληροφοριακά συστήματα σήμερα και ένα πολύ κοινό παράδειγμα είναι ο τηλεφωνικός κατάλογος. Όταν τον ψάχνεις, η αλφαβητική ταξινόμηση σου είναι απαραίτητη και την έχεις. Έτσι, έχεις τη δυνατότητα να ψάχνεις γρήγορα. Η ταξινόμηση γίνεται πριν τυπωθεί, όταν η υπηρεσία που τον ετοιμάζει έχει όλη τη χρονική άνεση να κάνει κάτι τέτοιο. ◀

Και τώρα, ας ξανάρθουμε στην ταξινόμηση. Ο αλγόριθμος που είδαμε απαιτεί, για μεγάλα N, χρόνο λN^2. Πρόσεξε τώρα το εξής:

- Κόβουμε τον πίνακά μας στη μέση. Έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε έναν από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικώς: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

Με τις διαδικασίες που γράψαμε αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα από αυτό, δεν μπορούμε να μη σκεφτούμε: «γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log N$, που φυσικά είναι καλύτερος από N^2. Αυτός ο αλγόριθμος χρειάζεται διπλή μνήμη από αυτήν που απαιτείται για την αποθήκευση του πίνακα. Αργότερα θα μάθεις ότι υπάρχουν αλγόριθμοι ταξινόμησης με χρόνο $N\log N$, χωρίς υπερβολικές απαιτήσεις μνήμης.

Αυτή είναι μια περίπτωση εφαρμογής της αρχής «διαίρει και βασίλευε» (divide and conquer) στον αλγόριθμό μας: Κόβουμε τον πίνακα στα δυο και λύνουμε το αρχικό πρόβλημα (ταξινόμηση) στα δυο κομμάτια.

9.6.1 Απόδειξη Ορθότητας της *binSearch*

Η απόδειξη ορθότητας θα γίνει με τη μέθοδο της επαγωγής.

Αρχικώς, στην [0..*l*) υπάρχει μόνον ο φρουρός "`-∞`" και ισχύει η "`-∞ <` x". Η δεξιά περιοχή είναι κενή: [*last*+1..*last*+1) και η ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x είναι (τετριμμένως) αληθής.

Αν v`[`*middle*`] <` x βάζουμε `l = middle + 1`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`] <` x θα ισχύει για κάθε *k* στο [0..*middle*] ή στο [0..*middle*+1) που είναι το [0..*l*). Για την ∀*k*: [*r*..*last*+1) • v`[`*k*`]` ≥ x δεν έχουμε αλλαγή.

Αν v`[`*middle*`]` ≥ x βάζουμε `r = middle`. Λόγω της ταξινόμησης του πίνακα η v`[`*k*`]` ≥ x θα ισχύει για κάθε *k* στο [*middle*..*last*] ή στο [*middle*..*last*+1) που είναι το [*r*..*last*+1). Για την ∀*k*: [0..*l*) • v`[`*k*`] <` x δεν έχουμε αλλαγή.

Μετά τη **while** θα έχουμε:

$$(l >= r)\ \&\&\ (\forall k:\ [0..l) \bullet v[k] < x)\ \&\&\ (\forall k:\ [r..last+1) \bullet v[k] \geq x)$$

Αφού $l \in [r..last+1)$ θα έχουμε: v`[`l`]` ≥ x. Παρομοίως, αφού l-1 ∈ [0..l) θα έχουμε: v`[`l-1`] <` x. Δηλαδή θα ισχύει η v`[`l-1`] <` $x \leq$ v`[`l`]`.

Για την απόδειξη του τερματισμού εξετάζουμε τη διαφορά $d = r - l$ που έχει θετική τιμή (από τη συνθήκη της **while**).

Από την $l <$ `r` παίρνουμε $l \leq$ *middle* `<` r (το "`=`" αριστερά χρειάζεται διότι η (*l*+*r*)`/`2 είναι ακέραιη διαίρεση.)

- Αν εκτελεσθεί η `l = middle + 1` η νέα τιμή διαφοράς $d' = r - (l+r)/2 - 1 < d$.
- Αν εκτελεσθεί η `r = middle` η νέα τιμή διαφοράς $d' = (l+r)/2 - l < d$.

Επομένως η $r - l$ παράγει –όσο εκτελείται η **while**– μια γνησίως φθίνουσα ακολουθία θετικών ακεραίων που δεν είναι δυνατόν να έχει άπειρο πλήθος όρων.

9.7 Ανακεφαλαίωση

Ένας πίνακας είναι μια ακολουθία τιμών ίδιου τύπου, όπως και ένα σειριακό αρχείο, αλλά:

- Ο πίνακας υλοποιείται στην κύρια μνήμη ενώ το αρχείο στη βοηθητική.
- Μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε στοιχείο του πίνακα με την ίδια ευκολία, ενώ στο σειριακό αρχείο για να πάμε στο ν-οστό στοιχείο πρέπει να περάσουμε από τα προηγούμενα $\nu - 1$.[7]
- *Μπορούμε να χειριστούμε οποιοδήποτε στοιχείο ενός πίνακα όπως μια μεταβλητή* ενώ για να χειριστούμε μια τιμή από ένα αρχείο θα πρέπει να τη φέρουμε στην κύρια μνήμη.

Τα ν στοιχεία ενός πίνακα αριθμούνται από 0..ν–1. Όλα τα στοιχεία του πίνακα έχουν το ίδιο όνομα και διακρίνονται μεταξύ τους από τον *δείκτη* που είναι ο αριθμός του στοιχείου μέσα σε αγκύλες.

Οι πίνακες χρησιμοποιούνται φυσικά για την επίλυση με ΗΥ μαθηματικών προβλημάτων όπου χρησιμοποιούνται πίνακες. Πέρα από αυτό όμως οι πίνακες διευκολύνουν τον προγραμματισμό σε περιπτώσεις που έχουμε πολλά ομοειδή (ίδιου τύπου) αντικείμενα που υφίστανται την ίδια επεξεργασία.

Η αναζήτηση τιμών σε έναν πίνακα είναι ένα πολύ συνηθισμένο πρόβλημα. Η ταχύτητα της αναζήτησης αυξάνεται σημαντικά αν τα δεδομένα μας είναι ταξινομημένα. Έτσι, έχουν επινοηθεί πολλοί αλγόριθμοι για ταξινόμηση.

---

[7] Αργότερα θα δούμε ότι μπορούμε να έχουμε και αρχεία *τυχαίας πρόσβασης*.

Ασκήσεις

Α Ομάδα

9-1 Έστω ένας πίνακας

```
double a[10];
```

Γράψε εντολές που θα τον «αναποδογυρίσουν». Δηλαδή να αντιμεταθέσουν τις τιμές των στοιχείων του:

(a[0] ↔ a[9]) (a[1] ↔ a[8]) (a[2] ↔ a[7]) (a[3] ↔ a[6]) (a[4] ↔ a[5])

9-2 Ένας τρόπος να δούμε «πόσο μεγάλος» είναι ένας μονοδιάστατος πίνακας A, με στοιχεία από 0 μέχρι $N - 1$, είναι να υπολογίσουμε το:

$$\|A\| = |A_0| + |A_1| + \ldots + |A_{N-1}|$$

Γράψε πρόγραμμα που θα διαβάζει τις τιμές των στοιχείων του A ($N = 5$) και θα υπολογίζει και θα τυπώνει το $\|A\|$.

Στη συνέχεια, αν $\|A\| \neq 0$, θα κάνει τον A *μοναδιαίο*, διαιρώντας όλα τα στοιχεία του δια $\|A\|$. Τέλος, θα τυπώνει τα στοιχεία του μοναδιαίου πίνακα.

Επανάλαβε τα παραπάνω αν: $\|A\| = \sqrt{A_0^2 + A_1^2 + \ldots + A_{N-1}^2}$.

Το ίδιο αν: $\|A\| = \max_{k=0..N-1} (|A_k|)$.

Τα μαθηματικά ονομάζουν τα τρία αποτελέσματα νόρμα-1, νόρμα-2 και νόρμα-∞ αντίστοιχα.

9-3 Λύσε ξανά την προηγούμενη άσκηση αφού γράψεις τρεις συναρτήσεις *norm*1, *norm*2 και *normInf* που υπολογίζουν τις τρεις νόρμες.

9-4 Τροποποίησε το πρόγραμμα συγχώνευσης για την περίπτωση που ο πίνακας w δίνεται ταξινομημένος σε *φθίνουσα* τάξη. **Προσοχή!** *Μόνον* ο w.

9-5 Γράψε μια

```
double vectorSumIf( const double x[], int n,
                    double lb, double ub )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία: $lb \le x[k] \le ub$.

Β Ομάδα

9-6 Έστω ότι έχουμε δύο πίνακες x, y με n στοιχεία. Λέμε ότι ο x *προηγείται λεξικογραφικά* του y αν υπάρχει κάποιο k, από 0 μέχρι $n-1$, τέτοιο ώστε να έχουμε: $x_k < y_k$ και $x_j == y_j$ για κάθε j από 0 μέχρι $k-1$.

Γράψε μια:

```
bool lt( int x[], int y[], int n)
```

που θα επιστρέφει τιμή:

- `true` αν ο x προηγείται λεξικογραφικά του y και
- `false` αν όχι.

9-7 *Μέθοδος των ελαχίστων τετραγώνων*. Παραλλαγή της Άσκ. 6-11. Γράψε πρόγραμμα που θα διαβάζει από ένα αρχείο text, με όνομα στο δίσκο lsq.txt, τα (x_k, y_k), $k = 0 .. N-1$ και θα υπολογίζει τα α και β. Σε κάθε γραμμή του αρχείου έχουμε ένα ζεύγος x_k, y_k. Υποθέτουμε το N δεν υπερβαίνει το 100. Οι τιμές των x_k, y_k που διαβάζονται θα αποθηκεύονται σε δύο πίνακες x, y με χώρο για 100 στοιχεία το πολύ.

Ένα μέτρο του πόσο καλή είναι η προσαρμογή ευθείας στα δεδομένα δίνεται από τον **συντελεστή πολλαπλής συσχέτισης** (multiple correlation coefficient) R^2 που ορίζεται ως εξής:

$$R^2 = \frac{\sum_{k=0}^{N-1}(\alpha x_k + \beta - < y >)^2}{\sum_{k=0}^{N-1}(y_k - < y >)^2} \quad \text{όπου: } <y> = \frac{1}{N}\sum_{k=0}^{N-1} Y_k$$

Η τιμή του R^2 είναι από 0 (τέλεια προσαρμογή) μέχρι 1. Συμπλήρωσε λοιπόν το πρόγραμμά σου ώστε μετά τον υπολογισμό των α και β να υπολογίζει και να γράφει το R^2.

***9-8** Απόδειξε ότι η *maxNdx* είναι σωστή, δηλαδή

```
// 0 <= from <= upto <= n-1
   mxp = from;
   m = from+1;
   while ( m <= upto ) )
   {
      if ( x[m] > x[mxp] )  mxp = m;
      m = m + 1;
   }
// from ≤ mxp ≤ upto && ∀j: from..upto • x[j] ≤ x[mxp]
```

Υπόδ.: Απόδειξε ότι η

$$from \le mxp \le m - 1 \;\&\&\; \forall j: from\,..\,m-1 \bullet \mathtt{x[}j\mathtt{]} \le \mathtt{x[}mxp\mathtt{]}$$

είναι αναλλοίωτη της **while**.

9-9 Γράψε μια:

```
int linSearchSrt( const int v, int n, int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον *ταξινομημένο*, σε αύξουσα τάξη, πίνακα v για την τιμή *x*. Απόδειξε ότι είναι σωστή.

9-10 Γράψε μια:

```
unsigned int linSearchMult( const int v, int n,
                            int from, int upto, int x )
```

που θα ψάχνει *γραμμικώς* στον πίνακα v –που υλοποιεί ένα *πολυσύνολο*– για τη *x* και θα επιστρέφει το *πλήθος* των στοιχείων του v που είναι ίσα με τη *x*.

9-11 Απόδειξε ότι το πρόγραμμα ταξινόμησης είναι σωστό, δηλαδή (στα `v[0]` και `v[`N`+1]` έχουμε τα $-\infty$ και $+\infty$)[8]:

```
// N > 0
   k = N;
   while ( k >= 2 ) // ∀j: k..N • v[j] ≤ v[j+1]
   {
      mxp = maxNdx( v, N+2, 1, k );
      sv = v[mxp];  v[mxp] = v[k];  v[k] = sv;
      k = k - 1;
   } // for
// ∀j: 1..N • v[j] ≤ v[j+1]
```

Γ Ομάδα

9-12 Έλυσες την Άσκ. 7-15; Λύσε τώρα και το πραγματικό πρόβλημα: Γράψε συνάρτηση

```
double mp( double a[], int N, double x )
```

που να υλοποιεί την παρακάτω συνάρτηση:

[8] Για να είναι σωστός ο αλγόριθμος θα πρέπει να αποδείξουμε ότι δεν κάνουμε εισαγωγή ή διαγραφή τιμών. Αφού όμως ο αλγόριθμός μας κάνει αντιμεταθέσεις τιμών στοιχείων μπορούμε να θεωρήσουμε ότι αυτό είναι εξασφαλισμένο.

$$m_p(x) = \prod_{k=0}^{N-1} \frac{1}{x-a_k} = \frac{1}{x-a_0} \times \frac{1}{x-a_1} \times \ldots \times \frac{1}{x-a_{N-1}}, \text{ όπου } N \geq 1$$

9-13 Έστω ότι έχουμε έναν πίνακα:

```
const int N = 50;

int v[N+2];
```

Στα v[0] και v[N+2] έχουμε τα $-\infty$ και $+\infty$ αντιστοίχως.

Στον πίνακα έχουμε ήδη τιμές στις θέσεις από v[1] μέχρι v[l] ($l < N$), ταξινομημένες κατ' αύξουσα τάξη. Θέλουμε να εισαγάγουμε μία ακόμη τιμή –της μεταβλητής x– στη θέση v[m] όπου $m \leq l + 1$, έτσι ώστε:

- να μη χάσουμε κάποια από αυτές που ήδη υπάρχουν,
- ο πίνακας να συνεχίσει να είναι ταξινομημένος κατ' αύξουσα τάξη.

9-14 (Συνέχεια της προηγούμενης) Με βάση τα παραπάνω μπορούμε να δούμε μιαν άλλη μέθοδο ταξινόμησης ενός πίνακα: την μέθοδο **κατ' ευθείαν εισαγωγής** (straight insertion sort):

```
for ( i = 2; i <= N; i = i+1 ) //I: το v[1]…v[i-1] ταξινομημένο
{
   Βάλε το v[i] στη σωστή θέση στο κομμάτι v[1]…v[i-1]
} // for
```

Άλλαξε το πρόγραμμα της ταξινόμησης ώστε να υλοποιεί αυτήν τη μέθοδο.

9-15 Μας δίνονται δύο αρχεία text, int1.txt και int2.txt, που περιέχουν ακέραιους αριθμούς ταξινομημένους κατ' αύξουσα τάξη. Γράψε πρόγραμμα που θα συγχωνεύσει τα περιεχόμενα των δύο αρχείων σε ένα (int3.txt) ταξινομημένο κατ' αύξουσα τάξη.

κεφάλαιο

10

Προγράμματα με Κείμενα

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να γράφεις προγράμματα που διαχειρίζονται μη-αριθμητικά δεδομένα. Να χρησιμοποιείς τις τεχνικές αυτές και σε αριθμητικά δεδομένα.

Προσδοκώμενα αποτελέσματα:

Είναι μάλλον απίθανο να γράφεις «πραγματικό» πρόγραμμα χωρίς να χρειαστεί να διαχειριστείς κείμενο, έστω και αν πρόκειται για κείμενο με αριθμούς. Θα μάθεις να γράφεις πιο «επαγγελματικά» προγράμματα.

Έννοιες κλειδιά:

- *ορμαθός χαρακτήρων*
- *τύπος (κλάση) string*
- *σύνδεση ορμαθών*
- *αναζήτηση υποορμαθού*
- *πίνακες χαρακτήρων*
- *μετατροπή ορμαθού σε αριθμό*
- *μετατροπή αριθμού σε ορμαθό*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Αν γράψεις στο πρόγραμμά σου:

```
cout << "πάν μέτρον εκατό πόντοι" << endl;
```

θα περιμένεις ως αποτέλεσμα, σύμφωνα με όσα έχουμε μάθει μέχρι τώρα, να δεις στην οθόνη

```
πάν μέτρον εκατό πόντοι
```

Ξέρουμε λοιπόν πώς να στείλουμε ένα κείμενο στην οθόνη μας μέσω του ρεύματος *cout*. Δεν ξέρουμε όμως πώς να αποθηκεύσουμε ένα κείμενο στη μνήμη ούτε πώς να το επεξεργαστούμε.

Η πιο απλή ιδέα, σύμφωνα με αυτά που ξέρουμε ήδη, είναι να δούμε το κείμενο ως πίνακα χαρακτήρων:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ','π','ό','ν','τ','ο','ι' };
```

Σωστό! Και αν δώσεις:

```
cout << a << endl;
```

θα δεις:

```
πάν μέτρον εκατό πόντοι
```

(είναι όμως πιθανό να δεις το κείμενό σου να ακολουθείται από μερικά «σκουπίδια».)

Η C++ σου επιτρέπει να κάνεις τη δήλωση πολύ πιο απλά:

```
char b[] = "πάν μέτρον εκατό πόντοι";
```

και αν δώσεις:

```
cout <<  b << endl;
```

θα δεις σίγουρα:

```
πάν μέτρον εκατό πόντοι
```

Έχουν διαφορά οι δύο δηλώσεις; Ναι. Αν ζητήσεις:

```
cout << (sizeof a) << "  " << (sizeof b) << endl;
```

θα πάρεις:

```
            23  24
```

Η διαφορά οφείλεται σε ένα `char(0)` (ή αλλιώς: `'\0'`), που μπαίνει ως τελευταίος χαρακτήρας στον b. Αυτό είναι κληρονομιά από τη C, που χρησιμοποιεί αυτόν τον χαρακτήρα ως *φρουρό* στους αλγόριθμους επεξεργασίας κειμένων της βιβλιοθήκης της. Το ότι στην πρώτη περίπτωση δεν έχουμε αυτόν τον `char(0)` έχει ως αποτέλεσμα να παίρνουμε τα «σκουπίδια» σε ορισμένες περιπτώσεις.

Η βασική ιδέα λοιπόν είναι αυτή: *Το κείμενο παριστάνεται ως πίνακας με στοιχεία τύπου* `char`. Η C έμεινε σε αυτό και απλώς περιλαμβάνει στις βιβλιοθήκες της πολλές συναρτήσεις που βοηθούν στην επεξεργασία τέτοιων πινάκων. Αυτές οι συναρτήσεις έχουν κληρονομηθεί και στη C++. Αλλά, στη C++ σχεδιάστηκε ένας ειδικός τύπος (κλάση) με το όνομα `string` (ορμαθός) που έχει ενσωματωμένες πολλές μεθόδους που μας χρειάζονται για να διαχειριστούμε ένα κείμενο.

Για να δουλέψεις με τον τύπο `string` θα πρέπει να περιλάβεις στο πρόγραμμά σου το αρχείο string ("`#include <string>`").

10.1 Δήλωση Μεταβλητών Τύπου *string*

Η πιο απλή περίπτωση δήλωσης είναι η εξής[1]:

[1] Αν δεν βάλεις το "`using namespace std`" θα πρέπει να δηλώνεις "`std::string`".

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s0;
```

Η *s0* έχει ως τιμή έναν ορμαθό *μηδενικού μήκους*, χωρίς περιεχόμενο. Στη συνέχεια μπορείς να του δώσεις την τιμή που θέλεις, όπως θα δούμε παρακάτω.

Πάντως, μπορείς να του δώσεις τιμή από την αρχή, με τη δήλωση:

```
   string s1( "ένας ορμαθός της C++" );
```

Να σου υπενθυμίσουμε εδώ, ότι αν ο ορμαθός που θα εκχωρήσεις ως τιμή έχει κάποιους ειδικούς χαρακτήρες θα πρέπει να χρησιμοποιήσεις τον χαρακτήρα "\" (δες τον Πίν. 4-3). Π.χ.:

```
   string path( "f:\\cwork\\prog7_1.cpp" );
   string quest( "what\'s this\?" );
```

Αν μετά τα παραπάνω δώσεις τις εντολές:

```
   cout << "    s0: " << s0 << endl;
   cout << "    s1: " << s1 << endl;
   cout << "  path: " << path << endl;
   cout << " quest: " << quest << endl;
```

θα πάρεις:

```
    s0:
    s1: ένας ορμαθός της C++
  path: f:\cwork\prog7_1.cpp
 quest: what's this?
```

Δεν μπορείς να δώσεις ως αρχική τιμή μια σταθερά τύπου **char**. Έτσι, η δήλωση:

```
   string a( 'a' );
```

είναι λάθος!

Μπορείς όμως να δώσεις:

```
   string a( 10,'a' );
```

οπότε η:

```
   cout << "     a: " << a << endl;
```

θα δώσει:

```
     a: aaaaaaaaaa
```

Δηλαδή η a πήρε ως τιμή έναν ορμαθό με 10 **'a'**. Αν λοιπόν θέλεις να δώσεις ως τιμή ένα **'a'** μπορείς να δώσεις ένα από τα παρακάτω:

```
   string a( 1,'a' );
   string a( "a" );
```

Μπορείς να δώσεις ως αρχική τιμή έναν πίνακα τύπου **char**:

```
   char b[] = "πάν μέτρον εκατό πόντοι";
   string s2( b );
```

Μπορείς όμως να δώσεις ως αρχική τιμή και την τιμή μιας άλλης μεταβλητής τύπου **string**:

```
   string s2( a );
```

ή έναν *υποορμαθό* μιας μεταβλητής τύπου **string**:

```
   string s3( s1, 5, 7 );
```

Με αυτήν τη δήλωση δίνουμε ως αρχική τιμή στην *s3* έναν υποορμαθό της *s1* που ξεκινάει από τον χαρακτήρα 5 (αρχίζουμε το μέτρημα από 0) και έχει μήκος 7 χαρακτήρες. Αν *s1* είναι αυτή που δηλώσαμε παραπάνω, τότε, αν ζητήσουμε να τυπωθεί η τιμή της *s3*, θα πάρουμε:

```
ορμαθός
```

10.2 Μετατροπή σε Απλό Πίνακα

Πριν προχωρήσουμε, ας δούμε πώς μπορείς να πάρεις το «περιεχόμενο» ενός αντικειμένου τύπου *string*, σε έναν πίνακα με στοιχεία τύπου `char`. Υπάρχουν δύο *μέθοδοι* γι' αυτό: η *c_str* και η *data* που κάνουν την ίδια –περίπου– δουλειά.

Ας πούμε ότι έχουμε:

```
const int sz = 50;
string s1( "ένας ορμαθός της C++" );
char ac[sz];
```

και παίρνουμε το `s1.c_str()`. Αυτό έχει `char(0)` στο τέλος. Έτσι, μπορείς να το χειριστείς με τις συναρτήσεις της C, π.χ. να τα αντιγράψεις σε έναν πίνακα σαν τον ac:

```
strcpy( ac, s1.c_str() );
```

Για τη *strcpy()* θα τα πούμε στη συνέχεια.

Ποια η διαφορά του `s1.data()`; *Δεν υπάρχει εγγύηση ότι θα έχει* `char(0)` *στο τέλος*. Έτσι, η

```
strcpy( ac, s1.data() );
```

δεν είναι σίγουρο ότι θα δουλέψει. Πάντως σε πολλές περιπτώσεις θα δεις να συμπεριφέρεται ακριβώς σαν το `s1.c_str()`.

Και για τις δύο συναρτήσεις υπάρχει ένας περιορισμός: *Μην προσπαθήσεις να τροποποιήσεις τους πίνακες που βγάζουν*. Για παράδειγμα μην προσπαθήσεις να κάνεις κάτι σαν:

```
(c1.c_str())[3] = 'Σ';
```

10.3 Εκχώρηση Τιμής και Αντιμετάθεση

Χρησιμοποιώντας τον τελεστή "`=`" της εκχώρησης, μπορείς να εκχωρήσεις σε μια μεταβλητή τύπου *string*:

- μια τιμή τύπου *string*,
- μια τιμή τύπου `char`,
- έναν ορμαθό χαρακτήρων

Έτσι, αν έχεις δηλώσει:

```
string s0;
string s1( "ένας ορμαθός της C++" );
char b[] = "πάν μέτρον εκατό πόντοι";
```

μπορείς να δώσεις:

```
s0 = s1;                cout << "    s0: " << s0 << endl;
s0 = 'a';               cout << "    s0: " << s0 << endl;
s0 = b[4];              cout << "    s0: " << s0 << endl;
s0 = "what\'s this\?";  cout << "    s0: " << s0 << endl;
```

και θα πάρεις από τις εντολές εξόδου:

```
 s0: ένας ορμαθός της C++
 s0: a
 s0: μ
 s0: what's this?
```

Εκτός όμως από το "`=`", μπορείς να εκχωρήσεις τιμή και με κάποια από τις μορφές της μεθόδου *assign()*: Ας πούμε ότι έχουμε δηλώσει:

```
string s1( "ένας ορμαθός της C++" );
char b[] = "0123456789";
string s2;
```

1. Μπορείς να δώσεις: "`s2.assign(s1)`" που είναι ίδιο με το "`s2 = s1`". Οι

```
s2.assign( s1 );              cout << s2 << endl;
```

θα δώσουν:

```
ένας ορμαθός της C++
```

2. Μπορείς να δώσεις: "`s2.assign(s1, p, n)`" που σημαίνει: Δώσε ως τιμή στην *s2* το κομμάτι της *s1* που έχει *n* (πλήθος) χαρακτήρες και ξεκινάει από τον χαρακτήρα *p*. Π.χ. οι

```
   s2.assign( s1, 5, 7 );          cout << s2 << endl;
```

θα δώσουν:

```
ορμαθός
```

3. Μπορείς να δώσεις: "`s2.assign(cs)`" όπου *cs* ορμαθός ή πίνακας με στοιχεία τύπου `char`. Π.χ. οι:

```
   s2.assign( b );                 cout << s2 << endl;
   s2.assign( "0123456789" );      cout << s2 << endl;
```

θα δώσουν:

```
0123456789
0123456789
```

3. Μπορείς να δώσεις: "`s2.assign(cs, n)`" όπου *cs* ορμαθός ή πίνακας με στοιχεία τύπου `char` και *n* ακέραιος, από 0 μέχρι το μήκος του *cs*. Στην *s2* εκχωρούνται οι *n* πρώτοι χαρακτήρες του *cs*. Π.χ. οι:

```
   s2.assign( b, 5 );              cout << s2 << endl;
   s2.assign( "0123456789", 5 );   cout << s2 << endl;
```

θα δώσουν:

```
01234
01234
```

4. Τέλος, μπορείς να δώσεις "`s2.assign(n, c)`" όπου *n* φυσικός και *c* τιμή τύπου `char`. Τιμή της *s2* είναι ένας ορμαθός με *n* φορές τον *c*. Π.χ. η:

```
   s2.assign( 5, 'a' );            cout << s2 << endl;
```

θα δώσει:

```
aaaaa
```

Αντιμετάθεση τιμών ξέρεις να κάνεις! Αν πρόκειται για τιμές τύπου *string* τα πράγματα είναι πιο εύκολα: υπάρχει σχετική μέθοδος που λέγεται *swap()*. Δες τις παρακάτω εντολές:

```
   cout << " s1: " << s1 << "  s2: " << s2 << endl;
   s2.swap( s1 );
   cout << " s1: " << s1 << "  s2: " << s2 << endl;
```

που δίνουν:

```
 s1: ένας ορμαθός της C++  s2: aaaaa
 s1: aaaaa  s2: ένας ορμαθός της C++
```

Δηλαδή: η "`s2.swap(s1)`" έχει ως αποτέλεσμα την αντιμετάθεση τιμών των μεταβλητών *s1* και *s2*. Αλλά, όπως θα δούμε αργότερα, η αντιμετάθεση με τη *swap()* είναι ασφαλέστερη από την αντιμετάθεση που ξέρουμε.

10.4 Ανάγνωση και Γραφή

Ας ξεκινήσουμε με το γράψιμο, για το οποίο έχουμε ήδη μιλήσει. Απλώς θα συμπληρώσουμε ότι ο τελεστής "`<<`" στέλνει τιμές μεταβλητών τύπου `string`, όχι μόνον μέσω του *cout* στην οθόνη, αλλά και μέσω οποιουδήποτε ρεύματος *ofstream* σε αρχείο text.

Πώς μπορούμε να διαβάσουμε την τιμή μιας μεταβλητής *a* τύπου `string` από το πληκτρολόγιο ή από κάποιο αρχείο; Δηλαδή, δεν μπορούμε να διαβάσουμε την τιμή της *a* από το *cin* με τον "`>>`"; Μπορούμε, αλλά... Ας κάνουμε μια δοκιμή. Δίνουμε:

```
   cout << "Τι εχεις να πεις;" << endl;
   cin >> a;
   cout << a << endl;
```

και έχουμε την εξής εκτέλεση:

```
Τι εχεις να πεις;
των φρονίμων τα παιδιά τα κάνει κρεμαστάρια<enter>
των
```

Δηλαδή, μόλις βρήκε κενό στάματησε το διάβασμα· το ίδιο θα γινόταν αν έβρισκε τέλος γραμμής (`'\n'`) ή στηλοθέτη (`'\t'`).

Η ανάγνωση γίνεται με την[2]:

```
getline( cin, a, '\n' );
```

που λέει τα εξής:

- διάβασε από το ρεύμα *cin*,
- αποθηκεύοντας αυτά που διαβάζεις στην *a*,
- μέχρι να βρεις τέλος γραμμής (`'\n'`).

Ως πρώτο όρισμα μπορείς να βάλεις και ρεύμα τύπου *ifstream*, δηλαδή μπορείς να διαβάζεις και από αρχείο.

Αν, ως απάντηση στην `getline(cin, a, '\n')`, πιέσεις απλώς το πλήκτρο <enter>, χωρίς να γράψεις κείμενο, η *a* γίνεται *κενή*.

Η *getline()* είναι ένα πολύ καλό εργαλείο και για την ανάγνωση αρχείων text. Ας πούμε ότι στο αρχείο `inpData.txt` έχουμε γραμμές της μορφής:

```
873\t537\tΑνδρικόπουλος\t2510 997 799\n
```

(με τα '`\t`' και '`\n`' συμβολίζουμε τους χαρακτήρες "tab" και "newline" αντιστοίχως.) Αν έχουμε δηλώσει:

```
ifstream tin( "inpData.txt" );
string s1, s2, s3, s4;
```

μπορούμε να διαβάσουμε μια γραμμή ως εξής:

```
getline( tin, s1, '\t' );
getline( tin, s2, '\t' );
getline( tin, s3, '\t' );
getline( tin, s4, '\n' );
```

Βέβαια, οι δύο ακέραιοι, στην αρχή, έχουν διαβαστεί στις *s1* και *s2* ως *string*. Στη συνέχεια θα δεις πώς μπορούμε να τις μετατρέψουμε σε `int`. Παρομοίως, αν έχουμε μια γραμμή με μια ημερομηνία

```
19.11.2008
```

μπορούμε να τη διαβάσουμε ως εξής:

```
getline( tin, s1, '.' );
getline( tin, s2, '.' );
getline( tin, s3, '\n' );
```

Αντί για την (*std*`::`)*getline()* μπορείς να αποθηκεύσεις αυτό που διαβάζεις σε ορμαθό της C –δηλαδή πίνακα χαρακτήρων– με τη *getline* της *cin*. Αν, ας πούμε, έχεις δηλώσει:

```
char q[100];
```

μπορείς να διαβάσεις κείμενο από το πληκτρολόγιο στον q και από εκεί να το αντιγράψεις στην *a*, ως εξής:

```
cin.getline( q, 100 );
a.assign( q );
```

Εδώ ζητούμε να διαβαστεί μια γραμμή από το πληκτρολόγιο και να αποθηκευτεί στον *q*· αλλά να διαβαστούν το πολύ 99 (= 100 – 1) χαρακτήρες. Στο μήκος αυτό (100) περιλαμβάνεται και μια θέση για το `char(0)` που θα σημειώνει το τέλος του κειμένου στον *q*. Στη συνέχεια, με την `a.assign(q)`, αντιγράφουμε το περιεχόμενο του *q* (χωρίς το `char(0)`) στην *a*.

[2] `std::getline` για την ακρίβεια...

Η απάντηση <enter> χωρίς κείμενο στη `cin.getline()` έχει παρόμοιο αποτέλεσμα με αυτό που είδαμε στην `std::getline`: ο *q* παραμένει «κενός» (με `char(0)` στη θέση q[0]).

Ας δούμε και ένα

Παράδειγμα

Θα κάνουμε πιο ευέλικτο το πρόγραμμα αντιγραφής αρχείων που είδαμε στο Κεφ. 8. Θα βάλουμε σε μεταβλητές τα ονόματα των αρχείων:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
   ifstream s;
   ofstream t;
   string sFlNm, tFlNm;
   char ch;

   cout << "όνομα αρχείου: ";  getline( cin, sFlNm, '\n' );
   s.open( sFlNm.c_str() );
   if ( s.fail() )
      cerr << "δεν μπορώ να ανοίξω το αρχείο " << sFlNm << endl;
   else
   {
      cout << "όνομα αντιγράφου: ";  getline( cin, tFlNm, '\n' );
      t.open( tFlNm.c_str() );
      if ( t.fail() )
      {
         s.close();
         cerr << "δεν μπορώ να δημιουργήσω το " << tFlNm << endl;
      }
      else // ok και τα δύο αρχεία ανοικτά
      {
         s.get( ch );
         while ( !s.fail() && !t.fail() )
         {  t.put( ch );  s.get( ch );  } // while
         if ( !s.eof() )
            cerr << "πρόβλημα κατά την αντιγραφή" << endl;
         t.close();
         s.close();
      } // if ( t.fail
   } // if ( s.fail
} // main
```

Όπως βλέπεις, βάλαμε μεν τα ονόματα των αρχείων σε μεταβλητές τύπου `string`, αλλά δεν μπορούμε να περάσουμε τα ονόματα των μεταβλητών στις *open* που περιμένουν ορμαθό χαρακτήρων ή πίνακα με στοιχεία τύπου `char`. Έτσι, περάσαμε τα `sFlNm.c_str()` και `tFlNm.c_str()`.

10.5 Συγκρίσεις

Μπορείς να συγκρίνεις δύο ορμαθούς μεταξύ τους είτε με τη μέθοδο *compare()* είτε με τους γνωστούς τελεστές (`<`, `>`, `==` κλπ).

Η "`s1.compare(s2)`" δίνει ακέραιη

- *αρνητική τιμή* αν η τιμή της *s1* προηγείται λεξικογραφικώς της τιμής της *s2*,
- *θετική* τιμή αν *η τιμή της s1 έπεται λεξικογραφικώς της τιμής της s2*,
- *μηδέν* (0) αν *οι τιμές των s1 και s2 είναι ίσες*.

Αν λοιπόν δηλώσουμε:

```
string s1( "abc" );
string s2( "bce" );
string s3( "abcd" );
string s4( "abc" );
```

οι:

```
cout << s1.compare(s2) << "  " << s2.compare(s1) << endl;
cout << s1.compare(s3) << "  " << s3.compare(s1) << endl;
cout << s1.compare(s4) << "  " << s4.compare(s1) << endl;
```

θα δώσουν:

```
-1  1
-1  1
0  0
```

που σημαίνουν:

- Η τιμή της *s1* ("**abc**") λεξικογραφικώς προηγείται της τιμής της *s2* ("**bce**"), ενώ
- η τιμή της *s2* λεξικογραφικώς έπεται της τιμής της *s1*. Παρομοίως,
- η τιμή της *s1* λεξικογραφικώς προηγείται της τιμής της *s3* ("**abcd**"), ενώ
- η τιμή της *s3* λεξικογραφικώς έπεται της τιμής της *s1*. Τέλος,
- οι τιμές των *s1* και *s4* είναι ίσες.

Μπορείς να συγκρίνεις και την τιμή μιας μεταβλητής τύπου **string** με έναν ορμαθό χαρακτήρων ή έναν πίνακα με στοιχεία τύπου **char**. Αν, για παράδειγμα

```
char a[] = "abd";
```

οι:

```
cout << s1.compare( "cdef" ) << endl;
cout << s2.compare( a ) << endl;
```

θα δώσουν:

```
-2
1
```

Μπορείς λοιπόν να γράφεις στο πρόγραμμά σου:

```
if ( s1.compare(s2) < 0 )
   cout << "το " << s1 << " προηγείται λεξικογραφικώς του "
        << s2 << endl;
else if ( s1.compare(s2) > 0 )
   cout << "το " << s1 << " έπεται λεξικογραφικώς του "
        << s2 << endl;
else
   cout << "τα " << s1 << " και " << s2
        << " είναι ίσα" << endl;
```

Πάντως μπορείς να χρησιμοποιήσεις και τους γνωστούς μας τελεστές σύγκρισης:

αντί για **s1.compare(s2)** θ **0** μπορείς να γράφεις **s1** θ **s2**

όπου θ κάποιος από τους "**>**", "**>=**", "**<**", "**<=**", "**==**", "**!=**".

Έτσι, μπορείς να γράψεις την παραπάνω διπλή **if** και ως εξής:

```
if ( s1 < s2 )
   cout << "το " << s1 << " προηγείται λεξικογραφικώς του "
        << s2 << endl;
else if ( s1 > s2 )
   cout << "το " << s1 << " έπεται λεξικογραφικώς του "
        << s2 << endl;
else
   cout << "τα " << s1 << " και " << s2
        << " είναι ίσα" << endl;
```

όπως και:

```
if ( s1 == "abc" )  cout << "eq" << endl;
```

Ας δούμε ένα παράδειγμα χρήσης της δυνατότητας σύγκρισης.

Παράδειγμα

Έχουμε έναν πίνακα με στοιχεία τύπου `string`:

```
   string v[N] = { "PASCAL", "BASIC", "FORTRAN", "COBOL", "RPG",
                   "ALGOL", "LISP", "PROLOG", "LOGO", "C++",
                   "PL/I", "ADA", "BCPL", "SNOBOL", "APL", "C" };
```

και θέλουμε να ταξινομήσουμε τα στοιχεία κατ' αλφαβητική σειρά.

Μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο που αντιγράφουμε από την §9.5.2:

```
// ταξινόμηση
   for ( k = N; k >= 2; k = k-1 )
   {
      meg = maxNdx( v, 1, k );
      sv = v[meg];  v[meg] = v[k];  v[k] = sv;
   } // for
```

μόνο που εδώ τα στοιχεία μας είναι στις θέσεις[3] v[0] μέχρι v[*N*-1]. Ακόμη η αντιμετάθεση τιμών των v[*mxp*] και v[*k*] μπορεί να γίνει με τη *swap*:

```
// ταξινόμηση
   for ( k = N-1; k >= 1; k = k-1 )
   {
      mxp = maxNdx( v, N, 0, k );
      v[mxp].swap( v[k] );
   } // for
```

Και με τη *maxNdx* τι γίνεται; Την αντιγράφουμε όπως είναι αλλάζοντας μόνον τον τύπο των στοιχείων του πίνακα από `int` σε *string*:

```
   int maxNdx( const string x[], int n, int from, int upto )
```

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <string>
using namespace std;

   int maxNdx( const string x[], int n, int from, int upto );

int main()
{
   const int N = 16;

   string v[N] = { "PASCAL", "BASIC", "FORTRAN", "COBOL", "RPG",
                   "ALGOL", "LISP", "PROLOG", "LOGO", "C++",
                   "PL/I", "ADA", "BCPL", "SNOBOL", "APL", "C" };
   int  k, mxp;
// ταξινόμηση
   for ( k = N-1; k >= 1; k = k-1 )
   {
      mxp = maxNdx( v, N, 0, k );
      v[mxp].swap( v[k] );
   } // for
// αποτέλεσμα
   for ( k = 0; k <= N-1; k=k+1 )  cout << v[k] << endl;
} // main
```

Αποτέλεσμα:

```
ADA
ALGOL
APL
```

---

[3] Φυσικά δεν υπάρχει πρόβλημα αν θέλεις να βάλεις φρουρούς στην αρχή και στο τέλος. Άλλαξε τη δήλωση σε:

```
   string v[N+2] = { " ", "PASCAL",... "C", " " };
```

και μετά δώσε:

```
   v[0].assign(5, char(0));  v[N+1].assign(5,char(255));
```

```
BASIC
BCPL
C
C++
COBOL
FORTRAN
LISP
LOGO
PASCAL
PL/I
PROLOG
RPG
SNOBOL
```

10.6 Μήκος Ορμαθού – Κενός Ορμαθός

Αφού τιμή μιας μεταβλητής τύπου **string** είναι ένας ορμαθός χαρακτήρων, έχει νόημα το μήκος της, που είναι ακριβώς το μήκος της τιμής της. Το μήκος μας δίνεται από τη μέθοδο **length** (μήκος).

Ας πούμε ότι έχουμε δηλώσει:

```
string s0, s1( "ένας ορμαθός της C++" );
string path( "f:\\cwork\\prog7_1.cpp" ),
       quest( "what\'s this\?" );
string a( 10,'a' ), c( "a" );
```

και δίνουμε:

```
cout << s0.length() << "  " << s1.length() << "  "
     << path.length() << "  " << quest.length() << "  "
     << a.length() << "  " << c.length() << endl;
```

Αποτέλεσμα:

```
0  20  20  12  10  1
```

Πρόσεξε ότι στην *path* το "\\" μετράει ως *ένας* χαρακτήρας. Ως ένας χαρακτήρας μετράνε και τα "\'" και "\?" στην *quest*.

Εκτός από τη *length()* υπάρχει και η *size()* που κάνει ακριβώς τα ίδια. Θα πάρεις τα παραπάνω αποτελέσματα και με τη:

```
cout << s0.size() << "  " << s1.size() << "  "
     << path.size() << "  " << quest.size() << "  "
     << a.size() << "  " << c.size() << endl;
```

Πρόσεξε ακόμη ότι το μήκος της *s0* είναι μηδέν. Η τιμή της *s0* είναι ο κενός ορμαθός, χωρίς κάποιον χαρακτήρα. Αυτό ελέγχεται με τη μέθοδο *empty()*, που μας δίνει τιμή **true** αν η τιμή της μεταβλητής είναι ο κενός ορμαθός (μηδενικό μήκος) και **false** αν έχει έστω και έναν χαρακτήρα. Π.χ. οι:

```
if ( s0.empty() )  cout << " το s0 είναι άδειο!" << endl;
             else  cout << " το s0 κάτι έχει μέσα" << endl;
if ( s1.empty() )  cout << " το s1 είναι άδειο!" << endl;
             else  cout << " το s1 κάτι έχει μέσα" << endl;
```

θα δώσουν:

```
το s0 είναι άδειο!
το s1 κάτι έχει μέσα
```

10.6.1 Το Μέγιστο Μήκος Ορμαθού

Και ποιο είναι το μέγιστο μήκος ορμαθού που μπορώ να αποθηκεύσω σε μεταβλητή *string*; Το βρίσκεις με τη *max_size()*:

```
string s1;
```

```
   cout << "max_size = " << s1.max_size() << endl;
```

Αποτέλεσμα (BC++ v.5.5):

```
max_size = 4294967281
```

10.6.2 Ο Τύπος "size_type"

Ο τύπος του αποτελέσματος της *length()*, της *size()* και της *max_size()* είναι *string*::*size_type*, που είναι ο τύπος της C++ για τα μεγέθη των αντικειμένων που παριστάνονται στον τύπο *string*.

Για παρόμοιες δουλειές η C μας κληρονομεί τον **size_t** που είναι ο τύπος του αποτελέσματος που επιστρέφει ο τελεστής "**sizeof**". Αν ψάξεις το **cstddef** (ή το **stddef.h**) θα βρεις τον ορισμό:

```
typedef unsigned int size_t;
```

ή κάτι παρόμοιο.

Μπορείς να θεωρείς ότι οι δυο τύποι είναι ταυτόσημοι.[4]

10.7 Σύνδεση – Επισύναψη

Πολύ συχνά έχουμε ανάγκη να **συνδέσουμε** (concatenate) δυο ή περισσότερους ορμαθούς. Αυτό μπορούμε να το κάνουμε «προσθέτοντας» ορμαθούς με το "**+**". Αν, για παράδειγμα, έχουμε δηλώσει:

```
   string s1( "πέντε" ), s2( "στο χέρι παρά δέκα" ), s3;
```

τότε οι εντολές:

```
   cout << ("κάλλιο " + s1 + " και " + s2 + " και καρτέρει")
        << endl;
   s3 = "κάλλιο " + s1 + " και " + s2 + " και καρτέρει";
   cout << s3 << endl;
```

θα μας δώσουν:

```
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

Με τον τελεστή "**+**" μπορούμε να συνδέσουμε:

- δύο μεταβλητές τύπου **string**,
- μεταβλητή τύπου **string** και ορμαθό χαρακτήρων,
- μεταβλητή τύπου **string** και πίνακα με στοιχεία τύπου **char** (το περιεχόμενό του θα πρέπει να τελειώνει με '**\0**'.)

Το αποτέλεσμα της πράξης μπορεί να εκχωρηθεί σε μια μεταβλητή τύπου **string**.

Πολύ συχνά υπάρχει ανάγκη να **επισυνάψουμε** (append) έναν ορμαθό στην τιμή μιας μεταβλητής τύπου **string**. Π.χ. αυτά που κάναμε στο παράδειγμα θα μπορούσαν να γίνουν και ως εξής:

```
   s3 = "κάλλιο ";
   s3 = s3 + s1;  s3 = s3 + " και ";  s3 = s3 + s2;
   s3 = s3 + " και καρτέρει";
```

Για την επισύναψη ο **string** έχει ειδική μέθοδο, την *append()*. Τα ίδια πράγματα θα μπορούσαν να γραφούν και ως εξής[5]:

---

[4] Πάντως, για να μην χρησιμοποιεί η C++ τον "**size_t**" διατηρεί το δικαίωμα για αλλαγές.

[5] Αλλά και ως εξής:

```
   s3 = "κάλλιο ";   s3 += s1;  s3 += " και ";
                     s3 += s2;  s3 += " και καρτέρει";
```

```
s3 = "κάλλιο ";
s3.append( s1 );  s3.append( " και " );  s3.append( s2 );
s3.append( " και καρτέρει" );
```

10.8 Αναζήτηση

Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο. Αυτό το ξέρεις καλά, αφού σίγουρα έχεις χρησιμοποιήσει πολλές φορές την εντολή **find** (ή **αναζήτησε**) του κειμενογράφου σου. Η κλάση `string` είναι εφοδιασμένη με αρκετές μεθόδους για τη δουλειά αυτή.

Ας ξεκινήσουμε με τη `find` (βρες). Αν έχεις δηλώσει "`string s`" και αν *t* είναι

- μια μεταβλητή τύπου `string` ή
- ένας ορμαθός χαρακτήρων ή
- πίνακας χαρακτήρων ή
- τιμή τύπου `char`

τότε

- Η `s.find(`*t*`)` ψάχνει την τιμή της *s*, *από αριστερά προς τα δεξιά*, για να εντοπίσει τον «υποορμαθό» *t*. Αν τον βρει επιστρέφει τον δείκτη του χαρακτήρα στην *s*, που αρχίζει ο *t*.
- Η `s.find(`*t*, *i*`)`, όπου *i* τιμή τύπου *string*`::`*size_type*, ψάχνει την τιμή της *s*, *από αριστερά προς τα δεξιά*, ξεκινώντας από τιμή δείκτη *i*, για να εντοπίσει τον «υποορμαθό» *t*. Αν τον βρει επιστρέφει τον δείκτη, στην *s*, που αρχίζει ο *t*.

Αν δεν βρεθεί η *t* μέσα στην *s* η *find* επιστρέφει μια απίθανη τιμή. Αυτήν την τιμή μπορείς να την ελέγχεις ως *string*`::`*npos*. Ο τύπος του αποτελέσματος είναι *string*`::`*size_type*.

Παράδειγμα

Γράφει ο Ποιητής:

ΑΞΙΟΝ ΕΣΤΙ στὸ πέτρινο πεζούλι
ἀντικρὺ τοῦ πελάγους ἡ Μυρτὼ νὰ στέκει
σὰν ὡραῖο ὀκτὼ ἢ σὰν κανάτι
μὲ τὴν ψάθα τοῦ ἥλιου στὸ ἕνα χέρι

Μπορούμε να αποθηκεύσουμε αυτό το κείμενο σε μια μεταβλητή

```
string text1;
```

ως εξής: [6]

```
text1 = "ΑΞΙΟΝ ΕΣΤΙ στο πέτρινο πεζούλι\n"
        "αντικρύ του πελάγους η Μυρτώ να στέκει\n"
        "σαν ωραίο οκτώ ή σαν κανάτι\n"
        "με την ψάθα του ήλιου στο ένα χέρι";
```

Με τα `'\n'` αποθηκεύουμε και τις αλλαγές γραμμής.

Δηλώνουμε ακόμη:

```
char a[] = "Μυρτώ";
```

Οι εντολές:

```
if ( text1.find("Μυρτώ") == string::npos )
```

Λέγαμε στο Κεφ. 2 ότι, για το αριθμητικό νόημα του "`+`", "`v += a`" σημαίνει: "`v = v + a`". Το ίδιο ισχύει και όταν ο τελεστής "`+`" σημειώνει τη σύνδεση.

[6] Μπορείς να το γράψεις και έτσι:

```
text1 = "ΑΞΙΟΝ ΕΣΤΙ στο πέτρινο πεζούλι\n";
text1.append( "αντικρύ του πελάγους η Μυρτώ να στέκει\n" );
text1.append( "σαν ωραίο οκτώ ή σαν κανάτι\n" );
text1.append( "με την ψάθα του ήλιου στο ένα χέρι" );
```

```
      cout << "Μυρτώ δεν βρέθηκε" << endl;
   else
      cout << text1.find("Μυρτώ") << endl;
   if ( text1.find("Μυρσίνη") == string::npos )
      cout << "Μυρσίνη δεν βρέθηκε" << endl;
   else
      cout << text1.find( "Μυρσίνη" ) << endl;
   cout << text1.find(a) << endl;
   cout << text1.find('M') << endl;
   cout << text1.find(a[0]) << endl;
```

θα δώσουν:

```
54
Μυρσίνη δεν βρέθηκε
54
54
54
```

Δηλαδή:

- Η λέξη `"Μυρτώ"` υπάρχει στη θέση με δείκτη 54 (γραμμές 1 και 3).
- Το γράμμα `'M'` υπάρχει στη 54 (γραμμές 4 και 5).
- Η λέξη `"Μυρσίνη"` *δεν* υπάρχει στην *text1* (γραμμή 2).

Παρατήρηση:▶

Για όσους δεν το πρόσεξαν (και δεν θύμωσαν) ήδη: Ο σωστός τρόπος να γράψουμε το πρόγραμμα είναι:

```
   size_t pos;
// . . .
   pos = text1.find("Μυρτώ");
   if ( pos == string::npos ) cout << "Μυρτώ δεν βρέθηκε" << endl;
                         else cout << pos << endl;
   pos = text1.find("Μυρσίνη");
   if ( pos == string::npos ) cout << "Μυρσίνη δεν βρέθηκε" << endl;
                         else cout << pos << endl;
   cout << text1.find(a) << endl;
   cout << text1.find('M') << endl;
   cout << text1.find(a[0]) << endl;
```

Η αναζήτηση κειμένου είναι χρονοβόρα διαδικασία και δεν αφήνουμε τέτοια πράγματα στην «καλή θέληση» (και ευφυΐα) του μεταγλωττιστή. Στο παράδειγμά μας το κείμενο δεν έχει ούτε 100 χαρακτήρες· άρα μικρό το κακό. Σε «πραγματικές συνθήκες» όμως…◀

Οι εντολές:

```
   cout << text1.find( "του" ) << endl;
   cout << text1.find( "του", 40 ) << endl;
```

θα δώσουν:

```
39
110
```

Δηλαδή: η λέξη "`του`" υπάρχει για πρώτη φορά στη θέση με δείκτη 39. *Ξεκινώντας την αναζήτηση από τη θέση* με δείκτη 40, η λέξη "`του`" *βρίσκεται για πρώτη φορά* με δείκτη 110.

Ας πάρουμε το κείμενο:

Νῦν ἡ ταπείνωση τῶν Θεῶν Νῦν ἡ σποδὸς τοῦ Ἀνθρώπου
Νῦν Νῦν τὸ μηδὲν

καὶ Αἰὲν ὁ κόσμος ὁ μικρὸς, ὁ Μέγας!

που το αποθηκεύουμε στη μεταβλητή:

```
   string text2;
// . . .
   text2 = "Νυν η ταπείνωση των θεών Νυν η σποδός του Ανθρώπου\n"
           "Νυν Νυν το μηδέν\n\n"
           "και Αιέν ο κόσμος ο μικρός, ο Μέγας!";
```

Αν έχουμε δηλώσει:

```
int ndx;
```

οι παρακάτω εντολές μας δίνουν όλους τους δείκτες όλων των θέσεων που αρχίζει η λέξη "Νυν":

```
ndx = text2.find( "Νυν" );
while ( 0 <= ndx && ndx < text2.length() )
{
   cout << ndx << "  ";
   ndx = text2.find( "Νυν", ndx+1 );
}
cout << endl;
```

Αποτέλεσμα:

```
0  25  51  55
```

Μια άλλη μέθοδος, η *rfind()*, είναι όμοια με τη *find()*, με μόνη διαφορά ότι ψάχνει *από δεξιά προς τα αριστερά* (από το τέλος προς την αρχή). Έτσι, για τα στοιχεία του παραπάνω παραδείγματος, οι εντολές:

```
cout << text1.rfind("του") << endl;
cout << text1.rfind("του", 108) << endl;
```

θα δώσουν:

```
110
39
```

Δες τώρα ένα άλλο πρόβλημα αναζήτησης: Ας πούμε ότι έχουμε την *text2* με την τιμή που έχει στο παράδειγμα και θέλουμε να βρούμε το πρώτο τονούμενο πεζό φωνήεν στο κείμενο. Μπορούμε να δώσουμε το εξής:

```
cout << text2.find_first_of("άέήίΐόύΰώ") << endl;
```

που θα μας δώσει:

```
10
```

Πράγματι, στη θέση με δείκτη 10 του κειμένου μας υπάρχει το 'ΐ' (ταπείνωση). Μπορούμε να συνεχίσουμε την αναζήτηση:

```
cout << text2.find_first_of("άέήίΐόύΰώ", 11) << endl;
```

Αποτέλεσμα:

```
22
```

Στη θέση με δείκτη 22 του κειμένου μας υπάρχει το 'ώ' (Θεών).

Δηλαδή, η μέθοδος *find_first_of()* μας δίνει *τον δείκτη της πρώτης θέσης όπου υπάρχει χαρακτήρας από κάποιο σύνολο*, που το δίνουμε *ως πρώτο όρισμα σε μορφήν ορμαθού*. Αν θέλουμε η αναζήτηση να μην ξεκινήσει από την αρχή, δίνουμε και δεύτερο όρισμα με τον δείκτη της επιθυμητή θέσης εκκίνησης.

Ό,τι είναι η *rfind()* για τη *find()* είναι η *find_last_of()* για τη *find_first_of()*: ψάχνει για τον τελευταίο χαρακτήρα που να ανήκει σε κάποιο σύνολο ή, αλλιώς, ψάχνει από δεξιά προς τα αριστερά.

Παρόμοια δουλειά κάνει και η *find_first_not_of()* η οποία ψάχνει για τον πρώτο χαρακτήρα που *δεν ανήκει* στο σύνολο-όρισμα. Η αντίστοιχη μέθοδος που ψάχνει από το τέλος είναι η *find_last_not_of()*.

Παρατήρηση: ▶

Θα πεις: «Καλά αυτά, αλλά ο Ποιητής γράφει πολυτονικό και εδώ κατακρεουργήθηκε!» και θα έχεις δίκιο! Αυτό όμως έγινε για να δείξουμε αυτά που θέλουμε με απλό και γρήγορο τρόπο. Υπάρχει ένας άλλος τύπος, ο *wstring*, ακριβώς σαν τον *string* αλλά με βάση τον `wchar_t`. Με αυτόν τον τύπο μπορείς να χειριστείς όλα τα σύμβολα του Unicode, επομένως και το πολυτονικό μας.◀

10.9 Αντικατάσταση – Διαγραφή – Εισαγωγή

Μια άλλη δουλειά, πολύ χρήσιμη, που την ξέρεις και αυτήν από τον κειμενογράφο σου (**replace** ή **αντικατάστησε**), είναι η αντικατάσταση ενός ορμαθού από έναν άλλον. Για μεταβλητές τύπου `string` μπορούσε να την επιτύχουμε με τη μέθοδο *replace()*. Δες ένα παράδειγμα.

Ας πούμε ότι έχουμε:

```
string target( "ενός" ), text1;
text1 ="Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου\n"
       "είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή\n")
       "ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.";
```

και δίνουμε τις εντολές:

```
cout << text1 << endl << endl;
text1.replace( text1.find(target), target.length(), "ΚΑΠΟΙΟΥ");
cout << text1 << endl;
```

Αποτέλεσμα:

```
Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ενός χαρακτήρα ή μιας λέξης ή
ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.

Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ΚΑΠΟΙΟΥ χαρακτήρα ή μιας λέξης ή
ενός κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.
```

Οι παράμετροι της `replace` στο παράδειγμά μας είναι οι εξής:

- Η πρώτη είναι ένας φυσικός που μας δίνει τον δείκτη της θέσης από την οποίαν ξεκινάει το προς αντικατάσταση κείμενο. Στην περίπτωσή μας βάλαμε τη θέση της τιμής του *text*1 που αρχίζει η τιμή της μεταβλητής *target* (δηλαδή τη λέξη "`ενός`") για πρώτη φορά (`text1.find(target)`).
- Η δεύτερη είναι και πάλι ένας φυσικός που δίνει το *μήκος* του κειμένου που θα αντικατασταθεί. Στην περίπτωσή μας βάλαμε το μήκος της τιμής της μεταβλητής *target* (δηλαδή της λέξης "`ενός`" που είναι 4).
- Η τρίτη παράμετρος είναι ο ορμαθός με τον οποίον θα γίνει η αντικατάσταση. Μπορεί να είναι ακόμη: τιμή τύπου `char` ή πίνακας χαρακτήρων ή μεταβλητή τύπου *string*.

Όπως βλέπεις τα μήκη των δύο ορμαθών ("`ενός`" και "`ΚΑΠΟΙΟΥ`") δεν είναι απαραίτητο να είναι ίσα. Αυτό φυσικά έχει ως συνέπεια να αλλάξει και το μήκος του κειμένου μας.

Αν θέλουμε να αντικαταστήσουμε όλες τις λέξεις "`ενός`" που θα βρούμε, δουλεύουμε όπως στην πολλαπλή *find*:[7]

```
ndx = text1.find( target );
while ( 0 <= ndx && ndx < text1.length() )
{
   text1.replace( ndx, target.length(), "ΚΑΠΟΙΟΥ" );
   ndx = text1.find( target, ndx+1 );
}
```

και παίρνουμε:

```
Ένα πολύ συνηθισμένο πρόβλημα επεξεργασίας κειμένου
είναι η αναζήτηση ΚΑΠΟΙΟΥ χαρακτήρα ή μιας λέξης ή
ΚΑΠΟΙΟΥ κομματιού κειμένου μέσα σε κάποιο άλλο κείμενο.
```

Με τη *replace()* μπορείς να κάνεις και *διαγραφή*. Αν δώσεις, π.χ.:

```
text1.replace( text1.find(target), target.length(), "" );
```

ζητάς να αντικατασταθεί η τιμή της *target*, την πρώτη φορά που θα βρεθεί στην τιμή της *text*1, με τον κενό ορμαθό, δηλ. να διαγραφεί.

---

[7] Το "`ndx+1`" θα δουλέψει συνήθως. Γενικώς, είναι λάθος. Το σωστό είναι "`ndx+μήκος("ΚΑΠΟΙΟΥ")`". Σκέψου το λιγάκι!

Υπάρχει όμως και μέθοδος *erase()* που κάνει διαγραφές. Στη συνήθη της μορφή παίρνει δύο ορίσματα: το πρώτο είναι ο δείκτης της θέσης από την οποίαν αρχίζει η διαγραφή και το δεύτερο είναι το πλήθος των χαρακτήρων που θα διαγραφούν. Μπορούμε λοιπόν να γράψουμε:

```
text1.erase( text1.find(target), target.length() );
```

Φυσικά, εκτός από αντικατάσταση και διαγραφή μπορείς να κάνεις και εισαγωγή με τη μέθοδο *insert()*. Είναι σαν την *append()* με τη διαφορά ότι:

- με την *append()* η επισύναψη γίνεται πάντοτε στο τέλος, ενώ
- με την *insert()* η εισαγωγή γίνεται όπου θέλουμε· το καθορίζουμε με την τιμή που δίνουμε στην πρώτη παράμετρο.

Ας ξαναδούμε το παράδειγμα που δώσαμε για την *append()*, αλλά κάπως πιο «ανακατεμένο». Είχαμε:

```
string s1( "πέντε" ), s2( "στο χέρι παρά δέκα" ), s3;
```

και δίνουμε:

```
s3 = " και καρτέρει";      cout << s3 << endl;
s3.insert( 0, s1 );        cout << s3 << endl;
s3.insert( 5, s2 );        cout << s3 << endl;
s3.insert( 0, "κάλλιο " ); cout << s3 << endl;
s3.insert( 12, " και " );  cout << s3 << endl;
```

Αποτέλεσμα:

```
 και καρτέρει
πέντε και καρτέρει
πέντεστο χέρι παρά δέκα και καρτέρει
κάλλιο πέντεστο χέρι παρά δέκα και καρτέρει
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

10.10 Διαχείριση Χαρακτήρων

Όπως στους πίνακες, έτσι και στις μεταβλητές τύπου *string*, μπορείς να διαχειρίζεσαι κάθε χαρακτήρα βάζοντας μετά από το όνομα της μεταβλητής τον κατάλληλο δείκτη. Αν έχεις δηλώσει:

```
string s3;
```

μπορείς να γράψεις για παράδειγμα:

```
if ( s3[0] == 'κ' )  s3[0] = 'K';
```

Βέβαια, χρειάζεται προσοχή: είναι δική σου ευθύνη να διασφαλίσεις ότι ο δείκτης θα παίρνει τιμές από 0 μέχρι `s3.length()-1`.

Αυτά τα πράγματα σου θυμίζουν πολύ τους πίνακες· αλλά η ομοιότητα σταματάει εδώ: Μια τιμή τύπου *string* κρύβει σίγουρα κάποιον πίνακα με τιμές τύπου `char` αλλά δεν είναι μόνον αυτό.

Εκτός από τη χρήση δείκτη, υπάρχει και άλλος τρόπος διαχείρισης των χαρακτήρων μιας μεταβλητής τύπου *string*: η μέθοδος `at`. Αυτό που γράψαμε παραπάνω μπορεί να γραφεί και ως εξής:

```
if ( s3.at(0) == 'κ' )  s3.at( 0 ) = 'K';
```

Αυτός ο τρόπος είναι ασφαλέστερος από αυτόν με τον δείκτη, διότι αν κάνεις λάθος και βγεις έξω από τα όρια της μεταβλητής σου θα ειδοποιηθείς, όπως θα μάθουμε αργότερα.

Παράδειγμα

Το παρακάτω πρόγραμμα, αντιστρέφει ένα κείμενο, που δίνεται ως στοιχείο εισόδου. Αν η αντίστροφη γραφή συμπίπτει με την ορθή, το πρόγραμμα εκφράζει τον ενθουσιασμό του! Στο πρόγραμμα χρησιμοποιούνται οι μέθοδοι:

- *at()*, για τη διαχείριση τιμής μιας μεταβλητής *string*, χαρακτήρα προς χαρακτήρα (`strIn.at(k)`)· για την ίδια δουλειά χρησιμοποιείται και η
- *length()*, διότι θέλουμε να επεξεργαστούμε όλους τους χαρακτήρες, από το τέλος προς την αρχή (`for (k = strIn.length()-1; k>=0; k=k-1)`),
- *compare()*, για τη σύγκριση τιμών (`strIn.compare(strOut) == 0`) δύο μεταβλητών τύπου `string`· σύγκριση όμως γίνεται και με τον τελεστή `!=` (`strIn != "T" && strIn != "t" && ...`),
- *append()*, για να κτίσουμε την τιμή μιας μεταβλητής τύπου *string*, χαρακτήρα προς χαρακτήρα. Για αυτήν συζητούμε και παρακάτω.

Πώς δουλεύει το πρόγραμμα; Αφού πάρουμε στη *strIn* το κείμενο που θα χειριστούμε αντιγράφουμε στη *strOut* έναν προς ένα τους χαρακτήρες της *strIn* αλλά από το τέλος προς την αρχή. Αυτό γίνεται με τη

```
      for ( k = strIn.length()-1; k >= 0; k = k-1 )
```

Έτσι αποκλείεται να βγούμε έξω από όρια και μπορούμε να χρησιμοποιήσουμε είτε τον δείκτη (`strIn[k]`) είτε την *at* (`strIn.at(k)`). Ποιο είναι το πρόβλημα με την *append*; Και οι δύο τρόποι μας δίνουν έναν χαρακτήρα και η *append* δεν δέχεται τέτοια παράμετρο. Και ποια λύση δίνουμε; Δηλώνουμε μια μεταβλητή

```
   string str1( "x" );
```

Το 'x' »κρατάει τη θέση» (place holder) για να έχουμε ορμαθό με μήκος 1. Στη συνέχεια το αντικαθιστούμε με κάθε χαρακτήρα που παίρνουμε από τη *strIn*:

```
         str1[0] = strIn.at(k);
```

Φυσικά αυτή δεν είναι η μοναδική λύση. Στην επόμενη παράγραφο θα δούμε μια καλύτερη.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   const string msg( "ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): " );

   int k;
   string strIn, strOut, str1( "x" );

   cout << msg;  getline( cin, strIn, '\n' );
   while ( strIn != "T" && strIn != "t" &&
           strIn != "Τ" && strIn != "τ" )
   {
      strOut = "";    // κενό
      for ( k = strIn.length()-1; k >= 0; k=k-1 )
      {
         str1[0] = strIn.at( k );
         strOut.append( str1 );
      }
      cout << " Η ανάποδη γραφή της: " << strIn
           << " είναι: " << endl;
      cout << strOut << endl;
      if ( strIn == strOut )
         cout << " Καλό αυτό!" << endl;
      cout << endl;
      cout << msg;  getline( cin, strIn, '\n' );
   } // while
   cout << " ΤΕΛΟΣ" << endl;
} // main
```

Να ένα παράδειγμα διαλόγου με το πρόγραμμα αυτό. Με κεκλιμένα οι απαντήσεις του χρήστη:

```
ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ABCDEFGHJIK
 Η ανάποδη γραφή της: ABCDEFGHJIK είναι:
```

```
KIJHGFEDCBA

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ΝΙΨΟΝ ΑΝΟΜΗΜΑΤΑ ΜΗ ΜΟΝΑΝ ΟΨΙΝ
 Η ανάποδη γραφή της: ΝΙΨΟΝ ΑΝΟΜΗΜΑΤΑ ΝΗ ΜΟΝΑΝ ΟΨΙΝ είναι:
ΝΙΨΟ ΝΑΝΟΜ ΗΝ ΑΤΑΜΗΜΟΝΑ ΝΟΨΙΝ

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ
 Η ανάποδη γραφή της: ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ είναι:
ΝΙΨΟΝΑΝΟΜΗΜΑΤΑΜΗΜΟΝΑΝΟΨΙΝ
 Καλό αυτό!

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): MADAM IN EDEN I'M ADAM
 Η ανάποδη γραφή της: MADAM IN EDEN I'M ADAM είναι:
MADA M'I NEDE NI MADAM

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): MADAMINEDENIMADAM
 Η ανάποδη γραφή της: MADAMINEDENIMADAM είναι:
MADAMINEDENIMADAM
 Καλό αυτό!

ΔΩΣΕ ΜΙΑ ΦΡΑΣΗ (Τ ΓΙΑ ΤΕΛΟΣ): Τ
 ΤΕΛΟΣ
```

10.11 Υποορμαθοί

Μερικές φορές θέλουμε να πάρουμε έναν **υποορμαθό** (substring), ένα κομμάτι ενός ορμαθού, για να το χειριστούμε χωριστά. Η μέθοδος *substr()* μας δίνει αυτήν τη δυνατότητα: μας δίνει μια νέα τιμή τύπου *string* με περιεχόμενο έναν υποορμαθό κάποιας άλλης τιμής του ίδιου τύπου.

Αν έχουμε μια μεταβλητή *s1* τύπου *string*, μπορούμε να πούμε ότι ένας υποορμαθός της ορίζεται από δύο φυσικούς αριθμούς:

- τον *δείκτη της θέσης* που αρχίζει και
- το *μήκος* που έχει.

Αυτές ακριβώς τις παραμέτρους περιμένει και η *substr()*. Αν:

```
string s1 = "ένας ορμαθός της C++", s2;

s2 = s1.substr( 5, 7 );
cout << s2 << endl;
```

θα πάρουμε:

```
ορμαθός
```

Μπορούμε να βάλουμε μόνο μία παράμετρο, τη θέση, οπότε το τέλος του υποορμαθού είναι το τέλος του αρχικού:

```
s2 = s1.substr( 5 );
cout << s2 << endl;
```

Αποτέλεσμα:

```
ορμαθός της C++
```

Παράδειγμα 1

Στο παράδειγμα της προηγούμενης παραγράφου, η αντιγραφή από τη *strIn* στη *strOut* μπορεί να γίνει ως εξής:

```
     strOut = "";    // κενό
     for ( k = strIn.length()-1; k >= 0; k = k-1 )
        strOut.append( strIn.substr(k,1) );
```

Παράδειγμα 2 ✍

Έχοντας δηλώσει:

```
string s1, s2, s3;
```

μετά την εκτέλεση των παρακάτω εντολών:

```
s1 = "ΦΑΣΟΥΛΙ ΦΑΣΟΥΛΙ ΓΕΜΙΖΕΙ ΤΟ ΣΑΚΟΥΛΙ";
s2 = s1.substr( 8, 8 );
s3 = s1.substr( s1.find('Τ'), 10 );
cout << s2 << s3 << endl;
s2 = s1.substr( 20, 20 );
cout << s2 << endl;
```

θα τυπωθούν τα ακόλουθα:

```
ΦΑΣΟΥΛΙ ΤΟ ΣΑΚΟΥΛΙ
ΖΕΙ ΤΟ ΣΑΚΟΥΛΙ
```

Από τη δεύτερη γραμμή βλέπεις ότι εσύ μπορεί να παραβείς τους περιορισμούς μήκους, αλλά η `substr` δεν θα υπερβεί τα όρια της αρχικής τιμής.
✍✍✍

10.12 Ρεύματα Από και Προς *string*

Μέσα στις πολλές δυνατότητες διαχείρισης εισόδου/εξόδου στοιχείων η C++ δίνει και τύπους **ρευμάτων από και προς *string*** (string streams). Έτσι, έχουμε τη δυνατότητα:

- Να παίρνουμε τα στοιχεία εισόδου –όπως και αν είναι αυτά– να κάνουμε ελέγχους εγκυρότητας και να γνωστοποιήσουμε στον χρήστη τυχόν προβλήματα ώστε να τα διορθώσει.
- Να ετοιμάζουμε (μορφοποιούμε) τα στοιχεία εξόδου όπως θέλουμε πριν τα βγάλουμε στην οθόνη ή σε κάποιο αρχείο text.

Αν βάλεις στο πρόγραμμά σου την οδηγία "`#include <sstream>`" σου δίνεται η δυνατότητα να δηλώσεις:

```
istringstream ssin( aString );
ostringstream ssout;
```

Το *ssin* είναι ένα ρεύμα σαν αυτά που ξέρουμε μόνο που δεν διαβάζει ούτε το πληκτρολόγιο ούτε κάποιο αρχείο αλλά το

```
string aString( "17 -375 145.78 31e4" );
```

Να ένα πρόγραμμα από τα παλιά (§2.3) κάπως αλλαγμένο:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main()
{
   int    k, j;
   double x, y, t;
   string aString( "17 -375 145.78 31e4" ), outString;
   istringstream ssin( aString );
   ostringstream ssout;

   ssin >> k >> j >> x >> y;
   ssout << k << ' ' << j << ' ' << x << ' ' << y;
   outString = ssout.str();
   cout << outString << endl;
}
```

Με την:

```
ssin >> k >> j >> x >> y;
```

διαβάζονται οι 17, -375, 145.78, 31*e*4 ως τιμές των *k*, *j*, *x*, *y* αντιστοίχως. Αν μετά από αυτήν προσπαθήσεις να διαβάσεις (`ssin >> t`) δεν θα τα καταφέρεις. Αν ελέγξεις την *ssin*`.eof()` θα τη βρεις `true`.

Με την

```
ssout << k << ' ' << j << ' ' << x << ' ' << y;
```

οι τιμές των μεταβλητών γράφονται σε έναν ενταμιευτή τύπου *string* που έχει το *ssout*. Αυτόν τον ενταμιευτή τον παίρνουμε με την `ssout.str()` και τον αντιγράφουμε σε μια άλλη μεταβλητή τύπου *string*. Φυσικά θα μπορούσαμε να γράψουμε κατ' ευθείαν:

```
cout << ssout.str() << endl;
```

Τα ρεύματα προς/από *string* έχουν τις ίδιες ιδιότητες (και δυνατότητες) με τα ρεύματα προς/από αρχεία. Επιπλέον όμως έχουν τη μέθοδο *str* για τον χειρισμό του ενταμιευτή:

- Με τις `ssin.str(aString)` και `ssout.str(aString)` βάζουμε ως ενταμιευτή την (τιμή που έχει εκείνη τη στιγμή η) *aString*. Για το *ssout* αυτό έχει μικρή σημασία αφού, με το πρώτο γράψιμο, ο ενταμιευτής θα «καθαριστεί».
- Με τις `as = ssin.str()` και `as = ssout.str()` αντιγράφουμε στην *as* (τύπου *string*) την τιμή του ενταμιευτή (εκείνη τη στιγμή). Στην περίπτωση αυτήν η πρώτη έχει μικρή χρησιμότητα αφού την έχουμε βάλει εμείς.

  Στη συνέχεια θα ασχοληθούμε με τον χειρισμό «αριθμητικών» ορμαθών.

10.12.1 Ορμαθοί και Αριθμοί

Οι πεπειραμένοι προγραμματιστές δεν γράφουν εντολές που να διαβάζουν αριθμητικά δεδομένα παρά μόνον στην περίπτωση που διαβάζουν αρχείο text που έχει ήδη ελεγχθεί. Συνήθως διαβάζουν ορμαθούς χαρακτήρων και στη συνέχεια τους μετατρέπουν σε αριθμητικές τιμές ελέγχοντας τυχόν λάθη. Ποιο είναι το κέρδος;

- Διαβάζοντας αριθμητικές τιμές μπορεί να συναντήσεις λάθη, π.χ. "`,`" αντί για "`.`" στην υποδιαστολή, ελληνικό έψιλον αντί για "`e`" στον εκθέτη και άλλα παρόμοια. Όπως ήδη ξέρεις, αυτά τα προβλήματα «διαταράζουν» (μπορεί και να διακόψουν) την ομαλή εκτέλεση του προγράμματός σου.
- 'Οταν διαβάζεις χαρακτήρες τα πάντα είναι δεκτά! Ο έλεγχος εγκυρότητας γίνεται από το πρόγραμμα και οι χειρισμοί λαθών είναι ευθύνη του προγραμματιστή.

Τα ρεύματα από τιμές *string* είναι ένα καλό εργαλείο για τη δουλειά αυτή.

Ας πούμε ότι διαβάζεις έναν ορμαθό χαρακτήρων στη μεταβλητή

```
   string aString;
// . . .
   getline( cin, aString, '\n' );
```

Έχεις ζητήσει από τον χρήστη να πληκτρολογήσει έναν ακέραιο –που θέλεις να αποθηκεύσεις στη μεταβλητή (`int`) *k*– αλλά αντί για "`7`" σου πληκτρολόγησε "`&`". Αυτή είναι και η τιμή της *aString*. Για να περάσουμε από την *aString* στην *k* θα χρειαστούμε ένα ρεύμα:

```
istringstream ssin;
```

στο οποίο θα βάλουμε ως ενταμιευτή την *aString*:

```
ssin.str( aString );
```

και από αυτό θα αποπειραθούμε να διαβάσουμε:

```
ssin >> k;
if ( ssin.fail() )  // αποτυχία
```

Αμέσως μετά ελέγχουμε, με τη γνωστή μας *fail*, αν τα καταφέρουμε.

Στην περίπτωσή μας θα μας πει ότι αποτύχαμε και θα πρέπει να δώσουμε το κατάλληλο μήνυμα προς τον χρήστη.

Ας πούμε τώρα ότι ο χρήστης πληκτρολόγησε "**1&**" και αυτή είναι η τιμή της *aString*. Στην περίπτωση αυτήν η **ssin.fail()** θα δώσει **false**, δηλαδή: όλα πήγαν καλά! Αλλά η *k* θα πάρει τιμή 1. Εδώ πώς πιάνουμε το λάθος; Προσπαθούμε να διαβάσουμε 1 χαρακτήρα. Αν τα καταφέρουμε θα πει ότι η τιμή της *k* δεν έχει προκύψει από ολόκληρη την τιμή της *aString*. Άρα έχουμε πρόβλημα. Αν δεν τα καταφέρουμε και πάρουμε **ssin.eof()** θα πει ότι όλα πήγαν καλά.

Ας τα δώσουμε όλα μαζί: Πρώτα οι δηλώσεις

```
   istringstream ssin;
   int    k;
   string aString;
   char   c;
```

Και μετά η ανάγνωση και οι έλεγχοι:

```
   getline( cin, aString, '\n' );
   ssin.str( aString );
   ssin >> k;
   if ( ssin.fail() )
   // λάθος
   else // κάτι διάβασε
   {
      ssin >> c;
      if ( ssin.eof() ) // OK, δεν έχει άλλα, τα διάβασε όλα
         cout << k << endl;
      else
      // λάθος
   }
```

Με παρόμοιο τρόπο διαβάζουμε και πραγματικές τιμές.

Στη συνέχεια δίνουμε ένα παράδειγμα για να δούμε τη χρήση αυτών που είπαμε στα προγράμματά μας.

Παράδειγμα

Ας πάρουμε το πρόγραμμα της ελεύθερης πτώσης, που είδαμε στην §2.12, και ας κάνουμε την ανάγνωση του ύψους μέσω ορμαθού χαρακτήρων:

```
#include <iostream>
#include <cmath>
#include <string>
#include <sstream>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec2, η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
   string s;
   istringstream ssin;
   bool done( false );
   char c;

// Διάβασε το h
   while ( !done || h < 0 )
   {
      cout << " Δώσε μου το αρχικό ύψος (h >= 0) σε m: ";
      getline( cin, s, '\n' );
      ssin.clear();
      ssin.str( s );   ssin >> h;
      if ( ssin.fail() )
         cout << "απαράδεκτοι χαρακτήρες (" << s << ")" << endl;
      else
      {
         ssin >> c;
         if ( ssin.eof() )
```

```
            done = true;
         else
            cout << "απαράδεκτος χαρακτήρας (\'" << c << "\')"
                 << endl;
      } // if ( !ssin.fail
   } // while

// (g == 9.81) && (0 <= h <= DBL_MAX)
// Υπολόγισε τα tP, vP
// ΤΑ ΥΠΟΛΟΙΠΑ ΙΔΙΑ
```

Όπως βλέπεις, αντί για την `cin >> h` έχουμε βάλει:

```
getline( cin, s, '\n' );
ssin.clear();
ssin.str( s );   ssin >> h;
```

και μετά έρχονται οι έλεγχοι. Πρόσεξε την "`ssin.clear()`": αν, μετά το λάθος που βρίσκουμε όταν προσπαθήσει να διαβάσει το "`*)`", δεν δώσουμε `ssin.clear()`, στην επόμενη προσπάθεια ανάγνωσης, θα βγάλει το ίδιο λάθος ακόμη και αν τα κάνουμε όλα σωστά. Όλα αυτά τα βάλαμε σε μια **while** από την οποίαν βγαίνει μόνον αν πάρει σωστή τιμή (και μη αρνητική). Αυτό είναι κάπως ανελαστικό. Καλύτερα θα ήταν αν δίναμε και κάποια διαφυγή αν ο χρήστης μετανιώσει και θέλει να τα παρατήσει (π.χ. να πιέσει το <esc>).

Στη συνέχεια βλέπεις ένα παράδειγμα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος (h >= 0) σε m: *)
απαράδεκτοι χαρακτήρες (*))
 Δώσε μου το αρχικό ύψος (h >= 0) σε m: 8o
απαράδεκτος χαρακτήρας ('o')
 Δώσε μου το αρχικό ύψος (h >= 0) σε m:  -80
 Δώσε μου το αρχικό ύψος (h >= 0) σε m:  80
 Αρχικό ύψος = 80 m
 Χρόνος Πτώσης = 4.03855 sec
 Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
```

Το αντίστροφο, η μετατροπή μιας αριθμητικής τιμής σε *string*, είναι πολύ απλή. Στο παρακάτω πρόγραμμα δίνουμε μερικά παραδείγματα από το Κεφ. 1 κάπως αλλαγμένα:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
   ostringstream sout;
   string aString;

   sout.precision( 8 );
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;

   sout.setf( ios_base::scientific, ios_base::floatfield );
   sout.precision(8);
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;

   sout.setf( ios_base::fixed, ios_base::floatfield );
   sout.precision(8);
   sout << 1234.5 << "  " << 123456.78 << "  "
        << 123456789.0123 << endl;
   aString = sout.str();
   cout << aString << endl;
}
```

Εδώ, αντί για το *cout*, γράφουμε στο ρεύμα *sout*. Τελικώς, αντιγράφουμε τον ενταμιευτή του *sout* στη μεταβλητή *aString*. Να η τιμή της:

```
1234.5  123456.78  1.2345679e+008
1.23450000e+003  1.23456780e+005  1.23456789e+008
1234.50000000  123456.78000000  123456789.01230000
```

Από εδώ και πέρα μπορείς να χειριστείς την τιμή αυτή με όλα τα εργαλεία που έχεις για τον τύπο *string*.

10.13 Η Κληρονομιά της C: Πίνακες με Χαρακτήρες

Θα δούμε εν συντομία τη διαχείριση κειμένου από τη C διότι:

1. θα τη δεις σε πολλά προγράμματα C++ (οι προγραμματιστές δύσκολα εγκαταλείπουν αυτά που ξέρουν) και
2. είναι χρήσιμη ακόμη και όταν δουλεύεις με τον τύπο *string*.

Όπως είδαμε στην εισαγωγή του κεφαλαίου:

- αποθηκεύουμε κείμενα σε πίνακες με στοιχεία τύπου `char`,
- στο τέλος του κειμένου βάζουμε *απαραιτήτως* έναν `char(0)`· αυτό πρέπει να το παίρνουμε υπόψη μας όταν δηλώνουμε τον πίνακα.

Αν δηλώσεις:

```
char a[100] = "πάν μέτρον εκατό πόντοι";
```

ή

```
char a[] = "πάν μέτρον εκατό πόντοι";
```

ο `char(0)` εισάγεται αυτομάτως. Αν όμως δηλώσεις:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ',
             'π','ό','ν','τ','ο','ι','\0' };
```

θα πρέπει να βάλεις το `char(0)` όπως βλέπεις εδώ.[8]

Είδαμε ακόμη ότι μπορείς να γράφεις στην οθόνη όπως ξέρουμε:

```
cout << a << endl;
```

και ότι μπορούμε να διαβάσουμε από το πληκτρολόγιο με τη *cin.getline()*. Αν

```
char a[100];
```

μπορείς να διαβάσεις κείμενο από το πληκτρολόγιο, ως τιμή του a, με την:

```
cin.getline( a, 100 );
```

Πάντως, η κλάση *istream* του *cin*, έχει και μια μορφή της *get()* με την οποία μπορείς να διαβάσεις κείμενο:

```
cin.get( a, 100 );
```

Και εδώ, το δεύτερο όρισμα, δείχνει το μέγιστο πλήθος χαρακτήρων που είναι δεκτοί. Και εδώ, στο μήκος περιλαμβάνεται και μια θέση για το `char(0)`. Αλλά, όταν δουλεύεις με τη *get* μην ξεχνάς να διαβάζεις και το «τέλος γραμμής» (<enter>, `'\n'`):

```
cin.get( a, 100 );  cin.get( ceol );
```

όπου η *ceol* είναι μεταβλητή τύπου `char`.

Στη συνέχεια θα δούμε μερικές από τις συναρτήσεις που έχει η C++ (από τη C) για να διαχειρίζεται πίνακες με κείμενα. Για να τις χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου:[9]

```
#include <string>
```

Μπορούμε να αλλάξουμε την τιμή της a με εντολή εκχώρησης σαν την:

---

[8] Το `'\0'` είναι που γράφεται συνήθως. Αν βάλεις `char(0)` ή απλώς `0` και πάλι είναι μια χαρά.

[9] Αν βάλεις "`#include <cstring>`" έχεις μόνον τις συναρτήσεις της C. Αν δεν βάλεις το "`c`" τα έχεις όλα.

```
a = "όποιος φυλάει τα ρούχα του είναι demodé";
```

Όχι! Για να αλλάξεις την τιμή της a χρειάζεται μια συνάρτηση της C++, η *strcpy()* (**str**ing **cpy**):

```
strcpy( a, "όποιος φυλάει τα ρούχα του είναι demodé" );
```

ή ακόμη:

```
strcpy( a, b );
```

αν ο b είναι ένας παρόμοιος πίνακας, π.χ.:

```
char b[] = "πάν μέτρον εκατό πόντοι";
```

Η *strncpy()* είναι σαν την *strcpy()*, αλλά έχει και τρίτη παράμετρο όπου περιμένει έναν φυσικό αριθμό που καθορίζει πόσοι χαρακτήρες θα αντιγραφούν από το δεύτερο όρισμα στο πρώτο. Θέλει λίγη προσοχή:

```
char a[50] = "abcdefghijklmnopq";

strncpy( a, "123456789", 15 );
cout << a << endl;
strcpy( a, "abcdefghijklmnopq" );
strncpy( a, "123456789", 5 );
cout << a << endl;
```

Αποτέλεσμα:

```
123456789
12345fghijklmnopq
```

Την πρώτη φορά ζητήσαμε να αντιγραφούν 15 χαρακτήρες ενώ το μήκος του δεύτερου ορίσματος είναι 9. Αντιγράφηκαν λοιπόν όλοι οι χαρακτήρες και το `char(0)` που σημειώνει το τέλος του ορμαθού· έτσι τιμή του a γίνεται το `123456789`. Τη δεύτερη φορά ζητήσαμε να αντιγραφούν 5 χαρακτήρες. Αυτοί αντικατάστησαν τους 5 πρώτους χαρακτήρες της τιμής του a. Αν θέλεις τιμή του a να γίνει το `12345` θα πρέπει να δώσεις:

```
strncpy ( a, "123456789", 5 );  a[5] = char( 0 );
```

Πολύ συχνά έχεις την εξής περίπτωση:

```
string cpps;
char   cs[ L ];
```

όπου *L* θετικός ακέραιος και θέλεις να αντιγράψεις στον cs την τιμή της *cpps* ή, εν πάσει περιπτώσει, «όσο χωράει». Αυτό γίνεται ως εξής:

```
strncpy ( cs, cpps.c_str(), L-1 );  cs[L-1] = char( 0 );
```

Δηλαδή: αντιγράφουμε *L* - 1 χαρακτήρες το πολύ και στην τελευταία θέση βάζουμε τον φρουρό ώστε ο cs να είναι διαχειρίσιμος. Αν το μήκος της τιμής της *cpps* είναι μικρότερο από *L* - 1 θα υπάρχει και άλλο `char(0)` πιο πριν και θα είναι ο πραγματικός φρουρός.

Μπορείς να συγκρίνεις δύο πίνακες με κείμενα, με τη *strcmp()*, που είναι σαν την `compare`. Η `strcmp(a, b)` επιστρέφει ακέραιη

- αρνητική τιμή *αν* η τιμή του a προηγείται λεξικογραφικώς της τιμής του b,
- *θετική* τιμή *αν η τιμή του* a *έπεται λεξικογραφικώς της τιμής του* b,
- *μηδέν* (0) *αν οι τιμές των* a *και* b *είναι ίσες*.

Η *strncmp* είναι σαν την *strcmp* αλλά παίρνει και τρίτο όρισμα, έναν φυσικό αριθμό, που μας λέει πόσους χαρακτήρες να συγκρίνει. Οι:

```
char a[50] = "abcdefghijklmnopq", b[] = "abcdrstuvwxyz";
cout << strcmp(a, b) << "  " << strncmp(a, b, 3) << endl;
```

δίνουν:

```
-13  0
```

Δηλαδή: η τιμή του a προηγείται λεξικογραφικώς της τιμής του b (strcmp(a, b) `==` -13 < 0), αλλά οι τρεις πρώτοι χαρακτήρες τους είναι ίδιοι (*strncmp*`(`a, b, 3`) ==` 0).

Πολύτιμη είναι και η *stricmp()* είναι σαν την *strcmp()* αλλά βλέπει κεφαλαία και πεζά (λατινικά) γράμματα ίσα. Οι:

```
char a[] = "firstName", b[] = "FirstName", c[] = "firstname";
cout << strcmp(a, b) << "  " << strcmp(a, c) << endl;
cout << stricmp(a, b) << "  " << stricmp(a, c) << endl;
```

δίνουν:

```
1  -1
0  0
```

Η *strnicmp()* είναι για την *stricmp()* ό,τι είναι η *strncmp* για την *strcmp*.

Η *strlen()* (**str**ing **len**gth), επιστρέφει τιμή τύπου `size_t` που είναι το μήκος του ορμαθού που έχουμε αποθηκεύσει, χωρίς το τελικό `char(0)`: οι

```
char a[100], b[] = "πάν μέτρον εκατό πόντοι";

strcpy( a, "όποιος φυλάει τα ρούχα του είναι demodé" );
cout << strlen(a) << "  " << strlen(b) << endl;
cout << strlen( "μέτρον άριστον" ) << endl;
```

δίνουν:

```
39  23
14
```

Ο ορμαθός a είναι κενός αν:

- είναι ίσος με `""` (`strcmp(a, "") == 0`) ή
- έχει μήκος μηδέν (`strlen(a) == 0`) ή
- έχει `char(0)` στην αρχή του (`a[0] == char(0)`).

Μπορείς να χρησιμοποιείς οποιαδήποτε από αυτές τις συνθήκες στα προγράμματά σου.

Η `strcat(a, b)` κάνει το ίδιο πράγμα με την `a.append(b)`: επισυνάπτει στο τέλος της τιμής του a την τιμή του b. Δες το παρακάτω:

```
char s1[] = "πέντε", s2[] = "στο χέρι παρά δέκα", s3[50];

strcpy( s3, "κάλλιο " );
strcat( s3, s1 );  strcat( s3, " και " );  strcat( s3, s2 );
strcat( s3, " και καρτέρει" );
cout << s3 << endl;
```

Αποτέλεσμα:

```
κάλλιο πέντε και στο χέρι παρά δέκα και καρτέρει
```

Υπάρχει και η *strncat()*, που παίρνει και τρίτο όρισμα, έναν φυσικό αριθμό, που λέει πόσους χαρακτήρες από το δεύτερο όρισμα θα επισυναφθούν στο τέλος του πρώτου.

Αυτά τα λίγα δεν εξαντλούν τα εργαλεία της C++ για τα "C-style strings". Μερικά ακόμη υπάρχουν στην επόμενη παράγραφο. Θα πρέπει όμως να τονίσουμε ότι τα δίνουμε πιο πολύ για να μη σου είναι άγνωστα αν τα δεις σε κάποιο πρόγραμμα. Στα προγράμματα που θα γράφεις εσύ θα πρέπει να δουλεύεις με την κλάση *string*, που είναι σαφώς πιο εύχρηστη.

10.13.1 Ορμαθοί C και Αριθμοί

Και η C σου δίνει δυνατότητα να γράφεις σε (και να διαβάζεις από) πίνακες χαρακτήρων. Ειδικώς για μετατροπές ορμαθών σε αριθμούς και αντιστρόφως σου δίνει ορισμένες συναρτήσεις που θα δούμε εδώ αλλά κι αργότερα. Οι δηλώσεις για τα εργαλεία αυτά υπάρχουν στο `cstdlib` και θα πρέπει να το περιλάβεις στο πρόγραμμά σου, όταν τα χρησιμοποιείς.

Το πιο απλά εργαλεία είναι οι συναρτήσεις *atof()* (**a**lphanumeric **to** **f**loat), *atoi()* (**a**lphanumeric **to** **i**nt) και *atoll()* (**a**lphanumeric **to** **l**ong). Αυτές παίρνουν έναν ορμαθό χαρακτήρων (ή έναν πίνακα με τέτοια τιμή) και μας δίνουν μια τιμή τύπου `double`, η πρώτη, τύπου `int` η δεύτερη και τύπου `long` η τρίτη. Π.χ. οι εντολές:

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main()
{
   char s1[] = "12345", s2[] = "1.23456789e10",
        s3[] = "1.23ab45678";
   long l;
   double d;

   l = atol( s1 );   d = atof( s2 );
   cout << l << "  " << d << endl;
```

θα δώσουν:

```
12345  1.23457e+10
```

Η μετάφραση από χαρακτήρες σε αριθμό σταματάει μόλις βρεθεί παράνομος χαρακτήρας. Π.χ. οι:

```
   l = atol( s3 );   d = atof( s3 );
   cout << l << "  " << d << endl;
```

θα δώσουν:

```
1  1.23
```

Τι έγινε εδώ; Η *atoll()*, προσπαθώντας να διαβάσει έναν ακέραιο από τον ορμαθό `1.23ab45678`, και αφού δεχτεί το `1`, βρίσκει την `'.'`, που δεν είναι δεκτή σε μια σταθερά τύπου `long`. Έτσι, μας επιστρέφει τιμή `1`. Για την *atof()* είναι δεκτοί οι χαρακτήρες μέχρι `1.23`· πρώτος παράνομος χαρακτήρας είναι ο `'a'`. Έτσι, μας επιστρέφει τιμή `1.23`.

Από αυτές τις συναρτήσεις δεν μπορείς να πάρεις παραπάνω πληροφορίες για το τι δεν πήγε καλά.

Πιο πλήρη δουλειά, σε περίπτωση λάθους, κάνουν οι *strtod()* και *strtol()* που θα δούμε αργότερα.

10.14 * Ο Τύπος *std*::*wstring*

Στην §4.7 είδαμε τον τύπο `wchar_t` που κάθε του τιμή αποθηκεύεται σε 16 δυαδικά ψηφία. Έτσι, όπως λέγαμε, «*έχει 65536 διαφορετικές τιμές, αντί για τις 256 του* `char` *και χρησιμοποιείται για την υλοποίηση του προτύπου Unicode.*» Και ακόμη «*Μια σταθερά τύπου* `wchar_t` *είναι μια σταθερά τύπου* `char` *με το πρόθεμα "*`L`*", π.χ.:*

`L'a'  L'%'  L'ξ'`»

Σε έναν πίνακα με στοιχεία τύπου `wchar_t` μπορούμε να βάζουμε ως τιμή έναν ορμαθό τιμών τύπου `wchar_t` είτε με τη δήλωση:

```
   wchar_t w[5]= L"αβγδ";
```

είτε αργότερα:

```
   wcscpy( w, L"wsx" );
```

Εδώ βλέπουμε τα εξής:

- Όπως μια σταθερά τύπου `char` με το πρόθεμα "`L`" γίνεται σταθερά τύπου `wchar_t`, έτσι και ένας ορμαθός τιμών τύπου `char` με το πρόθεμα "`L`" γίνεται ορμαθός τιμών τύπου `wchar_t`.
- Όπως η αντιγραφή ορμαθών τύπου `char` γίνεται με τη *strcpy*, η αντιγραφή ορμαθών τύπου `wchar_t` γίνεται με τη *wcscpy*.

Γενικώς, για κάθε συνάρτηση *strX* υπάρχει αντίστοιχη συνάρτηση *wcsX* για τη διαχείριση ορμαθών τύπου `wchar_t`.

Η C++ μας δίνει τον τύπο *std*::*wstring*, ίδιο σχεδόν με τον *std*::*string*, στις μεταβλητές του οποίου μπορούμε να αποθηκεύουμε ορμαθούς `wchar_t`. Έτσι, μπορούμε να ορίσουμε:

```
   wchar_t w[5]= L"αβγδ";
   wstring w0, w1( L"abc" ), w2( 4, L'q' ), w3( w );
```

Προσπαθώντας να δούμε τις τιμές των μεταβλητών *wstring* συναντούμε ένα πρόβλημα: Μια από τις διαφορές του *wstring* από τον *string* είναι ότι δεν μπορούμε να βγάλουμε την τιμή μεταβλητής *wstring* σε κάποιο ρεύμα με τον "<<". Δίνουμε μια πρόχειρη λύση –ας πούμε για τη *w1*– ως εξής:

```
int ndx;
for ( ndx = 0; ndx < w1.length(); ndx = ndx+1 )
   cout << static_cast<unsigned char>(w1[ndx]);
cout << endl;
```

Με αυτόν τον τρόπο μπορούμε να δούμε ότι το *w0* είναι κενό ενώ για τα *w1*, *w2* και *w3* παίρνουμε αντιστοίχως:

```
abc
qqqq
αβγδ
```

Μπορείς να αλλάξεις την τιμή μιας τέτοιας μεταβλητής με εκχώρηση, π.χ.:

```
w0 = L"qazw";
```

Με τη *c_str* μπορείς να βγάλεις το περιεχόμενο σε πίνακα και με τη *wcscpy* να το αντιγράψεις σε έναν πίνακα με στοιχεία `wchar_t`:

```
wcscpy( w, w1.c_str() );
```

Τέλος, για να μη σου μείνουν αμφιβολίες, δίνοντας:

```
cout << typeid(w1[0]).name() << "  " << sizeof(w1[0]) << endl;
```

παίρνεις αποτέλεσμα:[10]

```
wchar_t  2
```

10.15 Εν Κατακλείδι ...

Η C++ μας δίνει πολλά και αξιόλογα εργαλεία για να διαχειριστούμε κείμενο. Όλα περιλαμβάνονται (ή σχετίζονται) με την κλάση *string*. Μάθαμε λοιπόν:

- Να δηλώνουμε μεταβλητές τύπου *string*.
- Να αλλάζουμε την τιμή τους.
- Να διαβάζουμε από και να γράφουμε σε ρεύματα αρχείων (ή τα *cin*, *cout*) τιμές μεταβλητών *string*.
- Να συγκρίνουμε τιμές τύπου *string*.
- Να αναζητούμε κείμενο μέσα σε άλλο κείμενο.
- Να μετατρέπουμε τιμές *string* σε αριθμητικές και αντιστρόφως.

Είδαμε εν συντομία και τα αντίστοιχα εργαλεία της C· θα τα δεις να χρησιμοποιούνται σε πολλές περιπτώσεις.

Ασκήσεις

Α Ομάδα

10-1 Γράψε πρόγραμμα που θα διαβάζει έναν ορμαθό χαρακτήρων και θα αποφαίνεται αν είναι ή δεν είναι της μορφής *qq*. Π.χ. οι "αβγαβγ", "zppquzppqu" είναι της μορφής *qq*, ενώ οι "αβγδε", "zppqqppz" δεν είναι.

10-2 Ξαναγράψε το πρόγραμμα του παραδείγματος της §10.5, έτσι ώστε να διαβάζει τα ονόματα των γλωσσών από αρχείο text.

[10] Borland C++ 5.5.

10-3 Ξαναγράψε το πρόγραμμα του παράδ. 1 της §8.10 έτσι ώστε να διαβάζει από το αρχείο ολόκληρες γραμμές και όχι χαρακτήρα προς χαρακτήρα.

10-4 Κάνε το ίδιο για το πρόγραμμα του παραδ. 2 της §8.10.

Β Ομάδα

10-5 Γράψε μια:

```
string getRight( string s, int n )
```

που θα επιστρέφει ως τιμή τους *n* τελευταίους (δεξιότερους) του *s*. Αν $n \le 0$ η τιμή θα είναι κενή, ενώ αν $n \ge s.length()$ η τιμή θα είναι αυτή της *s*.

10-6 Ο αριθμός μιας πιστωτικής κάρτας δημιουργείται ως εξής:

- Τα πρώτα ψηφία χαρακτηρίζουν την κάρτα. Στον Πίν. 10-1 βλέπεις τα χαρακτηριστικά για ορισμένες πολύ γνωστές πιστωτικές κάρτες.
- Στη συνέχεια υπάρχουν τα ψηφία που δείχνουν την τράπεζα.
- Μετά υπάρχει ο αριθμός του πελάτη.
- Το τελευταίο ψηφίο είναι **ψηφίο ελέγχου** (check digit) επιλεγμένο έτσι ώστε να ισχύει η συνθήκη εγκυρότητας LUHN.

Τύπος Κάρτας	**Προθεμα**	**Μήκος**	**Αλγόριθμος Ψηφίου Ελέγχου**
MASTERCARD	51-55	16	mod 10
VISA	4	13, 16	mod 10
AMEX	34, 37	15	mod 10
Diners Club/ Carte Blanche	300-305 36, 38	14	mod 10
Discover	6011	16	mod 10
enRoute	2014 2149	15	any
JCB	3	16	mod 10
JCB	2131 1800	15	mod 10

Πίν. 10-1 Στη στήλη **Πρόθεμα** βλέπεις το χαρακτηριστικό κάθε κάρτας και στη στήλη **Μήκος** το μήκος του αριθμού που χρησιμοποιεί.

Η συνθήκη εγκυρότητας LUHN υπολογίζεται ως εξής:

Βήμα 1: Πολλαπλασίασε επί δύο τα ψηφία που έχουν άρτιες θέσεις (θέση 1 η τελευταία, 2 η προτελευταία κ.ο.κ.)

Βήμα 2: Πρόσθεσε όλα τα ψηφία των αριθμών που προέκυψαν από τους διπλασιασμούς και αυτά που βρίσκονται σε μονές θέσεις.

Αν το άθροισμα είναι πολλαπλάσιο του 10 ο αριθμός είναι έγκυρος.

Παράδειγμα

Έστω ότι έχουμε τον αριθμό:

49927398716

Βήμα 1:

```
        4 9 9 2 7 3 9 8 7 1 6
         x2  x2  x2  x2  x2
------------------------------
         18   4   6  16   2
```

Βήμα 2: 4 +(1+8)+ 9 + (4) + 7 + (6) + 9 +(1+6) + 7 + (2) + 6 = 70

Άρα ο αριθμός είναι έγκυρος.

α) Γράψε συνάρτηση *isValidCrCardNum* που θα παίρνει (ως όρισμα) μια τιμή τύπου *string* που θα είναι αριθμός πιστωτικής κάρτας, θα εξετάζει το ψηφίο ελέγχου και θα επιστρέφει ως τιμή `true` αν ο αριθμός είναι έγκυρος (σωστό ψηφίο ελέγχου) και `false` αν δεν είναι.

β) Γράψε συνάρτηση *appendChkDigit* που θα παίρνει (ως όρισμα) μια τιμή τύπου *string* που θα είναι αριθμός πιστωτικής κάρτας χωρίς το τελευταίο ψηφίο. Η συνάρτηση θα συμπληρώνει το τελευταίο ψηφίο με βάση τα όσα είπαμε παραπάνω.

Γράψε πρόγραμμα που θα σου επιτρέπει να δοκιμάσεις τις συναρτήσεις που έγραψες.

10-7 Γράψε πρόγραμμα που θα διαβάζει ένα κείμενο από ένα αρχείο text και θα το αποθηκεύει σε μια μεταβλητή τύπου `string`. Κατά την ανάγνωση θα αντικαθιστά κάθε αλλαγή γραμμής με ένα διάστημα (κενό). Στη συνέχεια θα διαβάζει λέξεις από το πληκτρολόγιο και θα μας λέει πόσες φορές υπάρχει κάθε μια από αυτές στο κείμενο. Θα τελειώνει όταν πληκτρολογήσουμε τη λέξη `zzz`.

Γ Ομάδα

10-8 (*Πιο ρεαλιστική παραλλαγή της άσκ.* 8-7) Ένα αρχείο text, με όνομα στο δίσκο misthoi.txt, είναι γραμμένο ως εξής: Σε κάθε γραμμή έχει ένα επώνυμο εργαζόμενου και τρεις αριθμούς ως εξής:

```
ΣΙΩΠΑΚΗΣ\t1234.5\t7\t2\n
```

(το '`\t`' παριστάνει τον χαρακτήρα tab και το '`\n`' την αλλαγή γραμμής.) Ο πρώτος αριθμός (πραγματικός) –εδώ 1234.5– είναι ο βασικός μισθός ενός υπαλλήλου, ο δεύτερος –εδώ 7– ο αριθμός ετών υπηρεσίας του ίδιου υπαλλήλου και ο τρίτος –εδώ 2– ο αριθμός των παιδιών του. Το πλήθος των στοιχείων είναι άγνωστο.

Το περιεχόμενο του αρχείου δεν έχει ελεγχθεί. Έτσι, μπορεί να έχει:

- Κενές γραμμές (μικρό το κακό).
- Γραμμές από τις οποίες λείπουν στοιχεία.
- Λαθεμένα στοιχεία που παραβιάζουν κάποια από τα παρακάτω:
  - Το επώνυμο του εργαζόμενου δεν πρέπει να είναι κενό.
  - 800 ≤ βασικός μισθός ≤ 3500,
  - 0 ≤ χρόνια υπηρεσίας ≤ 35,
  - 0 ≤ αριθμός παιδιών ≤ 12.

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα δημιουργεί ένα νέο –το log.txt– με καταγραφή μιας γραμμής για κάθε λάθος που θα βρίσκει. Π.χ.:

```
Line 7 salary: 35650
Line 7 number of children: -4
Line 11 number of years: 51
Line 18 empty
Line 26 salary: 955000
Line 28 incomplete line
Line 36 name empty
```

Υπόδ.: Διάβαζε ολόκληρες γραμμές (μέχρι να βρεις '`\n`') με τη *getline*. Σε κάθε γραμμή θα πρέπει να υπάρχει το "`\t`" ακριβώς 3 φορές.