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:

$f: \mathbb{R}^{*} \to \mathbb{R}$

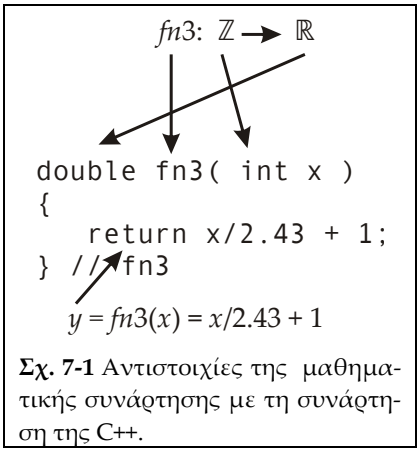
Όπως είδες στα παραδείγματα, στα μαθηματικά μπορούμε να περιορίσουμε το σύνολο αφετηρίας ώστε να γίνει ίσο με το πεδίο ορισμού. Στον προγραμματισμό αυτό δεν είναι πάντοτε εφικτό.

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

$dRound(x, n) - 0.5\cdot10^{-n} < x \le dRound(x, n) + 0.5\cdot10^{-n}$