κεφάλαιο

14

Συναρτήσεις III

Ο στόχος μας σε αυτό το κεφάλαιο:

Να καλύψουμε μερικά υπόλοιπα σχετικά με συναρτήσεις. Μερικά από αυτά είναι πολύ σημαντικά:

- Οι συναρτήσεις *ανάκλησης* (callback).
- Η *επιφόρτωση* συναρτήσεων και τελεστών.
- Τα *περιγράμματα* (templates) συναρτήσεων.
- Οι *εξαιρέσεις* (exceptions) και η διαχείρισή τους.

Προσδοκώμενα αποτελέσματα:

Η ενσωμάτωση της χρήσης των παραπάνω τεχνικών στην ανάπτυξη λογισμικού που κάνεις θα πάρει καιρό· η πλήρης αξιοποίησή τους έρχεται με τη σχετική ωριμότητα. Είναι όμως ένα σημαντικό βήμα προς την επαγγελματική διαδικασία ανάπτυξης λογισμικού. Ακόμη, σε προετοιμάζουν για το επόμενο σχετικό βήμα που θα δούμε στο τελευταίο κεφάλαιο του *Μέρους Β*.

Έννοιες κλειδιά:

- *συναρτήσεις* `inline`
- *προκαθορισμένες τιμές παραμέτρων*
- *παράμετρος-συνάρτηση*
- *βέλος προς συνάρτηση*
- *συνάρτηση ανάκλησης* (*callback*)
- *επιφόοτωση συναρτήσεων*
- *επιφόοτωση τελεστών*
- *γενικές συναρτήσεις*
- *περιγράμματα συναρτήσεων*
- *παράμετροι της* ***main***
- *στοίβα* (*stack*)
- *εξαιρέσεις*
- *αναδρομή*

Περιεχόμενα:

14.1 Συναρτήσεις "inline"

Η κλήση μιας συνάρτησης χρειάζεται κάποιο χρόνο εκτέλεσης (πέρα από το χρόνο εκτέλεσης των εντολών που έχει στο σώμα της). Μια παλιά (και παλιομοδίτικη) προγραμματιστική συνταγή λέει: αν μια συνάρτηση καλείται πολύ συχνά μη τη γράφεις ως χωριστή συνάρτηση αλλά όπου υπάρχει η κλήση της αντίγραψε όλες τις εντολές της. Κάτι τέτοιο βέβαια «φουσκώνει» πολύ το πρόγραμμα και δεν είναι εύκολα διαχειρίσιμο. Το πρόβλημα λύθηκε, από πολύ παλιά, με τις **μακροσυναρτήσεις** (macros): πρόκειται για συναρτήσεις που γράφονται μεν χωριστά αλλά ο μεταγλωττιστής αντικαθιστά τις κλήσεις τους με τις (μεταγλωττισμένες) εντολές τους.

Αν και σήμερα το πρόβλημα με τις κλήσεις συναρτήσεων δεν είναι και τόσο σοβαρό, η C++[1] μας δίνει τη δυνατότητα να ζητήσουμε από τον μεταγλωττιστή να χειριστεί κάποια συνάρτηση ως μακροσυνάρτηση. Αν γράψεις, π.χ.:

```
inline int max( int x, int y )
{
   int fvx;

   if ( x > y )  fvx = x;
          else  fvx = y;
   return fvx;
} // max int
```

ζητάς από τον μεταγλωττιστή να χειριστεί τη συνάρτηση ως μακροσυνάρτηση. Δεν είναι καθόλου σίγουρο ότι ο μεταγλωττιστής θα σε υπακούσει: υπακούει όταν η συνάρτηση είναι αρκετά απλή. Στην περίπτωσή μας η **if** μπορεί να τον αποτρέψει. Αν όμως γράψεις:

```
inline int max( int x, int y )  {  return (x > y) ? x : y;  }
```

είναι σίγουρο ότι θα σε υπακούσει.

14.2 Προκαθορισμένες Τιμές Παραμέτρων

Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση, ας την πούμε *inc*, με τις εξής προδιαγραφές:

```
void inc( int& x, int s )
```

[1] Η προδιαγραφή "`inline`" έχει περιληφθεί και στην τυποποίηση της C του 1999 (ISO/ IEC 1999).

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }
{ x == x_0 } `inc( x );` { x == $x_0 + 1$ }
Δηλαδή, η *inc* θα μπορεί να καλείται άλλοτε με ένα όρισμα και άλλοτε με δύο![2]

Ας πάρουμε μόνον την:

{ x == x_0 } `inc( x, a );` { x == $x_0 + a$ }

Αυτήν μπορούμε να τη γράψουμε; Εύκολα:

```
void inc( int& x, int s )
{
   x += s;
} // inc
```

Με μια μικρή μετατροπή μπορούμε να έχουμε αυτό που μας ενδιαφέρει:

```
void inc( int& x, int s = 1 )
{
   x += s;
} // inc
```

Τι λέει η νέα επικεφαλίδα; Αν δεν δοθεί δεύτερο όρισμα –αντίστοιχο της *s*– η *s* να θεωρηθεί ότι είναι 1.

Έτσι, αν στο πρόγραμμά μας δώσουμε:

```
   int p;

   p = 5;   inc( p, 2 );   cout << p << endl;
   p = 5;   p += 2;   cout << p << endl;
   p = 5;   inc( p );   cout << p << endl;
   p = 5;   ++p;   cout << p << endl;
```

θα πάρουμε:

```
7
7
6
6
```

Αν λοιπόν γράψουμε καταλλήλως μια συνάρτηση μπορούμε να την καλούμε παραλείποντας μερικά ή όλα τα ορίσματα.

Είναι φανερό ότι,

- ***Στην κλήση μιας συνάρτησης, δεν μπορείς να παραλείψεις ορίσματα που αντιστοιχούν σε παραμέτρους αναφοράς.***

Ακόμη:

- ***Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.***

Φυσικά, η συνάρτηση θα πρέπει να είναι γραμμένη καταλλήλως. Ας δούμε τι σημαίνει αυτό με ένα παράδειγμα. *Δεν επιτρέπεται να γράψεις*:

```
int f( double x, int y = 0, float z )
```

ενώ *μπορείς να γράψεις*:

```
int f( double x, int y = 0, float z = 1.5 )
```

Η κλήση της *f* μπορεί να είναι:

```
f( 1.56 )    f( a+b, a, b/2 )   f( p+0.5, n+1 )
```

- Στην πρώτη περίπτωση το 1.56 θα γίνει τιμή της *x*. Οι *y* και *z* θα έχουν τις ερήμην καθορισμένες τιμές τους 0 και 1.5 αντιστοίχως.
- Στη δεύτερη περίπτωση η τιμή της *a+b* θα γίνει τιμή της *x*, η τιμή της *a* θα γίνει τιμή της *y* και η τιμή της *b*/2 θα γίνει τιμή της *z*.

[2] Δηλαδή θα δουλεύει άλλοτε ως "`x += a`" και άλλοτε ως "`++x`" ή "`x += 1`".

- Στην τρίτη περίπτωση οι τιμές των p+0.5 και n+1 θα γίνουν τιμές των x και y αντίστοιχα. Η z θα ξεκινήσει με την ερήμην καθορισμένη τιμή της 1.5.

14.3 Παράμετρος – Συνάρτηση

Θέλουμε να γράψουμε μια συνάρτηση που θα υπολογίζει, προσεγγιστικά, μια λύση της εξίσωσης $f(x) = 0$ στο διάστημα $[a_0,b_0]$. Μια πολύ απλή και σίγουρη μέθοδος είναι αυτή της **διχοτόμησης**[3] (bisection).

Η μέθοδος αυτή προϋποθέτει ότι η f είναι συνεχής στο διάστημα $[a_0,b_0]$ και ότι η $f(a_0)f(b_0) \le 0$. Αν m_0 είναι το μέσο του διαστήματος $[a_0,b_0]$, ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
αλλιώς
   ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

Τα παραπάνω γράφονται εύκολα σε C++ ως εξής:

```
a = a0;  b = b0;
m = (a + b) / 2;
if ( f(a)*f(m) <= 0.0 )  b = m;
                    else  a = m;
```

Το «*ψάχνουμε για τη λύση στο διάστημα* $[a_0,m_0]$» το μεταφράσαμε «`b = m`», που βέβαια είναι σωστό. Παρακάτω θα δεις ότι μπορεί να μεταφραστεί και αλλιώς (Άσκ. 14-3).

Στο νέο διάστημα ψάχνουμε με τον ίδιο τρόπο. Πρόσεξε ότι οι τιμές των a, b μεταβάλλονται έτσι που το εύρος του διαστήματος να ελαττώνεται. Πότε σταματούμε; Όταν η τιμή της m προσεγγίσει ικανοποιητικά την τιμή της λύσης. Αλλά πόσο είναι το μέγιστο σφάλμα που μπορεί να έχουμε εδώ; Η απόσταση κάθε σημείου του διαστήματος $[a,b]$ από το μέσο του m είναι $|b - a|/2$ το πολύ. Αν ρ είναι η ακριβής τιμή της λύσης, τότε θα έχουμε και $|m - \rho| \le |b - a|/2$. Αν λοιπόν θέλουμε να υπολογίσουμε τη λύση με ακρίβεια ε, θα μικρύνουμε το διάστημα τόσο[4] ώστε $|b - a|/2 < \varepsilon$.

Ο αλγόριθμός μας θα είναι:

```
a = a0;  b = b0;
while ( fabs(b - a)/2 >= epsilon )
{
   m = (a + b) / 2;
   if (f(a)*f(m) <= 0.0)  b = m;
                     else  a = m;
} // while
root = m;
```

Όλα αυτά θα γίνονται με την προϋπόθεση ότι $f(a_0)f(b_0) \le 0$. Αν δεν ισχύει αυτή η συνθήκη για το αρχικό διάστημα, καλύτερα να μην αποπειραθούμε να προχωρήσουμε, αλλά να στείλουμε στο πρόγραμμα που καλεί τη διαδικασία ένα σήμα λάθους. Αυτό συνήθως γίνεται με μια παράμετρο, ας την πούμε *errCode*, μέσω της οποίας επιστρέφεται τιμή 0 αν όλα πήγαν καλά και 1 αν το αρχικό διάστημα ήταν λάθος.

Τι είδους συνάρτηση θα γράψουμε: με τύπο ή χωρίς τύπο; Η συνάρτησή μας θα τροφοδοτείται με την f και τα a_0, b_0 και ε και θα επιστρέφει την προσέγγιση της ρίζας και την *errCode*. Αφού θα επιστρέφει δύο τιμές θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο:

```
void bisection( ...
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

---

[3] Περισσότερα για τη μέθοδο αυτή σε οποιοδήποτε βιβλίο Αριθμητικής Ανάλυσης αλλά και στο βιβλίο Ανάλυσης της Γ′ Λυκείου. (Κατσαργύρης et al. 1992).
Δες και στο `http://en.wikipedia.org/wiki/Bisection_method`.

[4] Η αλήθεια είναι ότι συνήθως έχουμε πετύχει την ακρίβεια που θέλουμε πιο πριν.

Οι *a*0, *b*0, *epsilon* είναι παράμετροι τιμής, μια και θα έχουν πληροφορίες που δίνονται *προς* τη διαδικασία. Η *root* θα μεταφέρει τη λύση της εξίσωσης *από* τη διαδικασία προς το πρόγραμμα που την κάλεσε. Το ίδιο και η *errCode*. Και οι τελείες στην αρχή; Μα εκεί θα πρέπει να μπει μια παράμετρος που θα μας δίνει την *f* ! Τι παράμετρος θα είναι αυτή; Η C++ λέει: εκεί θα πρέπει να βάλουμε μια παράμετρο *βέλος* προς μια συνάρτηση που επιστρέφει τιμή τύπου **double**. Δηλαδή κάτι σαν:

```
double* f(double)
```

Το "**(double)**" μετά το *f* δείχνει ότι η *f* είναι συνάρτηση με μια παράμετρο, τύπου **double**. Τώρα θα γυρίσεις στον πίνακα προτεραιοτήτων (Παράρ. Ε) και θα δεις ότι πρώτα αναγνωρίζεται το "**f(double)**" και μετά το "*". Αλλά αφού το *f* είναι βέλος, το "**f(double)**" δεν έχει νόημα. Το "***f**" είναι συνάρτηση και νόημα έχει το "**(*f)(double)**". Άρα, το σωστό είναι:

```
double (*f)(double)
```

και η επικεφαλίδα είναι:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
```

Αλλά και *μέσα στο σώμα της συνάρτησης* –κατ' αρχήν– *δεν έχει νόημα να γράφουμε*: "**f(a)**" *αλλά*: "**(*f)(a)**". Να λοιπόν η συνάρτηση:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
{
   double m;

   if ( (*f)(a)*(*f)(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b-a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( (*f)(a)*(*f)(m) <= 0.0 )  b = m;
                                   else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Η C++ σου κάνει ένα δώρο· *μπορείς να γράφεις απλώς*:

```
   if ( f(a)*f)(b) > 0 )
```

και

```
         if ( f(a)*f(m) <= 0.0 )  b = m;
```

αντιστοίχως.

Τώρα να δούμε πώς θα καλέσουμε τη *bisection()*. Ας πούμε ότι θέλουμε λύση της εξίσωσης $x - ln(x) - 2 = 0$ στο διάστημα [0.1, 1]. Κατ' αρχάς, γράφουμε μια συνάρτηση που υπολογίζει την τιμή της $x - ln(x) - 2$:

```
double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και –π.χ. μέσα στη **main**– βάζουμε την εντολή:

```
   bisection( &q, 0.1, 1.0, 1e-5, riza, errCode );
```

Τι σημαίνει **&q**; Ας πούμε ότι είναι είναι ένα βέλος προς τη θέση της μνήμης όπου είναι γραμμένες οι εντολές της *q()*.

Υπάρχει όμως και εδώ το δώρο της C++· σου επιτρέπει να παραλείψεις το **&** και να γράψεις απλούστερα:

```
   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
```

Αν θέλεις να βρείς τη λύση της εξίσωσης συνx = 0 στο διάστημα [0,π] δώσε την εντολή:

```
   bisection( cos, 0.0, 4*atan(1), 1e-5, riza, errCode );
```

Φυσικά, στην περίπτωση αυτή, δεν θα πρέπει να ξεχάσεις να βάλεις "**#include cmath**" όπου υπάρχει το υπόδειγμα των *cos()* και *atan()*.

Να ένα παράδειγμα χρήσης της *bisection()*:

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

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   const double pi( 4*atan(1.0) );
   double riza;
   int   errCode;

   bisection( q, 0.1, 1.0, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
   bisection( cos, 0.0, pi, 1e-5, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
double q( double x )
   // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Στη συνέχεια βλέπεις το αποτέλεσμα του παραπάνω προγράμματος.

```
 Ρίζα = 0.1585983
 Ρίζα = 1.570784
```

Παρατηρήσεις:▶ σχετικώς με τη μέθοδο της διχοτόμησης

1. Πρόσεξε ότι μετά από n επαναλήψεις το σφάλμα είναι το πολύ $|b - a|/2^n$. Επομένως, θα μπορούσαμε ισοδυνάμως αντί για το ε να δίνουμε στη συνάρτηση κάποιο *nMax*.
2. Αν θελήσεις να χρησιμοποιήσεις τη *bisection()* πρόσεξε το εξής: το κριτήριο τερματισμού $|b - a|/2 < \varepsilon$ συχνά δεν είναι αρκετό. Ένα άλλο κριτήριο που μπορεί να χρησιμοποιείται (μαζί με αυτό που έχουμε ή –ισοδυνάμως– μαζί με μέγιστο αριθμό επαναλήψεων) είναι το $|f(x)| < \varepsilon'$.◀

14.4 * Συναρτήσεις και Βέλη – Συναρτήσεις Ανάκλησης

Το είδαμε λοιπόν και αυτό: βέλος προς συνάρτηση! Ας κάνουμε όμως μια αφαίρεση: να δούμε τι πρόβλημα είχαμε να λύσουμε και πώς το λύσαμε.

Πολύ συχνά έχουμε να γράψουμε πρόγραμμα που δεν έχει ορισμένες τις προδιαγραφές του μέχρι την τελευταία λεπτομέρεια. Στην περίπτωσή μας έπρεπε να γράψουμε συνάρτηση για την προσεγγιστική επίλυση της εξίσωσης $f(x) = 0$ χωρίς να ξέρουμε ποια ακριβώς είναι η f. Όταν γράφουμε τη *bisection* το μόνο που ξέρουμε για την f είναι ότι:

f : `double` → `double`

Τι κάνουμε λοιπόν; Δίνουμε λύση στο πρόβλημά μας –δηλαδή γράφουμε συνάρτηση– *με παράμετρο* την *f*. Και το εργαλείο που χρησιμοποιούμε για την υλοποίηση είναι το *βέλος προς τη συνάρτηση f*. Λέμε ότι η παράμετρος *f* περιμένει μια **συνάρτηση ανάκλησης** (callback function)[5]. Στο παράδειγμά μας, τέτοιες είναι η *q* και η *cos*.

Ας δούμε άλλο ένα παράδειγμα από την API των Windows[6]. Οι προγραμματιστές που την έγραψαν είχαν το εξής πρόβλημα: Όταν κάποιος προγραμματιστής (που χρησιμοποιεί την API για να γράψει μια εφαρμογή Windows) θέλει να βγάζει το πρόγραμμά του ένα **πλαίσιο διαλόγου** (dialog box) θα πρέπει να το σχεδιάσει και να γράψει μια συνάρτηση που θα καθορίζει όλη την αλληλεπίδρασή του με τον χρήστη. Αλλά ο προγραμματιστής της εφαρμογής δεν θα πρέπει να ασχολείται με το πώς θα εμφανιστεί το πλαίσιο· αυτό γίνεται με πάγια διαδικασία. Έγραψαν λοιπόν μια συνάρτηση:

```
int DialogBox( HINSTANCE hInstance,
               PCTSTR pTemplate,    // όνομα του πλαισίου διαλόγου
               HWND hWndParent,
               DLGPROC pDialogFunc )    // βέλος προς τη συνάρτηση
                           // διαχείρισης του πλαισίου διαλόγου
```

Στη δεύτερη παράμετρο περιμένει το όνομα του πλαισίου διαλόγου (από αυτό θα βρει το σχέδιο). Στην τέταρτη παράμετρο περιμένει βέλος προς τη συνάρτηση διαχείρισης του πλαισίου διαλόγου που είναι μια συνάρτηση ανάκλησης.

Η *DialogBox*, αφού κάνει αυτά που γίνονται για κάθε πλαίσιο διαλόγου, θα καλέσει τη συνάρτηση ανάκλησης για τη διαχείριση του συγκεκριμένου πλαισίου. Παράδειγμα επικεφαλίδας τέτοιας συνάρτησης:

```
bool CALLBACK about( HWND hDlg, UINT iMessage,
                     WPARAM wParam, LPARAM lParam )
```

ή, απλούστερα:

```
bool about( HWND hDlg, unsigned int iMessage,
            unsigned short int wParam, long int lParam )
```

Και η αντίστοιχη κλήση:

```
        DialogBox( hInstance, "AboutBox", hWnd, &about );
```

ή:

```
        DialogBox( hInstance, "AboutBox", hWnd, about );
```

Τα παραπάνω δείχνουν παραδείγματα που χρησιμοποιείς βέλη προς συναρτήσεις (και δεν μπορείς να αποφύγεις.) Μπορεί να δεις και άλλες «εξωτικές» (ή, για άλλους, «τρομακτικές») περιπτώσεις αλλά να έχεις υπόψη σου ότι αναφέρονται κατά κύριο σε προγραμματισμό στη γλώσσα C. Στη C, με παρόμοιες τεχνικές, υλοποιούνται «γενικές» συναρτήσεις. Η C++ σου δίνει εργαλεία που (θα τα δούμε στη συνέχεια) που σου επιτρέπουν να αποφεύγεις τις «εξωτικές» περιπτώσεις.

Στο (Haendel 2001) θα βρεις πολλά πράγματα για βέλη προς συναρτήσεις, για τα προβλήματα που χρησιμοποιούνται και σχετικά εργαλεία και τεχνικές της C++. Στο –πιο απλό– (Hosey 2007) θα βρεις πολλά πράγματα για βέλη στη C και στη C++.

Εδώ θα δώσουμε μερικά παραδείγματα άλλων συνδυασμών συναρτήσεων και βελών που μπορεί να συναντήσεις.

Ξεκινούμε με την πιο απλή αλλά και πολύ συνηθισμένη περίπτωση: *συνάρτηση που επιστρέφει βέλος*. Τέτοιες συναρτήσεις έχουμε δει ήδη: *strcpy*, *strcat* και άλλες παρόμοιες. Στις περιπτώσεις αυτές όμως το αποτέλεσμα «έβγαινε» από δυο μεριές: ως αποτέλεσμα της συνάρτησης και ως τιμή της πρώτης παραμέτρου. Θα γράψουμε τώρα μια συνάρτηση:

```
const char* myStrLT( const char* s1, const char* s2 )
```

---

[5] Θα διαβάσεις αλλού ότι «συναρτήσεις ανάκλησης είναι αυτές που καλούμε με βέλος».

[6] Windows Application Program Interface (διεπαφή προγράμματος εφαρμογής).

που θα επιστρέφει ως τιμή βέλος προς τον ορμαθό –*s1* ή *s2*– που προηγείται λεξικογραφικώς. Προφανώς, είναι πολύ απλή:

```
const char* myStrLT( const char* s1, const char* s2 )
{
   const char* fv( s1 );
   if ( strcmp(s2, fv) < 0 ) fv = s2;
   return fv;
} // myStrLT
```

Συναρτήσεις που θα βγάζουν βέλη προς πίνακες διαφόρων τύπων –και όχι μόνον `char`– θα γράψουμε αρκετές.

Ας έλθουμε τώρα στο δεύτερο παράδειγμα. Τι είναι εκείνο το "`DLGPROC`"; Αν ψάξεις στο winuser.h θα δεις ότι είναι όνομα τύπου. Σε απλουστευμένη μορφή:

```
typedef int (*DLGPROC)( HWND, unsigned int,
                        unsigned short int, long int );
```

Δηλαδή έχουμε έναν τύπο βέλους προς συνάρτηση! Ο τύπος του `&`*about* είναι ακριβώς *DLGPROC*.

Παρομοίως, στην προηγούμενη παράγραφο, θα μπορούσαμε να ορίσουμε:

```
typedef double Real1( double );
```

Ο *Real1* είναι ο τύπος που περιλαμβάνει τις συναρτήσεις με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας τον θα μπορούσαμε να γράψουμε:

```
void bisection( Real1* f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Θα μπορούσαμε επίσης να ορίσουμε:

```
typedef double (*PReal1)( double );
```

Ο *PReal1* είναι ο τύπος που περιλαμβάνει *βέλη* προς συνάρτηση με μια παράμετρο τύπου `double` και επιστρεφόμενη τιμή ίδιου τύπου. Χρησιμοποιώντας αυτόν τον τύπο η επικεφαλίδα της *bisection* γράφεται:

```
void bisection( PReal1 f,
                double a0, double b0, double epsilon,
                double& root, int& errCode )
```

Να δούμε τώρα *έναν πίνακα συναρτήσεων*! Ας πούμε ότι θέλουμε έναν πίνακα συναρτήσεων[7] τύπου *Real*1. Εδώ υπάρχει ένα πρόβλημα: αν δοκιμάσεις να δηλώσεις:

```
   Real1 funcArr[3];
```

ο μεταγλωττιστής δεν θα το επιτρέψει. Γιατί; Όλα τα στοιχεία του πίνακα πρέπει να έχουν το ίδιο μέγεθος· πόσο θα είναι αυτό; Αν δηλώσεις:

```
   PReal1 funcArr[3];
```

όλα πάνε καλά!

Γυρίζουμε στο παράδειγμα της προηγούμενης παραγράφου και το εμπλουτίζουμε με μια ακόμη συνάρτηση (για τη λύση της εξίσωσης $e^x = x+2$):

```
double em2( double x )
{
   return exp(x)-x-2;
} // em2
```

Έστω ότι θέλουμε να προσεγγίσουμε ρίζες των εξισώσεων:

$$x = lnx + 2 \text{ για } x \in [0.1,1], \quad συνx = 0 \text{ για } x \in [0,\pi], \quad e^x = x+2 \text{ για } x \in [0,2]$$

Μπορούμε να το κάνουμε ως εξής;

```
   const double pi( 4*atan(1.0) );
```

[7] «Τι να τον κάνουμε;» θα ρωτήσεις. Αν γράφεις πρόγραμμα C++ μάλλον δεν θα χρειαστείς τέτοιο πράγμα. Αν γράφεις C...

```
   PReal1 funcArr[3];
   double a[3], b[3];
   double riza;
   int   errCode;

   funcArr[0] = &q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = &cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = &em2;  a[2] = 0.0;  b[2] = 2.0;

   for ( int k(0); k < 3; ++k )
   {
      bisection( funcArr[k], a[k], b[k], 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
   }
```

Μην αμφιβάλλεις ότι μπορούμε να δώσουμε απλούστερα:

```
   funcArr[0] = q;  a[0] = 0.1;  b[0] = 1.0;
   funcArr[1] = cos;  a[1] = 0.0;  b[1] = pi;
   funcArr[2] = em2;  a[2] = 0.0;  b[2] = 2.0;
```

Ακόμη, μπορούμε να δηλώσουμε τον πίνακα χωρίς να ορίσουμε τον *PReal*1:

```
   double (*funcArr[3])( double );
```

Τέλος, ας δούμε και μια *συνάρτηση που επιστρέφει (βέλος προς) συνάρτηση*. Γράφουμε την:

```
PReal1 funcSel( PReal1 funcArr[], int n, int sel )
{
   return funcArr[sel];
}
```

Όπως βλέπεις αυτή επιστρέφει βέλος προς συνάρτηση και μπορούμε να γράψουμε (για λόγους επίδειξης και μόνον):

```
   bisection( funcSel(funcArr, 3, 1), a[1], b[1], 1e-5,
              riza, errCode );
```

Βεβαίως, θα μπορούσαμε να παραλείψουμε τους ορσμούς των ενδιάμεσων τύπων και να γράψουμε κατ' ευθείαν:

```
double (*(funcSel(double (*funcArr[])(double),
                  int n,int sel )))(double)
{
   return funcArr[sel];
}
```

Δεν σου αρέσει, ε;! Ας ελπίσουμε ότι δεν θα χρειαστεί να γράψεις τέτοιες συναρτήσεις.

Τελειώσαμε; Προς το παρόν: Ναι! Αλλά, υπάρχουν και χειρότερα που θα τα δεις αργότερα…[8]

14.5 Επιφόρτωση Συναρτήσεων

Στην §13.9.3 είδαμε τη συνάρτηση *swap()* που είναι πολύ χρήσιμη για όλους τους τύπους και όχι μόνον για τον `int`.

Να τη γράψουμε λοιπόν και για άλλους τύπους! Και για να μην αλλάζουμε πολύ το όνομα από τον έναν τύπο στον άλλο, θα κάνουμε το εξής: θα αλλάξουμε αυτήν που γράψαμε σε *swapI*, θα γράψουμε μια *swapD*, για μεταβλητές τύπου `double`, μια *swapC*, για μεταβλητές τύπου `char`… Ναι αυτά θα κάνουμε μόνο που δεν χρειάζεται να αλλάζουμε ονόματα.

Το πρόβλημα «*swap* για διάφορους τύπους» μπορεί να λυθεί έτσι:

```
#include <iostream>
```

[8] Έλα, αστείο ήταν!

```
using namespace std;

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

void swap( int& x, int& y );
void swap( double& x, double& y );
void swap( char& x, char& y );
void swap( WeekDay& x, WeekDay& y );

int main()
{
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

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

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

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

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

Εδώ, γράψαμε *τέσσερις διαφορετικές συναρτήσεις* –για τέσσερις τύπους (`int`, `double`, `char`, `WeekDay`)– *με το ίδιο όνομα*. Ο μεταγλωττιστής διαλέγει τη συνάρτηση που ταιριάζει στην κάθε κλήση ανάλογα με τους τύπους των ορισμάτων που βρίσκει σε αυτήν. Αυτό που κάναμε ονομάζεται **επιφόρτωση συναρτήσεων** (function overloading).

Το παράδειγμά μας δείχνει μια καλή χρήση επιφόρτωσης συναρτήσεων· *όλες οι συναρτήσεις που γράψαμε κάνουν την ίδια δουλειά* σε μεταβλητές διαφόρων τύπων.

Μπορείς όμως να γράφεις συναρτήσεις με το ίδιο όνομα που να κάνουν τελείως διαφορετικά πράγματα. Για παράδειγμα, μαζί με τις άλλες *swap()* να επιφορτώσεις και αυτήν:

```
double swap( int a, double b ) {  return (a+b)/2;  }
```

Ο μεταγλωττιστής δεν έχει αντίρρηση, αλλά κάτι τέτοιο δεν είναι και η πιο σοφή ιδέα για ένα πρόγραμμα.

Ας δούμε τώρα μερικά προβλήματα που μπορεί να συναντήσεις με τις επιφορτώσεις συναρτήσεων. Ας πούμε ότι ορίζεις:

```
void f( int& x, int& y ) { /*...*/ }
int f( int x, int y ) { /*...*/ return ...; }
```

```
void f( int& x, int y=0 ) { /*...*/ }
```

και μέσα στο πρόγραμμά σου δίνεις:

```
   int j1, j2;
// . . .
   f( 17, j2 );
   f( j1 );
   f( j1, j2 );
```

Βάλε τώρα τον εαυτό σου στη θέση του μεταγλωττιστή: Ποιαν από τις τρεις συναρτήσεις θα διάλεγες;

- Για την πρώτη περίπτωση τη δεύτερη συνάρτηση· είναι η μόνη που μπορεί να δεχθεί το "`17`" ως πρώτο όρισμα
- Για τη δεύτερη περίπτωση την τρίτη συνάρτηση· είναι η μόνη που μπορεί να κληθεί με ένα όρισμα.
- Για την τρίτη περίπτωση; Οποιαδήποτε! Και οι τρεις ταιριάζουν!

Αυτό το πρόβλημα μπορεί να σου παρουσιάζεται συχνά. Για να το λύνεις θα διαβάζεις προσεκτικά το διαγνωστικό που έβγαλε ο μεταγλωττιστής και θα σκέφτεσαι τα εξής:

- ***Ο μεταγλωττιστής επιλέγει –μεταξύ συναρτήσεων με το όνομα που δίνεται στην κλήση– την «πιο κοντινή» με βάση την υπογραφή (signature) της συνάρτησης, που για τη C++ είναι: το όνομα της συνάρτησης και οι τύποι των παραμέτρων.***[9]

Πιο συγεκριμένα:

- Πρώτη προτεραιότητα δίνεται σε συνάρτηση που οι τύποι των παραμέτρων της είναι ακριβώς ίδιοι με αυτούς των ορισμάτων της κλήσης. Έτσι, αν έχουμε τις συναρτήσεις:

```
void f( char x )                    // (a)
void f( int x )                     // (b)
void f( double x )                  // (c)
```

και δώσουμε:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (a).

- Δεύτερη προτεραιότητα δίνεται σε συναρτήσεις που οι τύποι των παραμέτρων τους είναι δυνατόν να προκύψουν από αυτούς των ορισμάτων της κλήσης με *ακέραιη προαγωγή*[10] ή κάποια από τις `float` σε `double` και `double` σε `long double`. Έτσι αν έχεις τις

```
void f( int x )                     // (b)
void f( double x )                  // (c)
```

για την:

```
   f( 'c' );
```

ο μεταγλωττιστής θα επιλέξει την (b).

- Τρίτη προτεραιότητα έχουν συναρτήσεις που οι τύποι των ορισμάτων τους προκύπτουν από αυτόν του ορίσματος με πάγιες μετατροπές τύπου όπως `int` σε `double`, `double` σε `int` και άλλες που θα μάθουμε αργότερα. Έτσι, αν δεν υπάρχουν οι επιλογές (a) και (b) ο μεταγλωττιστής θα δεχθεί τη (c).[11]

---

[9] Προσοχή! Ο επιστρεφόμενος τύπος δεν είναι μέρος της υπογραφής για τη C++· έτσι, στο παράδειγμά μας (τρίτη περίπτωση) ταιριάζει και η δεύτερη συνάρτηση.

[10] Δες το Παράρ. Ε.

[11] Υπάρχουν άλλα δύο σκαλοπάτια στην κλίματα προτεραιότητας που έχουν σχέση με πράγματα που θα μάθουμε αργότερα.

14.6 Επιφόρτωση Τελεστών

Οι τελεστές είναι συναρτήσεις. Ας πάρουμε τον "`+`". Μπορείς να τον δεις ως:

+: `int`×`int` → `int`

Ναι, αλλά μπορώ να γράψω και `1.2 + 3`. Φυσικά! Έχουμε επιφόρτωση των:

+: `double`×`int` → `double`

+: `int`×`double` → `double`

και βέβαια δεν τελειώσαμε εδώ. Ως άσκηση, σκέψου μερικές ακόμη συναρτήσεις «επιφορτωμένες» στον "`+`".

Η C++ μας επιτρέπει να επιφορτώνουμε όποιον (σχεδόν) θέλουμε από τους τελεστές της, όπως επιφορτώνουμε τις (άλλες) συναρτήσεις. Έχει όμως κάποιους περιορισμούς:

- *Δεν μπορούμε να πειράξουμε τους τελεστές που είναι ορισμένοι στους πρωτογενείς τύπους*. Π.χ. δεν μπορείς να επιφορτώσεις τον τελεστή "`*`" για πράξη μεταξύ ακεραίων. Μπορούμε να επιφορτώνουμε τελεστές για δικούς μας τύπους.
- *Δεν μπορούμε να αλλάξουμε το συντακτικό χρήσης των τελεστών*: αν ένας τελεστής είναι ενικός, όπως ο "`++`", θα πρέπει να επιφορτωθεί ως ενικός, αν είναι δυαδικός, όπως ο "`/`", θα πρέπει να επιφορτωθεί ως δυαδικός.
- *Δεν επιτρέπεται να επιφορτώσουμε τους τελεστές*: "`::`", "`.`", "`.*`", "`?:`".[12]

Και γιατί να επιφορτώσουμε έναν τελεστή, θα ρωτήσεις. Για να κάνουμε τη ζωή μας (προγραμματιστικώς) πιο εύκολη. Αν, για παράδειγμα, η *wd* είναι τύπου *WeekDay*, είναι προτιμότερο να προχωρούμε στην επόμενη μέρα εβδομάδας γράφοντας `++wd` αντί για `next(wd)` ή `advance(wd)`. Βεβαίως, να τονίσουμε ότι, αν επιφορτώσουμε για τη δουλειά αυτή τον "`--`", τότε... καλύτερα να μην επιφορτώσουμε! Να θυμάσαι πάντοτε έναν βασικό κανόνα:[13]

- ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***
  Από τεχνική άποψη:
- ***Η επιφόρτωση ενός τελεστή, ας πούμε του "`@`", γίνεται με την επιφόρτωση της συνάρτησης "`operator@()`".***

Προς το παρόν, ο μόνος τύπος που έχουμε ορίσει είναι ο *WeekDay* (§4.8). Στη συνέχεια λοιπόν θα δώσουμε παραδείγματα επιφόρτωσης των τελεστών "`++`" (ενικός) και "`<<`" (δυαδικός) για τον τύπο αυτόν και θα δεις τις τεχνικές λεπτομέρειες της επιφόρτωσης.

14.6.1 Τελεστής για Έξοδο Στοιχείων Τύπου *WeekDay*

Θέλουμε να επιφορτώσουμε τον "`<<`" έτσι ώστε αν η *wd*, τύπου *WeekDay*, έχει τιμή `sunday` και δώσουμε:

```
tout << wd << endl;
```

να γραφεί στο αρχείο text που έχει συνδεθεί με το *tout*:

```
Κυριακή
```

Αν, αντί για *tout*, γράψουμε στο *cout* θα πρέπει να το δούμε στην οθόνη.

Ένας δυαδικός τελεστής, σαν τον "`<<`", υλοποιείται (επιφορτώνεται) με συνάρτηση που έχει δύο παραμέτρους και όνομα "`operator<<`":

- Η πρώτη αντιστοιχεί στο όρισμα που εμφανίζεται αριστερά του "`<<`", στο παράδειγμά μας το "`tout`".

---

[12] Αργότερα θα τους μάθουμε όλους!

[13] Αυτή είναι και η σύσταση DCL11 του (CERT 2009): "Preserve operator semantics when overloading operators."

- Η δεύτερη αντιστοιχεί στο όρισμα που εμφανίζεται δεξιά του "`<<`", στο παράδειγμά μας το "`wd`".

  Αν θυμηθούμε τώρα ότι, όπως λέγαμε στην §13.9.2:

- «*Η* `cout << " x = "` *είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.» (Για την περίπτωσή μας να σκέφτεσαι "`tout`" αντί για "`cout`")
- «*Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* ***in out*** *(αναφοράς).*»
- «*Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο cout βάζε παράμετρο τύπου* `ostream` *και όχι* `ofstream`.»

καταλήγουμε στο συμπέρασμα ότι η επιφόρτωση θα πρέπει να γίνει με συνάρτηση που έχει επικεφαλίδα:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
```

και όχι "`void operator<<(…`" όπως λένε οι κανόνες μας.

Τα υπόλοιπα είναι απλά:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   string fv;
   switch ( rhs )
   {
      case sunday:    fv = "Κυριακή";  break;
      case monday:    fv = "Δευτέρα";  break;
      case tuesday:   fv = "Τρίτη";  break;
      case wednesday: fv = "Τετάρτη";  break;
      case thursday:  fv = "Πέμπτη";  break;
      case friday:    fv = "Παρασκευή";  break;
      case saturday:  fv = "Σάββατο";  break;
   } // switch
   return ( tout << fv );
} // operator<< WeekDay
```

Βέβαια, μπορούμε να το γράψουμε και έτσι:

```
ostream& operator<<( ostream& tout, WeekDay rhs )
{
   switch ( rhs )
   {
      case sunday:    return ( tout << "Κυριακή" );
      case monday:    return ( tout << "Δευτέρα" );
      case tuesday:   return ( tout << "Τρίτη" );
      case wednesday: return ( tout << "Τετάρτη" );
      case thursday:  return ( tout << "Πέμπτη" );
      case friday:    return ( tout << "Παρασκευή" );
      case saturday:  return ( tout << "Σάββατο" );
   } // switch
} // operator<< WeekDay
```

Αυτή η μορφή είναι πιο γρήγορη από την πρώτη αλλά παραβιάζει (άλλον) έναν από τους κανόνες μας: έχει πολλές `return`!

Όπως θα δούμε και στη συνέχεια, έτσι θα επιφορτώνουμε τους τελεστές εξόδου (αλλά και εισόδου) για όποιους τύπους μας ενδιαφέρει.

14.6.2 Ο Τελεστής "++" για τον Τύπο *WeekDay*

Όπως, πιθανότατα, μαντεύεις, αυτός ο τελεστής θα επιφορτωθεί με συνάρτηση που έχει όνομα "`operator++()`" και μια παράμετρο. Θέλουμε να τον ορίσουμε έτσι που *να μας δίνει την επόμενη ημέρα εβδομάδας*, όπως για ακέραιους μας δίνει τον επόμενο ακέραιο. Αν δεν δώσουμε δικό μας ορισμό, οι

```
   WeekDay d;
   . . .
```

```
   ++d;
```

δεν έχουν πρόβλημα αν η *d* έχει τιμή από *sunday* μέχρι *friday*: η τιμή της *d* προχωρεί στην επόμενη μέρα (ας πούμε: `d = d + 1`). Αν όμως η *d* έχει τιμή *saturday* τότε παίρνει τιμή έξω από τα όρια του τύπου ενώ θα έπρεπε να «ξαναγυρίζει» στη *sunday*.

Να πώς διορθώνουμε αυτό το πρόβλημα:[14]

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday ) lhs += 1;
                    else lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Ας δούμε τώρα μια δοκιμή (με οποιαδήποτε από τις δύο μορφές του `operator<<`):

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

enum WeekDay { sunday, def, tuesday, wednesday, thursday, friday,
               saturday };

ostream& operator<<( ostream& tout, WeekDay rhs );
WeekDay operator++( WeekDay& lhs );

int main()
{
// . . .
   WeekDay d( sunday );
   for ( int k(0); k < 10; ++k )
   {
      cout << d << endl;
      ++d;
   }
// . . .
} // main
```

Αποτέλεσμα:

```
Κυριακή
Δευτέρα
Τρίτη
Τετάρτη
Πέμπτη
Παρασκευή
Σάββατο
Κυριακή
Δευτέρα
Τρίτη
```

[14] Η Borland C++ v.5.5 θα δεχτεί αυτήν τη μορφή. Η gcc θα σου πει ότι δεν έχει ορισθεί ο "`+=`" για τον τύπο *WeekDay*. Στην περίπτωση αυτή γράψε την επιφόρτωση ως εξής:

```
WeekDay operator++( WeekDay& lhs )
{
   if ( lhs < saturday )
   {
      int iv( static_cast<int>(lhs) );
      ++iv;
      lhs = static_cast<WeekDay>(iv);
   }
   else
      lhs = sunday;
   return lhs;
} // operator++( WeekDay
```

Πού καλείται η `operator++()`; Στην εντολή "`++d`"! Δεν το πιστεύεις; Λοιπόν άλλαξε τη "`++d`" σε:

```
operator++( d );
```

Εδώ δεν έχεις πρόβλημα: Καλείται η συνάρτηση `operator++()` με όρισμα *d*. Ε, το πρόγραμμά σου θα περάσει από τον μεταγλωττιστή και εκτελούμενο θα δώσει ακριβώς τα ίδια αποτελέσματα που πήραμε παραπάνω! Μπορείς να θεωρήσεις ότι αυτή είναι η «κανονική» κλήση ενώ η "`++d`" είναι μια συντομογραφία με οικείο συντακτικό.

Και πού καλείται η `operator<<()`; Στην "`cout << d`". Για να το πιστέψεις, άλλαξε και εδώ τη "`cout << d << endl`" σε:

```
operator<<( cout, d ) << endl;
```

και θα δεις ότι –και πάλι– το πρόγραμμά θα δώσει τα ίδια αποτελέσματα! Και εδώ, μπορείς να θεωρήσεις ότι η η "`cout << d`" είναι συντομογραφία ενώ «κανονική» κλήση είναι η "`operator<<( cout, d )`".

Παρατήρηση: ▶

Η `operator++()` δεν είναι συμβατή με τους κανόνες μας: αλλάζει την τιμή της παραμέτρου και επιστρέφει τιμή. Εδώ όμως προτιμούμε να είμαστε συμβατοί με τη φιλοσοφία της C++ για τους τελεστές "`++`" και "`--`".

Στην επόμενη υποπαράγραφο θα μάθουμε ότι η «φιλοσοφία της C++» (για την ακρίβεια της C) απαιτεί τύπο αποτελέσματος της συνάρτησης `operator++()` όχι `WeekDay` αλλά `WeekDay&`.◀

Και αν θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" τι κάνουμε; Κάτι «παράξενο»:

```
WeekDay operator++( WeekDay& lhs, int )
{
   WeekDay sv( lhs );
   ++lhs;
   return sv;
} // operator++( WeekDay, int
```

Η (ανύπαρκτη) δεύτερη παράμετρος είναι *σήμα προς τον μεταγλωττιστή ότι εδώ μιλάμε για τον μεταθεματικό τελεστή* και όχι για τον προθεματικό.

Όσο για την υλοποίηση: χρησιμοποιούμε τον προθεματικό που έχουμε ήδη ορίσει.

14.6.3 Η Πράξη της Εκχώρησης (ξανά)

Έχουμε μάθει ότι με την `v =` *Π* γίνονται τα εξής:

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

Σε αυτό βασίζεται και η πολλαπλή εκχώρηση:

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

που εκτελείται ως:

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

Τι θα έλεγες τώρα αν έβλεπες:

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

Δεν σου κάνει νόημα, ε; Δες το παρακάτω πρόγραμμα:

```
#include <iostream>

using namespace std;

int main()
```

```
{
   int u(1), v(2), w(3);
   (w = v) = u;
   cout << u << "  " << v << "  " << w << endl;
}
```

Αποτέλεσμα:

```
1  2  1
```

Η πρώτη έκπληξη είναι ότι έγινε δεκτή, δηλαδή η "`(w = v)`" θεωρήθηκε ως τιμή-*l*. Η δεύτερη έκπληξη είναι ότι η *w* πήρε τιμή "`1`" (της *u*) και όχι "`2`" (της *v*).

Παρόμοια ισχύουν και για τις συντομογραφίες της εκχώρησης. Για παράδειγμα οι:

```
   cout << u << endl;
   (u += 7) = 11;
   cout << u << endl;
   (++u) = 17;
   cout << u << endl;
```

δίνουν:

```
1
11
17
```

Πώς εξηγούνται τα παραπάνω; Με την υποσημείωση της §11.3 «*Πρόσεξε: «η* `v`*» και όχι «η τιμή της* `v`*». Θα καταλάβεις αργότερα...*» Το «αργότερα» είναι τώρα.

- Ας πάρουμε την "`(w = v) = u`". Η τιμή της *v* αποθηκεύεται στην *w*, αλλά το αποτέλεσμα της πράξης "`w = v`" είναι *η μεταβλητή w*, δηλαδή μια τιμή-*l*. Έτσι, στη συνέχεια εκτελείται η πράξη "`w = u`" και η *w* παίρνει την τιμή της *u* που είναι "`1`".
- Ας πάρουμε την "`(++u) = 17`" που είναι ισοδύναμη με "`(u=u+1) = 17`". Η "`u=u+1`" θα επιστρέψει ως αποτέλεσμα τη μεταβλητή *u* και στη συνέχεια θα εκτελεσθεί η "`u = 17`".

Για να είμαστε σύμφωνοι με αυτά, όταν επιφορτώνουμε οποιονδήποτε τελεστή εκχώρησης θα πρέπει να βγάζουμε ως αποτέλεσμα της πράξης τη «μεταβλητή του αριστερού μέρους». Θα βάλουμε λοιπόν ως τύπο αποτελέσματος όχι τον τύπο *T* της μεταβλητής αλλα τον *T*`&`.

- ***Για να μην αποκλίνουμε από τη «φιλοσοφία της C++» (της C) θα πρέπει και εμείς να επιφορτώνουμε τους τελεστές εκχώρησης ως:***

  ```
  T& operator=( T& lhs, const T& rhs )
  T& operator+=( T& lhs, const T& rhs )
  T& operator++( T& lhs )
  ```

  (*όχι* `const T&`).[15] ***Τα ίδια ισχύουν και για τους "--", "-=", "*=" κλπ.***

  Επομένως, θα ξαναγράψουμε την επιφόρτωση του

```
WeekDay& operator++( WeekDay& v )
{
   if ( v < saturday ) v += 1;
                  else v = sunday;
   return v;
} // operator++( WeekDay
```

Για τους μεταθεματικούς "`++`", "`--`" ισχύουν αυτά που είπαμε στην προηγούμενη παράγραφο. Στην περίπτωση αυτή δεν μπορούμε να επιστρέψουμε τύπο αναφοράς, αφού επιστρέφεται μια μεταβλητή (*sv*) τοπική στη συνάρτηση.

14.6.4 Γενικώς ...

Γενικώς, μπορούμε να πούμε ότι

[15] Δεν θα παραλείψουμε όμως να τονίσουμε ότι η σύσταση 34 της (ELLEMTEL 1992) λέει: "An assignment operator ought to return a `const` reference to the assigning object."

- Επιφορτώνουμε έναν *δυαδικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( Tl lhs, Tr rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *Tl* ο τύπος της πρώτης παραμέτρου και *Tr* ο τύπος της δεύτερης παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, y )...`" είτε ως "`...x @ y...`". Στην πρώτη παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*) και στη δεύτερη (*rhs*) αυτό που εμφανίζεται μετά από αυτόν (*y*).
- Επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή `@` με μια συνάρτηση
  `Trv operator@( T rhs )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x )...`" είτε ως "`...@x...`". Στη μοναδική παράμετρο (*rhs*) θα πάει το όρισμα που εμφανίζεται μετά από τον τελεστή (*x*).
- Επιφορτώνουμε έναν *μεταθεματικό ενικό* τελεστή `@` (όταν υπάρχει και *προθεματικός ενικός* τελεστής `@`) με μια συνάρτηση
  `Trv operator@( T lhs, int )`
  όπου *Trv* είναι ο τύπος του αποτελέσματος, *T* ο τύπος της παραμέτρου. Μπορούμε να τον χρησιμοποιούμε είτε ως "`...operator@( x, 1 )...`" είτε ως "`...x@...`". Στην πρώτη (και στην πραγματικότητα) μοναδική παράμετρο (*lhs*) θα πάει το όρισμα που εμφανίζεται πριν από τον τελεστή (*x*).

Αργότερα θα δούμε ότι για κλάσεις μερικά από αυτά τα πράγματα θα αλλάξουν. Αλλά –να επαναλάβουμε με άλλα λόγια– σε κάθε περίπτωση, το ουσιώδες είναι το εξής:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

14.7 Γενικές Συναρτήσεις

Στην §14.5 γράψαμε και επιφορτώσαμε τέσσερις φορές τη *swap*. Αλλά, η μόνη διαφορά στις τέσσερις συναρτήσεις είναι ο τύπος των παραμέτρων. Ο μεταγλωττιστής της C++ μας δίνει την εξής δυνατότητα: να του δώσουμε το σχέδιο –το «πατρόν– και να γράψει εκείνος όποιες συναρτήσεις θα χρειαστούν. Αυτό το σχέδιο λέγεται **περίγραμμα συνάρτησης** (function template).

Για το περίγραμμα θα δεις και τον όρο **γενική συνάρτηση** (generic function). Πάντως, όπως θα καταλάβεις στη συνέχεια *μια γενική συνάρτηση δεν είναι συνάρτηση αφου δεν έχει συγκεκριμένο πεδίο ορισμού ή/και πεδίο τιμών*.

14.7.1 Περιγράμματα Συναρτήσεων

Ξαναγράφουμε το παράδειγμα της §14.5 ως εξής:

```
#include <iostream>

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

enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday };

template < typename T >
void swap( T& x, T& y );

int main()
{
```

```
   int j1( 10 ), j2( 20 );
   double d1( 1.23 ), d2( 2.34 );
   char c1( 'A' ), c2( 'B' );
   WeekDay m1( sunday ), m2( tuesday );

   swap( j1, j2 );   cout << j1 << "  " << j2 << endl;
   swap( d1, d2 );   cout << d1 << "  " << d2 << endl;
   swap( c1, c2 );   cout << c1 << "  " << c2 << endl;
   swap( m1, m2 );   cout << int(m1) << "  " << int(m2) << endl;
} // main

template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Η βασική διαφορά[16] είναι: αντί για τις τέσσερις συναρτήσεις έχουμε το:

```
template < typename T >
void swap( T& x, T& y )
{
   T s( x );
   x = y;  y = s;
} // swap
```

Αυτό είναι ένα **περίγραμμα** (template) της συνάρτησης *swap* με παράμετρο τον τύπο *T*. Αντί για "`typename T`" θα μπορούσαμε να γράψουμε και "`class T`". Τα "`class`", "`template`" και "`typename`" είναι λέξεις-κλειδιά της C++.

Το πρόγραμμα αυτό δίνει:

```
20  10
2.34  1.23
B  A
2  0
```

Όταν ο μεταγλωττιστής βρει τη "`swap(j1, j2)`" δημιουργεί ένα **στιγμιότυπο** (instance) του περιγράμματος σε μια συνάρτηση σαν την

```
void swap( int& x, int& y )
```

Το στιγμιότυπο δημιουργείται *αυτομάτως* αφού δεν υπάρχει αμφιβολία για το ότι θα πρέπει να βάλει όπου *T* τον `int`.

Παρομοίως, όταν βρεί τη "`swap(d1, d2)`" ο μεταγλωττιστής θα δημιουργήσει ένα στιγμιότυπο:

```
void swap( double& x, double& y )
```

κ.ο.κ.

Ένα περίγραμμα μπορεί να έχει ως παραμέτρους

- έναν ή περισσότερους *τύπους* ή/και
- *ακέραιους*,
- *βέλη* ή *αναφορές*.

και μπορεί να μας δώσει μια συνάρτηση –που τη λέμε **στιγμιότυπο του περιγράμματος** (template instance)– με δύο τρόπους:

- Όπως στα παραπάνω παραδείγματα, όπου οι τύποι που θα αντικαταστήσουν τους τύπους-παραμέτρους συνάγονται αυτομάτως από τους τύπους των ορισμάτων της κάθε κλήσης. Αυτή είναι η **συναγόμενη δημιουργία στιγμιοτύπου** (implicit instantiation).
- Με τη **ρητή** (explicit) δημιουργία στιγμιοτύπου κατά την οποία γράφουμε, στην κλήση, τις τιμές των παραμέτρων μετά το όνομα του περιγράμματος. Π.χ. στο παράδειγμά

[16] Υπάρχει και η «μικροδιαφορά» στο "`using`". Θα τη συζητήσουμε παρακάτω.

μας, αντί για `swap(d1, d2)`, θα μπορούσαμε να γράψουμε: `swap<double>(d1, d2)`. Ας δούμε ένα

Παράδειγμα 1

Έστω ότι θέλουμε να μετατρέψουμε σε περίγραμμα τη *vectorSum* (§12.1) Μια απλή προσπάθεια μας οδηγεί στο εξής:

```
template < typename CT >
CT vectorSum( const CT x[], int n, int from, int upto )
{
   CT sum( 0 );

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

Εδώ θα πρέπει να σκεφτούμε την εξής περίπτωση: Έστω ότι βάζουμε ως *CT* τον `double`. Παρ' όλο όμως που τα στοιχεία του πίνακα είναι `double` μπορεί το άθροισμα τους να είναι μεγαλύτερο από το *DBL_MAX*. Σε μια τέτοια περίπτωση θα μπορούσαμε να πάρουμε το σωστό αποτέλεσμα σε `long double`. Την αλλάζουμε λοιπόν:

```
template < typename CT, typename RT >
RT vectorSum( const CT x[], int n, int from, int upto )
{
   RT sum( 0 );

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

Τώρα όμως έχουμε άλλο πρόβλημα: Έστω ότι έχουμε δηλώσει

```
   double ar[7971];
   long double res;
```

Αν δοκιμάσουμε να δώσουμε

```
   res = vectorSum( ar, 7971, 0, 7970 );
```

ο μεταγλωττιστής δεν θα μπορέσει να βγάλει άκρη. Το σωστό είναι να γράψουμε τις οδηγίες για τη δημιουργία στιγμιοτύπου:

```
   res = vectorSum<double, long double>( ar, 7971, 0, 7970 );
```

Όταν η παράμετρος είναι *ακέραιος*

- Αυτή μπορεί να χρησιμοποιηθεί στη συνάρτηση ως σταθερά.
- Το αντίστοιχο όρισμα θα πρέπει να είναι σταθερά.

Παράδειγμα 2

Στο Παράδ. 6 της §13.9.3 συζητήσαμε για δυο συναρτήσεις

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
void output2DAr( ostream& tout,
                 const int* a, int nRow, int nCol )
```

για να μη γράφουμε ξανά και ξανά τις ίδιες εντολές στο πρόγραμμα πολλαπλασιασμού πινάκων (Παράδ. 4, §12.4). Εκεί γράψαμε την πρώτη από αυτές και εσύ, πού έλυσες την Άσκ. 13-1, έχεις γράψει τη δεύτερη.

Τώρα όμως μας κατεβαίνει μια ιδέα: Αφού μπορούμε να βάλουμε στο περίγραμμα ακέραιη παράμετρο που έρχεται στη συνάρτηση ως σταθερά, να βάλουμε εκεί τον αριθμό στηλών του πίνακα και να δηλώσουμε τον πίνακα ως δισδιάστατο:

```
template< typename T, unsigned int nCol >
void input2DAr( istream& tin, T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
```

```
} // input2DAr
```

Καλό; Εξαιρετικό! Να κάνουμε έτσι και την άλλη:

```
template < typename T, unsigned int nCol >
void output2DAr( ostream& tout,
                 const T a[][nCol], int nRow )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Τώρα, στο πρόγραμμά μας διαβάζουμε με τις:

```
   input2DAr< int, m >( atx, a, l );
   input2DAr< int, n >( atx, b, m );
```

γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr< int, m >( cout, a, l );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr< int, n >( cout, b, m );
   cout << " Στοιχεία του πίνακα c" << endl;
   output2DAr< int, n >( cout, d, l );
```

και όλα είναι μια χαρά![17] Πρόσεξε ότι εδώ *είναι απαραίτητη η ρητή δημιουργία στιγμιοτύπων* αφού ο αριθμός στηλών του πίνακα δεν υπάρχει στην κλήση ώστε να μπορεί να συναχθεί η τιμή της δεύτερης παραμέτρου.

Αλλά τίποτε δεν δίνεται δωρεάν στον κόσμο αυτόν! Αν χρησιμοποιήσουμε τις συναρτησεις με τον «γραμμοποιημένο» πίνακα το μέγεθος του εκτελέσιμου είναι 156160 ψηφιολέξεις ενώ με τα περιγράμματα έχουμε 156672.[18]

Γιατί; Διότι στην πρώτη λύση έχουμε δύο συναρτήσεις ενώ στη δεύτερη έχουμε πέντε: δύο *input2DAr* και τρεις *output2DAr*! Γράφηκαν αυτομάτως βέβαια, αλλά γράφηκαν.

Και μετά από αυτό, να απαντήσουμε και το ερώτημα που βάλαμε στην υποσημείωση: Αν θέλουμε παράμετρο για το εύρος πεδίου θα τη βάλουμε στη συνάρτηση. *Αν τη βάλεις ως παράμετρο του περιγράμματος για κάθε τιμή της θα έχουμε διαφορετικό στιγμιότυπο*· αυτό είναι υπερβολή.

Ας πούμε και κάτι ακόμη που –προς το παρόν– μπορεί να μην σου κάνει πολύ νόημα:

- Μπορείς να έχεις στιγμιότυπα της *swap* για οποιονδήποτε τύπο
  - έχει τον τελεστή εκχώρησης και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *vectorSum()* που έχουν για δεύτερο όρισμα (*RT*) οποιονδήποτε τύπο
  - έχει τον τελεστή "`+=`" ώς `operator+=( RT, CT )` και
  - επιτρέπει δήλωση με αρχική τιμή
- Μπορείς να έχεις στιγμιότυπα της *output2DAr* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`<<`" για έξοδο στοιχείων προς αρχείο text. Π.χ. μπορείς να έχεις στιγμιότυπο για τον *WeekDay*.

---

[17] Πάντως εκείνο το "`3`" στη "`tout.width(3)`" είναι μαγική σταθερά. Να το βάλουμε ως παράμετρο στη συνάρτηση ή μήπως στο περίγραμμα;

[18] Borland C++, v.5.5.

- Μπορείς να έχεις στιγμιότυπα της *input2Dar()* για οποιονδήποτε τύπο υπάρχει ορισμένος ο "`>>`" για είσοδο στοιχείων από αρχείο text. Π.χ. *δεν* μπορείς να δημιουργήσεις στιγμιότυπο για τον *WeekDay*.

Ας πούμε τώρα ότι θέλουμε να μετατρέψουμε σε περίγραμμα και τη *max*. Αυτό είναι εύκολο:

```
template< typename T >
T max( T x, T y )  {  return (x > y) ? x : y;  }
```

Από αυτό το περίγραμμα μπορούμε να πάρουμε στιγμιότυπα για οποιονδήποτε τύπο ορίζεται ο τελεστής "`>`", όπως είναι οι `int`, `double`, `char` κ.ο.κ. Και αν θέλουμε να συγκρίνουμε δύο ορμαθούς C (πίνακες χαρακτήρων); Τώρα *T* είναι ο `char*` και προφανώς δεν μπορούμε να πάρουμε στιγμιότυπο. Η C++ όμως μας δίνει τη δυνατότητα να κάνουμε μια **εξειδίκευση** (specialization) του περιγράμματος για τον `char*`:

```
template<>
char* max<char*>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

- Η δήλωση μιας εξειδίκευσης περιγράμματος αρχίζει με "`template<>`".
- Οι τιμές των παραμέτρων για την εξειδίκευση γράφονται μέσα σε "`<`", "`>`" μετά το όνομα του περιγράμματος.

Αν οι τιμές των παραμέτρων της εξειδίκευσης συνάγονται από τους τύπους των παραμετρων της συνάρτησης μπορείς να τις παραλείψεις. Για τη *max* μπορούμε να γράψουμε:

```
template<>
char* max<>( char* x, char* y )
{  return (strcmp(x, y) > 0) ? x : y;  }
```

Ακόμη, μπορείς να παραλείψεις τις παραμέτρους εξειδίκευσης αν α) υπάρχουν προκαθορισμένες τιμές για τις παραμέτρους του περίγράμματος και β) η εξειδίκευση γίνεται για τις προκαθορισμένες τιμές. Με τις προκαθορισμένες τιμές που βάλαμε στις παραμέτρους του *output2DAr()* γράφοντας:

```
template <>
void output2DAr<>( ostream& tout,
                   const int a[][nCol], int nRow )
// . . .
```

δηλώνουμε εξειδίκευση για τύπο `int` και *nCol* `==` 2.

14.7.1.1 Η «Μικροδιαφορά» στο "`using`"

Όταν αλλάξαμε το πρόγραμμα για να δοκιμάσουμε το περίγραμμα της *swap* βγάλαμε το γενικό

```
using namespace std;
```

και βάλαμε συγκεκριμένα:

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

Γιατί; Διότι στο *sdt* υπάρχει ήδη –μεταξύ άλλων– περίγραμμα *swap()* που μας έρχεται έτοιμο από τη βιβλιοθήκη της C++! Στα «άλλα» περιλαμβάνονται περιγράμματα για τις *min()*, *max()* και άλλες συνηθισμένες συναρτήσεις. Δοκίμασέ τα! Αν δεν φτάνει το "`using namespace std`" βάλε και "`#include <algorithm>`".

14.7.2 Επιφόρτωση στο Περίγραμμα

Ας γυρίσουμε στο περίγραμμα της *swap()* για να σκεφτούμε την εξής περίπτωση: μπορούμε να το χρησιμοποιήσουμε για να αντιμεταθέσουμε τις τιμές των:

```
   char s1[15], s2[20];
```

Όχι, φυσικά! Θα πρέπει να γράψουμε μια:

```
   void swap( char x[], char y[] )
```

και να την επιφορτώσουμε στο περίγραμμα που προϋπάρχει. Ας πούμε ότι τη γράφουμε:

```
void swap( char x[], char y[] )
{
   string s( x );
   strcpy( x, y );
   strcpy( y, s.c_str() );
} // swap( char*
```

Παρατηρήσεις: ▶

1. Γιατί «ας πούμε...»; Διότι για να δουλέψει θα πρέπει η τιμή του x να «χωράει» στον y και το αντίστροφο. Αυτό το αφήνουμε στον προγραμματιστή-χρήστη της συνάρτησης.
2. Για να τη δοκιμάσεις θα πρέπει –εκτός από τις "`using std::cout`" και "`using std::endl`"– να βάλεις και μια "`using std::string`".◀

Όταν ο μεταγλωττιστής βρει την εντολή:

```
   swap( s1, s2 );
```

θα κάνει τη σωστή επιλογή.

Αν θέλεις κάνε και το εξής πείραμα: όρισε

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }
```

δήλωσε:

```
   int k( 23 );
   double e( 7.33 );
```

και ζητήσε:

```
   qsc( k, 'a' );
   qsc( e, 'c' );
```

Η πρώτη κλήση θα εξυπηρετηθεί από εξειδίκευση του περιγράμματος ενώ η δεύτερη από την απλή συνάρτηση και θα πάρεις:

```
in template
in function
```

Μπορείς να κάνεις και επιφόρτωση περιγραμμάτων: Για παράδειγμα, μπορείς πέρα από τους παραπάνω ορισμούς της *qsc* να δώσεις και

```
template< typename T >
void qsc( int x, T c )
{  cout << "in template 2" << endl;  }
```

Δεν υπάρχει οποιοδήποτε πρόβλημα εκτός από την περίπτωση που θα βάλεις στο πρόγραμμά σου:

```
   qsc( k, 'a' );
```

Στην περίπτωση αυτήν ο μεταγλωττιστής δεν μπορεί να καταλάβει ποιο περίγραμμα εννοείς.

14.7.2.1 Επιφόρτωση, Εξειδίκευση και Άλλα Ψιλά Γράμματα...

Όταν ο μεταγλωττιστής βρει κλήση σε συνάρτηση που μπορεί να προκύψει από περίγραμμα που είναι επιφορτωμένο με άλλο (-α) περίγραμμα (-τα) ή/και απλή (-ές) συνάρτηση (-σεις) επιλέγει την κατάλληλη συνάρτηση ως εξής:

- Από όλα τα περιγράμματα με το όνομα συνάρτησης της κλήσης γίνεται προσπάθεια δημιουργίας στιγμιοτύπων που να ταιριάζουν με την κλήση.

- Από όλα τα στιγμιότυπα και τις απλές συναρτήσεις που τυχόν ταιριάζουν επιλέγεται η «πιο κοντινή» προς την κλήση με βάση αυτά που είπαμε στο τέλος της §14.5.
- Αν υπάρχει «ισοπαλία» στιγμιοτύπου περιγράμματος και απλής συνάρτησης προτεραιότητα έχει η απλή συνάρτηση.
- Αν η επιλογή είναι στιγμιότυπο κάποιου περιγράμματος και υπάρχει εξειδίκευση με την ίδια υπογραφή χρησιμοποιείται η εξειδίκευση.

Ας πούμε ότι έχουμε:

```
template< typename T >
void qsc( T x, char c )
{  cout << "in template" << endl;  }

void qsc( double x, char c )
{  cout << "in function" << endl;  }

template<> void qsc<double>( double x, char c )
{  cout << "in template spec" << endl;  }
```

Δηλαδή: γράφουμε μια εξειδίκευση του περιγράμματος ακριβώς για την περίπτωση που έχουμε την επιφόρτωση! Τα καταφέραμε να μπερδέψουμε τον μεταγλωττιστή με την κλήση:

```
   qsc( e, 'c' );
```

Όχι! Το πρόγραμμα θα μεταγλωττισθεί και θα δώσει:

```
in function
```

Ας δούμε γιατί γίνεται αυτό: Σύμφωνα με αυτά που είπαμε παραπάνω έχουμε να επιλέξουμε μεταξύ του στιγμιοτύπου:

```
void qsc<double>( double x, char c );
```

και της απλής συνάρτησης

```
void qsc( double x, char c );
```

Αφού στην κλήση "`qsc(e,'c')`" το πρώτο όρισμα είναι `double` και το δεύτερο `char` και τα δύο έχουν πρώτη προτεραιότητα. Αφού έχουμε «ισοπαλία» επιλέγεται η απλή συνάρτηση.

Όπως βλέπεις, στο παράδειγμά μας, ο μεταγλωττιστής δεν πρόκειται να φτάσει να εξετάσει την «υποψηφιότητα» της εξειδίκευσης του περιγράμματος για την κλήση "`qsc(e,'c')`".

Αν είχαμε επιλογή του στιγμιοτύπου του περιγράμματος τότε θα είχαμε χρήση της εξειδίκευσης.

14.8 Η Στοίβα

Στην επόμενη παράγραφο θα προσπαθήσουμε να δούμε πώς δουλεύει *ο μηχανισμός των εξαιρέσεων* της C++. Στη συνέχεια θα (ξανα)μιλήσουμε για *αναδρομή*. Και στις δύο περιπτώσεις θα αναφερθούμε στη στοίβα. Καλό είναι λοιπόν να πούμε από πριν δυο λόγια για αυτήν την πολύτιμη περιοχή μνήμης.

Μέχρι στιγμής έχουμε γνωρίσει δύο είδη[19] –από την άποψη της διαχείρισης– μνήμης:

- τη **στατική** (static), για τα στατικά αντικείμενα· οτιδήποτε βάλουμε εκεί δημιουργείται μια φορά και παραμένει στην ίδια διεύθυνση μέχρι το τέλος της εκτέλεσης του προγράμματος,
- την **αυτόματη** (automatic) ή μνήμη **στοίβας** (stack) για τα ορίσματα και τα τοπικά αντικείμενα συναρτήσεων και ομάδων· ό,τι βάλουμε εκεί ζει όσο ζει η αντίστοιχη

[19] Αργότερα θα μάθουμε και ένα τρίτο είδος: τη **δυναμική** μνήμη ή μνήμη **σωρού**.

συνάρτηση: δημιουργείται με την κλήση της και καταστρέφεται με το τέλος της εκτέλεσής της.

Το παρακάτω πρόγραμμα σου επιτρέπει να διακρίνεις αυτές τις δύο περιοχές:

```
#include <iostream>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long fl( double p )
{
   long s;

   cout << " &p = " << &p << "    &s = " << &s << endl;
   p = fd(p, 1.56);
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   y = fl(5);
   x[2] = fd( x[0], y );
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B178
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 &p = 0012FF44    &s = 0012FF38
 &p1 = 0012FF28    &p2 = 0012FF30
 &p1 = 0012FF3C    &p2 = 0012FF44
```

Εδώ, αφού πάρεις υπόψη σου ότι έχουμε δεκαεξαδικό σύστημα, παρατήρησε τα εξής:

- Η αποθήκευση της καθολικής μεταβλητής *g* έγινε σε άλλη περιοχή της μνήμης (0041B178) από αυτήν που αποθηκεύονται οι υπόλοιπες (0012FF...). Η *g* είναι στη στατική μνήμη ενώ οι υπόλοιπες είναι στη μνήμη στοίβας.
- Όπως αναμένεται, η διεύθυνση της παραμέτρου αναφοράς *a*3, στην *q*, είναι ίδια με αυτήν της *b* στη **main**.
- Η εκτέλεση του προγράμματος αρχίζει από τη **main**. Έτσι, οι πρώτες μεταβλητές που υλοποιούνται στη στοίβα είναι οι τοπικές της **main** στις διευθύνσεις από 0012FF54 (x) μέχρι 0012FF88 (*a*).
- Με την κλήση της *fl* υλοποιούνται και οι δικές της τοπικές μεταβλητές στη στοίβα από 0012FF38 (*s*) μέχρι 0012FF44 (*p*).
- Με την κλήση της *fd*, μέσα από την *fl*, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF28 (*p*1) μέχρι 0012FF30 (*p*2).
- Μετά την ολοκλήρωση της κλήσης της *fl*, όλη η μνήμη που χρησιμοποιήθηκε από την *fl* και την *fd* επιστρέφεται στη στοίβα. Αυτό φαίνεται από το επόμενο βήμα:

- Με την κλήση της *fd*, από τη **main**, υλοποιούνται οι τοπικές μεταβλητές της στη στοίβα από 0012FF3C (*p*1) μέχρι 0012FF44 (*p*2). Η μνήμη αυτή είχε χρησιμοποιηθεί προηγουμενως για την υλοποίηση των μεταβλητών της *fl*.

Αυτή η λειτουργία της αυτόματης μνήμης, ό,τι εισάγεται τελευταίο να εξάγεται πρώτο (Last In First Out), είναι χαρακτηριστικό της, συχνότατα χρησιμοποιούμενης, δομής στοιχείων που ονομάζεται **στοίβα** (stack, LIFO stack).

14.8.1 Η Συνάρτηση *stackavail*

Αν δουλεύεις με τον μεταγλωττιστή Borland C++, v.5.5 (ή v.5.02) μπορείς να δεις πιο καλά τη χρήση της στοίβας χρησιμοποιώντας τη συνάρτηση (εκτός προτύπου) *stackavail* που δίνει τη διαθέσιμη μνήμη στοίβας:

```
#include <iostream>
#include <malloc.h>

using namespace std;

long g = 0;

double fd( double p1, double p2 )
{
   cout << " μέσα στην fd, stackavail: " << stackavail() << endl;
   cout << " &p1 = " << &p1 << "    &p2 = " << &p2 << endl;
   return p1*p2/2;
} // fd

long int fl( double p )
{
   long s;

   cout << " μέσα στην fl, stackavail: " << stackavail() << endl;
   cout << " &p = " << &p << "    &s = " << &s << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   p = fd(p, 1.56);
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
   s = (p + 1.56)/2;
   return s;
} // fl

int main()
{
   long int a, b;
   double x[5], y;

   cout << " &g = " << &g << endl;
   cout << " &a = " << &a << "    &b = " << &b << endl
        << " x = " << x << "    &y = " << &y << endl;
   cout << " πριν κληθεί η fl, stackavail: " << stackavail()
        << endl;
   y = fl(5);
   cout << " μετά την fl, stackavail: " << stackavail() << endl;
   cout << " πριν κληθεί η fd, stackavail: " << stackavail()
        << endl;
   x[2] = fd( x[0], y );
   cout << " μετά την fd, stackavail: " << stackavail() << endl;
}
```

Αποτέλεσμα (Borland C++, v.5.5 σε Windows XP):

```
 &g = 0041B17C
 &a = 0012FF88    &b = 0012FF84
 x = 0012FF54    &y = 0012FF7C
 πριν κληθεί η fl, stackavail: 1048376
```

```
μέσα στην fl, stackavail: 1048356
&p = 0012FF40    &s = 0012FF34
πριν κληθεί η fd, stackavail: 1048356
μέσα στην fd, stackavail: 1048332
&p1 = 0012FF24    &p2 = 0012FF2C
μετά την fd, stackavail: 1048356
μετά την fl, stackavail: 1048376
πριν κληθεί η fd, stackavail: 1048376
μέσα στην fd, stackavail: 1048352
&p1 = 0012FF38    &p2 = 0012FF40
μετά την fd, stackavail: 1048376
```

Πρόσεξε τα εξής:

- Πριν κληθεί η *fl* έχουμε μνήμη στοίβας 1048376 ψηφιολέξεις. Μόλις άρχισε η εκτέλεσή της η διαθέσιμη μνήμη γίνεται 1048356 ψηφιολέξεις. Οι 20 ψηφιολέξεις διατίθενται για την υλοποίηση των μεταβλητών (8+4) και για τη διαχείριση της κλήσης. Η διαθέσιμη μνήμη στοίβας ξαναπαίρνει την αρχική της τιμή «`μετά την fl, stackavail: 1048376`».
- Η κάθε κλήση της *fd* κοστίζει 24 ψηφιολέξεις αλλά σε διαφορετική περιοχή της στοίβας κάθε φορά.

14.9 Διαχείριση Εξαιρέσεων με Δυο Λόγια

Ας ξεκινήσουμε με το παράδειγμα που είδαμε στην §7.8. Γράψαμε τη συνάρτηση *V*, που δεν ορίζονταν στα σημεία $\{ k: \mathbb{Z} \bullet 2k \}$:

```
double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      cerr << " η v κλήθηκε με όρισμα: " << x << endl;
      exit( EXIT_FAILURE );
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Εδώ εφαρμόσαμε το βήμα 7 της συνταγής του Πλ. 7.4 που λέει:
«*Αν η συνάρτηση είναι μερική, γράψε τις εντολές που εξαιρούν τις τιμές του ορίσματος για τις οποίες δεν ορίζεται η συνάρτηση:*

```
if (x δεν ανήκει στο πεδίο ορισμού)
{
   cerr << " η ... κλήθηκε με x = " << x << endl;
   exit( EXIT_FAILURE );
}
else
   Υπολόγισε την τιμή της συνάρτησης»
```

Φυσικά η συνταγή αυτή εφαρμόζεται και στις συναρτήσεις χωρίς τύπο. Αν ας πούμε, έχουμε το πρόβλημα:

> *Γράψε συνάρτηση, με το όνομα pqer, που θα τροφοδοτείται μέσω των ορισμάτων της με τρεις πραγματικές τιμές, ας πούμε x, y, z και θα υπολογίζει και θα επιστρέφει τις τιμές των παραστάσεων:*

$$t = \frac{xy}{x^2 - y^2} z^{x-y}, \quad u = \frac{xy - \dfrac{1}{x}}{z}, \quad \text{αν } |x| \neq |y| \text{ και } z > 0.$$

θα γράψουμε:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
   {
      cerr << " η pqer κλήθηκε με ορίσματα " << x << ", "
           << y << ", " << z << endl;
      exit( EXIT_FAILURE );
   }
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Αυτός ο τρόπος διαχείρισης τέτοιων καταστάσεων –ορίσματα εκτός πεδίου ορισμού– έχει δυσάρεστο αποτέλεσμα. Π.χ. οι παρακάτω εντολές προσπαθούν να δώσουν πίνακα τιμών της *v* από -5 ως 5 ανά 0.5:

```
   for ( int k(-10); k <= 10; ++k )
      cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v κλήθηκε με όρισμα: -4
```

Θα δούμε τώρα έναν άλλον τρόπο, σαφώς πιο ευέλικτο:

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

using namespace std;

double v( double x );

int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " η v δεν ορίζεται στο " << x << endl;
      }
   } // for (int k...
} // main

double v( double x )
{
   const double a( -1 ), b( 1 ), T( b-a );
   double m( floor( (x-a)/T ) ), x0( x - m*T ), fv;

// (-1 <= x0 < 1) && (v(x0) == v(x))
   if ( x0 == 0 )
   {
      throw x;
   }
// (-1 <= x0 < 1) && (x0 != 0) && (v(x0) == v(x))
   fv = -1/fabs(x0+2) - 1/fabs(x0) - 1/fabs(x0-2);
   return fv;
} // v
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
```

```
η v δεν ορίζεται στο -4
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
η v δεν ορίζεται στο -2
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
η v δεν ορίζεται στο 4
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, έχουμε πολύ πιο «ήπια» αντιμετώπιση του σφάλματος. Παίρνουμε τουλάχιστον τις τιμές στις οποίες υπολογίζεται η συνάρτησή μας.

Ποιες διαφορές έχουμε τώρα; Οι «μαγικές» λέξεις είναι: `throw`, `try` και `catch`.

- Όταν καταλαβαίνουμε ότι έχουμε τιμή εκτός πεδίου ορισμού, με τη "`throw x`" **ρίχνουμε** (ή **εγείρουμε**) **μιαν εξαίρεση** (throw (raise) an exception). Η *x* είναι το **αντικείμενο της εξαίρεσης** (exception object). Ύστερα από αυτό διακόπτεται η εκτέλεση της συνάρτησης *v()* και ο έλεγχος επιστρέφει στη συνάρτηση που την κάλεσε, στην περίπτωσή μας στη **main**.
- Στη **main**, η κλήση της *v* γίνεται μέσα σε μια εντολή `try` (δοκίμασε). Στην ομάδα της εντολής `try` βάζουμε τις εντολές –κλήσεις συναρτήσεων– από τις οποίες περιμένουμε ότι μπορεί να ριχθεί κάποια εξαίρεση.
- Η εξαίρεση **συλλαμβάνεται** από κάποια `catch` (σύλλαβε) που ακολουθεί την `try`. Μια `catch` συλλαμβάνει εξαιρέσεις του τύπου που αναφέρεται στην παράμετρό της. Μέσα στην ομάδα της `catch` κάνουμε **διαχείριση της εξαίρεσης** (exception handling): λέμε στον υπολογιστή τι θέλουμε να κάνει όταν τη συλλάβει.

Στο παράδειγμα που δώσαμε δεν τονίζουμε και τόσο την υπεροχή αυτού του τρόπου. Ας δούμε μια άλλη παραλλαγή, κάπως πιο «εποικοδομητική»:

```
int main()
{
   for ( int k(-10); k <= 10; ++k )
   {
      try
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      }
      catch( double x )
      {
         cout << " v(" << (x+1e-5) << ") = " << v(x+1e-5) << endl;
      }
   } // for (int k...
} // main
```

που μας δίνει:

```
v(-5) = -2.33333
v(-4.5) = -3.06667
v(-3.99999) = -100001
v(-3.5) = -3.06667
v(-3) = -2.33333
v(-2.5) = -3.06667
v(-1.99999) = -100001
v(-1.5) = -3.06667
. . .
v(3.5) = -3.06667
v(4.00001) = -100001
v(4.5) = -3.06667
v(5) = -2.33333
```

Εδώ, όπως βλέπεις, στη διαχείριση της εξαίρεσης, *ξανακαλούμε τη συνάρτηση* με τιμή που σίγουρα δεν έχει πρόβλημα, αλλά είναι κοντά στην τιμή που έριξε την εξαίρεση. Βέβαια, σε μια πραγματική εφαρμογή, κάτι τέτοιο είναι απίθανο να έχει νόημα.

Βάζοντας διαφορετικά τις **try-catch** παίρνουμε αποτελέσματα παρόμοια με αυτά που παίρναμε με την *exit()*:

```
int main()
{
   try
   {
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
} // main
```

Αποτέλεσμα:

```
 v(-5) = -2.33333
 v(-4.5) = -3.06667
 η v δεν ορίζεται στο -4
```

Πάντως το πρόγραμμα είναι ουσιωδώς διαφορετικό:

- Στην αρχική μορφή *η συνάρτηση παίρνει την απόφαση* να διακόψει την εκτέλεση του προγράμματος καλώντας την *exit()*.
- Στη νέα μορφή η συνάρτηση στέλνει μήνυμα ότι συνάντησε ανυπέρβλητο πρόβλημα. Την απόφαση για το τι θα γίνει παίρνει ο «γενικός διεύθυντής», δηλαδή η **main**.

Θα πρέπει όμως, να διαλύσουμε δυο συνηθισμένες παρεξηγήσεις:

- Θα διαχειριζόμαστε πάντοτε τις εξαιρέσεις στη **main**; Όχι. Στη συνέχεια, θα δούμε πώς αποφασίζουμε πού και πώς μπορεί να γίνει η καλύτερη διαχείριση της κάθε εξαίρεσης.
- Οι εξαιρέσεις ρίχνονται μόνο από *συναρτήσεις* που καλούμε στην ομάδα **try**; Όχι! Αργότερα θα δεις παραδείγματα που θα έχουμε εντολές **throw** *μέσα στην ομάδα* της **try**.

Ας πούμε τώρα ότι στο ίδιο πρόγραμμα χρησιμοποιούμε και την *pqer*, που την αλλάζουμε ως εξής:

```
void pqer( double x, double y, double z, double& t, double& u )
{
   if ( x == y || x == -y || z <= 0 )
      throw " η pqer κλήθηκε με ορίσματα εκτός πεδίου ορισμού";
   t = x*y*pow(z, x-y)/(x*x-y*y);
   u = (x*y-1/x)/z;
} // pqer
```

Τώρα, η *pqer* ρίχνει εξαίρεση τύπου **char*** (το βέλος προς το μήνυμα).

Θα πεις: «Ναι, αλλά τώρα, όταν πιάσουμε την εξαίρεση, το πολύ-πολύ να γράψουμε το μήνυμα. Δεν μπορούμε όμως να έχουμε τις τιμές των *x*, *y*, *z* που προκάλεσαν το πρόβλημα και να τις χειριστούμε "πιο δημιουργικά".» Αργότερα θα μάθουμε πώς μπορούμε να περνούμε τέτοιες πληροφορίες.

Για να γράψουμε τη **main** έχουμε το εξής πρόβλημα: η *v* μπορεί να ρίξει αντικείμενο εξαίρεσης τύπου **double** ενώ η *pqer* μπορεί να ρίξει αντικείμενο τύπου **char***. Πώς τα συλλαμβάνουμε; Δες πώς γίνεται η **main**:

```
int main()
{
   double t, u;

   try
   {
```

```
   // . . .
      pqer( 1, 2, 3, t, u );
   // . . .
      for ( int k(-10); k <= 10; ++k )
      {
         cout << " v(" << (k/2.0) << ") = " << v(k/2.0) << endl;
      } // for (int k...
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
} // main
```

Όπως βλέπεις, μια `try` μπορεί να έχει περισσότερες από μια `catch` για να χειριστεί διάφορες εξαιρέσεις που προέρχονται από μια ή περισσότερες συναρτήσεις που καλούνται μέσα στην ομάδα της.

Αν βάλεις μια `catch( ... )` συλλαμβάνεις όλες τις εξαιρέσεις. Π.χ. στο τελευταίο παράδειγμα μπορείς να βάλεις:

```
   try
   {
   // . . .
   }
   catch( double x )
   {
      cout << " η v δεν ορίζεται στο " << x << endl;
   }
   catch( char* x )
   {
      cout << x << endl;
   }
   catch ( ... )
   {
      cout << " μη αναμενόμενη εξαίρεση" << endl;
   }
```

Τέλος, να πούμε ότι μπορείς, αν θέλεις, να δηλώνεις στην επικεφαλίδα της συνάρτησης τις **προδιαγραφές εξαιρέσεων** (exception specification) –δηλαδή τους τύπους των εξαιρέσεων που μπορεί να ρίξει– ως εξής:

```
double v( double x ) throw( double );
void pqer( double x, double y, double z,
           double& t, double& u ) throw( char* );
double f( double x ) throw( double, int );
```

Αν μια συνάρτηση ρίξει εξαίρεση εκτός προδιαγραφών προκαλεί διακοπή της εκτέλεσης του προγράμματος. Προσοχή όμως: η

```
double g( double x ) throw()
```

σημαίνει ότι *από τη g() δεν θα ριχτεί ούτε θα περάσει οποιαδήποτε εξαίρεση* και είναι διαφορετική από την

```
double g( double x )
```

που σημαίνει ότι *από την g() περιμένεις οποιαδήποτε εξαίρεση.*

14.9.1 Μια Ιστορία με Εξαιρέσεις

Τώρα, θα δούμε ένα πιο πολύπλοκο παράδειγμα για να σου δώσουμε μια καλύτερη αίσθηση της λειτουργίας του μηχανισμού των εξαιρέσεων. Ξαναγράφουμε το πρόγραμμα του παραδ. 1 της §7.7 ως εξής:

```
#include <iostream>

using namespace std;

unsigned long int comb( int m, int n );

int main()
{
   int m, n;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   try
   {
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << comb(m,n) << endl;
   }
   catch( int )
   {
      cout << " Λάθος δεδομένα" << endl;
   }
} // main

// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
unsigned long int natProduct( int m, int n )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// 0 < m <= n
   unsigned long int fv( m );
   for ( int k(m+1); k <= n; ++k )  fv *= k;
   return fv;
} // natProduct

// factorial -- Υπολογίζει το a!
unsigned long int factorial( int a )
{
   return (a == 0) ? 1 : natProduct(1, a);
} // factorial

// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
unsigned long int comb( int m, int n )
{
   unsigned long int fv;

   if ( n < m-n )  fv = natProduct(m-n+1, m)/factorial(n);
             else  fv = natProduct(n+1, m)/factorial(m-n);
   return fv;
} // comb
```

Κατ' αρχάς, να εξηγήσουμε τι κάνουμε: Ας πούμε ότι έχουμε να υπολογίσουμε τους $\binom{5}{2}$. Στο αρχικό πρόγραμμα ζητούσαμε τον υπολογισμό:

$$\binom{5}{2} = \frac{5!}{2!(5-2)!} = \frac{5!}{2!3!} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5}{2!\cdot 1\cdot 2\cdot 3}$$

Φυσικά, μπορούμε να απλοποιήσουμε το 1·2·3 και να αποφύγουμε άσκοπες πράξεις. Η *comb()* κάνει ακριβώς αυτό, επιλέγοντας τη μέγιστη απλοποίηση. Για τον σκοπό αυτόν

όμως χρειαζόμαστε τη `natProduct( int m, int n )` που υπολογίζει το γινόμενο φυσικών αριθμών: $m\cdot(m+1)\cdot\ldots\cdot(n-1)\cdot n$, με την προϋπόθεση $0 < m \le n$. Αν η *natProduct* κληθεί χωρίς να ισχύει η προϋπόθεση ρίχνει εξαίρεση.

Έχοντας τη *natProduct()* μπορούμε να γράψουμε, όπως βλέπεις, πολύ πιο απλά τη *factorial()*.

Ο μόνος έλεγχος εγκυρότητας των δεδομένων γίνεται στη *natProduct()*. Αυτό έχει ως αποτέλεσμα να είναι το πρόγραμμα καθαρογραμμένο!

Αν η *natProduct()* κληθεί με ορίσματα που δεν ικανοποιούν την $0 < m \le n$ ρίχνει την εξαίρεση (τύπου `int`) "`-1`". Ας πούμε λοιπόν ότι, από λάθος, ζητείται στη **main** ο υπολογισμός των συνδυασμών $\binom{8}{10}$.

- Η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση "`comb(m, n)`" της *comb()*. Επειδή η $n < m-n$ δεν ισχύει, η εκτέλεση συνεχίζεται στην περιοχή του `else`. Η *comb()* για να κάνει τον υπολογισμό θα ζητήσει τους υπολογισμούς "`natProduct(11, 8)`" και "`factorial(-2)`". Ας πούμε τώρα ότι ζητείται πρώτα ο υπολογισμός "`factorial(-2)`".
- Και πάλι η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *factorial()*. Η *factorial()*, με τη σειρά της, κάνει την κλήση `natProduct(1, -2)`.
- Και αυτήν τη φορά η στοίβα φορτώνεται με όσα απαιτούνται για την κλήση της *natProduct()*. Η *natProduct()* βρίσκει ότι έχουμε $m \le 0$, δεν κάνει υπολογισμούς ρίχνει την εξαίρεση "`-1`".
- Τα σχετικά με την κλήση της *natProduct()* φεύγουν από τη στοίβα και επιστρέφουμε στη *factorial()*.
- Η *factorial()* δεν συλλαμβάνει την εξαίρεση. Τα σχετικά με την κλήση της *factorial()* φεύγουν από τη στοίβα και επιστρέφουμε στην *comb()*.
- Αλλά ούτε η *comb()* έχει κάποια `catch(int ...)` για να συλλάβει την εξαίρεση· τα σχετικά με την κλήση της *comb()* φεύγουν από τη στοίβα και επιστρέφουμε στη **main** όπου η εξαίρεση συλλαμβάνεται.

Το πέρασμα του ελέγχου από κάθε συνάρτηση προς αυτήν που την κάλεσε χωρίς να γίνεται οποιαδήποτε άλλη δουλειά λέγεται **ξετύλιγμα της στοίβας** (stack unwinding). Αν η εξαίρεση δεν συλληφθεί ούτε στη **main** θα προκαλέσει τελικώς διακοπή της εκτέλεσης του προγράμματος.

Αργότερα θα δούμε τη διαχείριση των εξαιρέσεων πιο εκτεταμένα, αλλά προς το παρόν θα τονίσουμε ότι:

1. Ο μηχανισμός των εξαιρέσεων, που υλοποιείται με τις εντολές `throw`, `try` και `catch`, επιτρέπει στον προγραμματιστή να σχεδιάσει σωστά το πρόγραμμά του ώστε να διαχειρίζεται προβληματικές καταστάσεις.
2. Στις συναρτήσεις χωρίς τύπο, μπορείς, αν δεν θέλεις να χρησιμοποιήσεις εξαιρέσεις, να βάζεις μια παράμετρο από την οποίαν θα επιστρέφεις κάποια πληροφορία σχετικώς με το σφάλμα. Ξαναδές το παράδειγμα της §13.11. Πάντως το ξετύλιγμα της στοίβας με τις εξαιρέσεις είναι πιο απλό.
3. Σε συνέχεια του παραπάνω: Δες πόσο απλά και *καθαρά* γράφηκαν οι *factorial()* και *comb()*!

14.10 Αναδρομή (ξανά)

Όπως λέγαμε και στην §7.10, στη C++ υπάρχει η δυνατότητα αναδρομικής διατύπωσης μιας συνάρτησης, όπως και στα μαθηματικά υπάρχει η δυνατότητα αναδρομικής διατύπωσης μερικών ορισμών. Είδαμε ακόμη δύο παραδείγματα: μια συνάρτηση που υπολογίζει το $n!$ και μια που υπολογίζει τον μέγιστο κοινό διαιρέτη δύο ακεραίων και είχαμε παρατηρήσει

ότι ο αναδρομικός τρόπος, συγκρινόμενος με τον ισοδύναμο επαναληπτικό τρόπο, είναι πιό απλός, πιό σύντομος και πλησιέστερος προς τον μαθηματικό ορισμό.

Συχνά όμως, όπως θα δούμε παρακάτω, ο αναδρομικός τρόπος είναι λιγότερο αποδοτικός από τον επαναληπτικό και σε χρόνο και σε μνήμη.

Τί πληρώνουμε για αυτήν την καλύτερη γραφή; Για να γίνει ο υπολογισμός με την αναδρομική μορφή θα χρειαστούμε n+1 ενεργοποιήσεις της συνάρτησης, που κάθε μια παίρνει μνήμη από τη στοίβα. Χρησιμοποιούμε δηλαδή μνήμη ανάλογη του n και φυσικά αντίστοιχο χρόνο.

Ας δούμε δύο ακόμη παραδείγματα. Το πρώτο είναι κλασικό και πολυσυζητημένο. Ο (Dijkstra 1976), στη μονογραφία του "A Discipline of Programming", άφηνε έξω από το δομημένο προγραμματισμό την αναδρομή. Οι θιασώτες της φυσικά δεν το δέχτηκαν. Οι (Manna & Waldinger 1978), που δούλευαν ερευνητικώς στην αυτόματη σύνθεση προγράμματος, θεωρούσαν την αναδρομή απαραίτητη τουλάχιστον στην αρχική σχεδίαση· έγραψαν λοιπόν μια εργασία που είναι −οξύτερη από ό,τι συνηθίζεται στις επιστημονικές εργασίες− κριτική στις θέσεις του Dijkstra. Στην εργασία τους σχολιάζουν δυο (το 6ο και το 2ο) από τα «οκτώ μικρά παραδείγματα» του Dijkstra. Στη συνέχεια θα δούμε το πρώτο από αυτά τα παραδείγματα.

Παράδειγμα

Μας δίνονται δυο ακέραιοι, $x > 1$ και $y \geq 0$ (προϋπόθεση), και θέλουμε να βρούμε κάποιο z τέτοιο ώστε να ισχύει η (απαίτηση):

R: $z == x^y$

Ο Dijkstra προτείνει να χρησιμοποιήσουμε μια βοηθητική μεταβλητή h, τέτοια ώστε να ισχύει η

P: $h \cdot z == x^y$

και να γράψουμε μια

while ($h \neq 1$) «συμπίεσε» το h κρατώντας αναλλοίωτη την P

Για να ισχύει αρχικώς η αναλλοίωτη βάζουμε: "`h = x^y; z = 1`". Είναι φανερό ότι, όταν τελειώσει η εκτέλεση της **while**, θα έχουμε $h == 1$ που μαζί με την αναλλοίωτη μας δίνουν την R.

Η τιμή x^y δεν μας είναι γνωστή, ώστε να την εκχωρήσουμε αρχικά στην h. Το πρόβλημα αυτό μπορεί να λυθεί αν γράψουμε την h ως xx^{yy} οπότε η αναλλοίωτή μας γράφεται

P: $xx^{yy} \cdot z == x^y$

ενώ το πρόγραμμά μας γίνεται:

```
xx = x;  yy = y;  z = 1;
while (yy != 0)
   «συμπίεσε» το yy κρατώντας αναλλοίωτη την P
```

Η «συμπίεση» μπορεί να γίνεται με την εντολή "`yy = yy - 1`", οπότε η P παραμένει αναλλοίωτη αν συγχρόνως αλλάζουμε και την τιμή της `z = z*xx`. Είναι φανερό ότι η εκτέλεση της **while** κάποτε τερματίζεται. Να λοιπόν η συνάρτηση υπολογισμού της δύναμης:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      --yy;
      z *= xx;
   } // while
   return z;
```

```
} // power
```

Ύστερα από αυτό, ο Dijkstra ψάχνει για πιο γρήγορο αλγόριθμο. Παρατηρεί ότι αν ο *yy* είναι άρτιος και βάλουμε **xx = xx²** και **yy = yy/2** η τιμή της *h* δεν αλλάζει:

$$h == xx^{yy} == (xx^2)^{yy/2}$$

Έτσι, καταλήγει στην[20]

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int xx( x ), yy( y ), z( 1 );
   while ( yy != 0 )
   {
      while ( even(yy) )
      {
         xx *= xx;
         yy /= 2;
      } // while (even(yy))
      --yy;
      z *= xx;
   } // while
   return z;
} // power
```

Ο τερματισμός της εσωτερικής **while** είναι σίγουρος, διότι ο μόνος άρτιος που μπορεί να διαιρείται επ' άπειρον δια 2, χωρίς να δώσει περιττό, είναι ο 0 (μηδέν), αλλά από αυτόν μας προστατεύει η εξωτερική **while**.

Ενώ η πρώτη λύση έχει χρόνο εκτέλεσης ανάλογο του *y*, η δεύτερη έχει χρόνο εκτέλεσης ανάλογο του $log_2 y$.

Οι Manna και Waldinger αντιτείνουν ότι η αναλλοίωτη, η συνθήκη συνέχισης και –τελικώς– το πρόγραμμα δεν βγήκαν, αλλά γράφτηκαν μια και ήταν γνωστά εκ των προτέρων. Και δίνουν, ως το πιο απλό πρόγραμμα που θα μπορούσε να γραφτεί για την περίπτωση, το:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
   if ( y == 0 )  fv = 1;
            else  fv = x * power( x, y-1 );
   return fv;
} // power
```

Τί πιο απλό! Δεν γράφουμε παρά τις βασικές ιδιότητες: $x^0 == 1$ και $x^y == x \cdot x^{y-1}$.

Η βελτίωση του αλγορίθμου, η αντίστοιχη αυτής που κάνει ο Dijkstra, είναι επίσης απλούστατη:

```
unsigned int power( int x, int y )
{
   if ( x <= 1 || y < 0 )
   {
      throw -1;
   }
   unsigned int fv;
```

[20] Η *even* παίρνει ένα όρισμα τύπου **int** και μας δίνει **true** αν το όρισμα είναι άρτιος (even), **false** αν το όρισμα είναι περιττός. Θα μπορούσες να τη γράψεις ως εξής:

```
bool even( int x )  {  return ( x % 2 == 0 );  }
```

```
   if ( y == 0 )
      fv = 1;
   else if ( even(y) )
   {  unsigned int z( power(x,y/2) );
      fv = z*z;  }
   else
      fv = x * power( x, y-1 );
   return fv;
} // power
```

και αν είχες πρόβλημα να καταλάβεις το βελτιωμένο πρόγραμμα του Dijkstra, δεν πρέπει να έχεις δυσκολία να καταλάβεις αυτό εδώ!

Δεν μπορούμε παρά να θαυμάσουμε την ομορφιά και την απλότητα του αναδρομικού προγράμματος. Αλλά, δεν είναι δωρεάν! Τι πληρώνουμε; Για σκέψου πώς θα εκτελεσθεί αυτή η συνάρτηση; Οι διαδοχικές αναδρομικές κλήσεις εισάγουν στη στοίβα τις τιμές y, y-1,... 1, 0. Στη συνέχεια αφαιρεί όλες αυτές τις τιμές, υπολογίζοντας το γινόμενο $1 \cdot x \cdot x \cdot \ldots \cdot x$. Στη βελτιωμένη περίπτωση οι κλήσεις είναι, στην καλύτερη περίπτωση, $log_2 y$. Μικρό το κακό (φυσικά για μικρά y);

Πάντως, υπάρχουν και περιπτώσεις όπου η απλότητα του αναδρομικού προγράμματος δρα σαν «σειρήνα» που μας τραβάει στον «κακό δρόμο». Τυπικό παράδειγμα οι αριθμοί Fibonacci, που ως γνωστόν ορίζονται ως εξής:[21]

$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$ για $n \geq 2$

Μεταφράζουμε λοιπόν αμέσως:

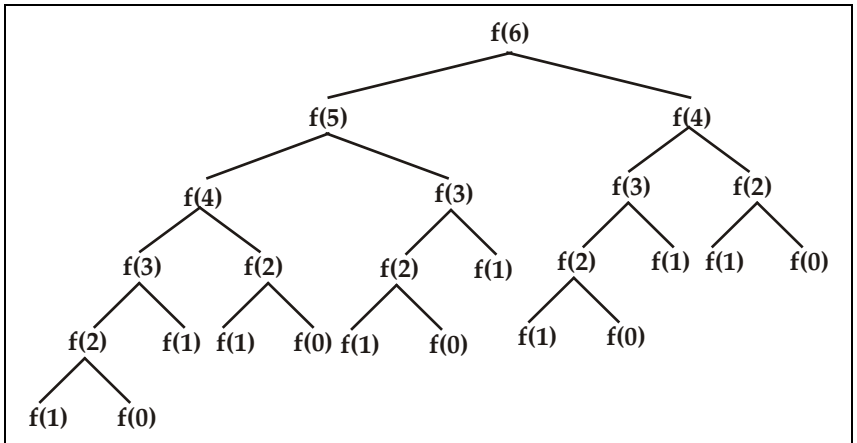
```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fv;
   if ( n < 2 )  fv = n;
           else  fv = f(n-1) + f(n-2);
   return fv;
} // f
```

και στο Σχ. 14-1 βλέπεις τι γίνεται για να υπολογιστεί ο $f(6)$. Ο $f(4)$ υπολογίζεται 2 φορές, το $f(3)$ 3, το $f(2)$ 5 και το $f(1)$ 8.

Στην περίπτωση αυτή συμφέρει να γράψουμε μια επαναληπτική συνάρτηση, φροντίζοντας να φυλάγουμε *δυο* προηγούμενους αριθμούς σε κάθε βήμα:

```
unsigned int f( int n )
{
   if ( n < 0 )
   {
      throw n;
   }
   unsigned int fn, fp, fpp;
   if ( n < 2 )
   {
      fn = n;
   }
   else
   {
      fp = 0;  fn = 1;
      for ( int j(1); j < n; ++j )
      {
         fpp = fp;  fp = fn;
         fn = fp + fpp;
      } // for
```

[21] Έλυσες την Άσκηση 7-14;



Σχ. 14-1 Αναδρομικές κλήσεις για τον υπολογισμό f(6).

```
   } // if (n < 2)
   return fn;
} // f
```

Για τον υπολογισμό της δύναμης είπαμε «μικρό το κακό»· εδώ δεν μπορούμε να πούμε το ίδιο. Θα γράφουμε αναδρονικές συναρτήσεις, αλλά καλό θα είναι να σκεφτόμαστε που οδηγούν (και πόσο καλές είναι άλλες εναλλακτικές λύσεις).

Ας ξαναγυρίσουμε στο παράδειγμα της *power()*. Μερικοί μπορεί να αναρωτιούνται «Είναι δυνατόν, στο επαναληπτικό πρόγραμμα να σπαζοκεφαλιάζω για αναλλοίωτες και για τερματισμούς, ενώ όταν γράφω το αναδρομικό να μην υπάρχουν τέτοια προβλήματα;» Όχι βέβαια! Δεν συμβαίνει κάτι τέτοιο. Εκείνο που συμβαίνει είναι ότι αυτά τα πράγματα είναι, συχνά, πολύ απλά για το αναδρομικό πρόγραμμα. Η αναλλοίωτη εμφανίζεται τώρα σαν συμμόρφωση προς τις προδιαγραφές της συνάρτησης· δηλαδή, η κλήση μιας συνάρτησης είτε απ' έξω είτε από μέσα από το σώμα του υποπρογράμματος, με αναδρομή, θα πρέπει να γίνεται με τους ίδιους όρους. Για να έχουμε τερματισμό, σε μια αναδρομική συνάρτηση *θα πρέπει να υπάρχει μια τουλάχιστον περίπτωση εκτέλεσης χωρίς αναδρομική κλήση*. Ακόμη, *οι αναδρομικές κλήσεις θα πρέπει να οδηγούν σε μια τέτοια εκτέλεση*.

Στην *power()* οι προδιαγραφές είναι:

το όρισμα που αντιστοιχεί στη x πρέπει να είναι θετικός ακέραιος,
το όρισμα που αντιστοιχεί στη y, πρέπει να είναι μη αρνητικός ακέραιος,
το αποτέλεσμα είναι ίσο με x^y.

Οι δυο προϋποθέσεις ελέγχονται στην αρχή της συνάρτησης. Φυσικά, δεν φτάνει αυτό: Κάθε αναδρομική κλήση μειώνει την τιμή της y κατά 1 εκτός από την περίπτωση που έχουμε y == 1. Η παραπάνω παρατήρηση μας βεβαιώνει και για τον τερματισμό. Για y == 1 δεν έχουμε αναδρομική κλήση και είναι σίγουρο ότι θα φτάσουμε σε μια κλήση της συνάρτησης με y == 1.

14.11 * Ακαθόριστο Πλήθος Παραμέτρων

Είδαμε πιο πριν ότι η κλήση μιας συνάρτησης μπορεί να έχει λιγότερες πραγματικές από τις τυπικές παραμέτρους. Η C++ σου επιτρέπει να γράφεις συναρτήσεις με μη προκαθορισμένο πλήθος παραμέτρων.

Ας πούμε ότι θέλουμε μια συνάρτηση που θα τροφοδοτείται με μια τιμή τύπου `char` και μη προκαθορισμένο πλήθος τιμών τύπου `double` και αν η τιμή τύπου `char` είναι `'>'` θα μας επιστρέφει ως τιμή τη μέγιστη από τις τιμές τύπου `double` ενώ αν είναι `'<'` θα μας επιστρέφει την ελάχιστη.

Σύμφωνα με όσα έχουμε μάθει, η συνάρτησή μας, ας την πούμε *maxormin()*, θα είναι συνάρτηση με τύπο: θα επιστρέφει τιμή τύπου **double**. Θα έχει μια παράμετρο τιμής, τύπου **char**, ας την πούμε *tel*, δηλαδή:

```
double maxormin(char tel, ...)
```

και μετά; Ε λοιπόν: η C++ δέχεται αυτήν την επικεφαλίδα·[22] τα **αποσιωπητικά** (ellipsis) δείχνουν ότι, όταν κληθεί, μπορεί να πάρει πολλά ορίσματα.

Δες πώς γράφεται η συνάρτηση και ας τη συζητήσουμε στη συνέχεια:

```
#include <iostream>
#include <cstdarg>

using namespace std;

double maxormin( char tel, ... )
{
   if ( tel != '<' && tel != '>' )
   {
      throw tel;
   }
   va_list ap;
   va_start( ap, tel );

   double x( va_arg(ap, double) );
   if ( x == 0 )
   {
      throw 0;
   }
   double fv( x );
   if ( tel == '>' )
   {
      while ( x != 0 )
      {
         if ( x > fv )  fv = x;
         x = va_arg( ap, double );
      }
   }
   else // tel == '<'
   {
      while ( x != 0 )
      {
         if ( x < fv )  fv = x;
         x = va_arg( ap, double );
      } // while (x != 0)
   } // if (tel == '>')
   va_end( ap );
   return fv;
} // maxormin

int main()
{
   double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
   double mn = maxormin( '<', 1.1, 2.2, 1.3, 0.4, 7.1, 0.0 );
   cout << mx << "  " << mn << endl;
} // main
```

1. Περιλάβαμε στο πρόγραμμά μας το cstdarg. Στο αρχείο αυτό ορίζεται ο τύπος *va_list* και δηλώνονται οι συναρτήσεις *va_arg()*, *va_start()* και *va_end()*.

2. Μέσα στη συνάρτησή μας δηλώσαμε τη μεταβλητή *ap* τύπου *va_list*. Πρόκειται για έναν πίνακα.

[22] Κληρονομιά από τη C!

3. Με την κλήση "`va_start(ap, tel)`" τροφοδοτούμε τη *va_start()* με τη μοναδική σταθερή παράμετρο (*tel*) και αυτή μας επιστρέφει πληροφορίες για τις υπόλοιπες πραγματικές παραμέτρους στην *ap*.

4. Καλούμε ξανά και ξανά τη "`va_arg(ap, double)`". Κάθε φορά μας επιστρέφει ως τιμή *την τιμή του επόμενου ορίσματος* και αλλάζει την τιμή της *ap* ώστε στη συνέχεια να προχωρήσουμε παρακάτω (στο μεθεπόμενο). Πρόσεξε ότι δεύτερο όρισμα στην κλήση της *va_arg* είναι ένας *τύπος* (!). Είναι *ο τύπος του ορίσματος που περιμένουμε να διαβάσουμε*.

5. Στο τέλος βάζουμε την κλήση της `va_end(ap)`. Είναι απαραίτητη για την ολοκλήρωση της κλήσης (αυτή θα κάνει ανάταξη της στοίβας).

Ας πούμε ότι η συνάρτησή μας καλείται να βρει τον μέγιστο. Ξεχνούμε για λίγο τις δύο **if** και παρακολουθούμε την εκτέλεση:

```
x = va_arg( ap, double );  // πάρε το πρώτο όρισμα στη x...
fv = x;                    // ... και θεώρησε ότι είναι μέγιστο
x = va_arg( ap, double );  // πάρε το επόμενο όρισμα στη x...
while ( x != 0 )           // όσο δεν είναι 0 (μηδέν)
{
   if ( x > fv )  fv = x;   //  αν x > fv διόρθωσε το μέγιστο
   x = va_arg( ap, double );//  πάρε το επόμενο όρισμα στη x...
} // while
```

Ο τρόπος που υπολογίζουμε το μέγιστο είναι ο γνωστός αλλά είναι φανερό ότι η **while** ελέγχεται με φρουρό την τιμή 0 (μηδέν) στη *x*! Ακριβώς! Όπως μπορείς να δεις στην κλήση της συνάρτησης έχουμε:

```
double mx = maxormin( '>', 1.1, 2.2, 2.0, 0.4, 0.0 );
```

Το "`0.0`" στο τέλος είναι φρουρός! Δεν θα μπορούσαμε να βάλουμε μετρούμενη επανάληψη; Βέβαια, αρκεί να βάλουμε άλλη μια σταθερή παράμετρο στην αρχή (1η ή 2η) που να περνάει στη συνάρτηση το πλήθος των ορισμάτων που ακολουθούν.

Αν θελήσεις να γράψεις τέτοια συνάρτηση (συνήθως δεν είναι η καλύτερη λύση) πρόσεξε τα εξής:

1. Ο φρουρός θα πρέπει να είναι του ίδιου τύπου με τα άλλα ορίσματα και να επιλέγεται όπως ξέρουμε (το "`0.0`" στο παράδειγμά μας δεν είναι πολύ καλή επιλογή). Αν προτιμήσεις να περάσεις το πλήθος πρόσεξε να μετρήσεις σωστά!

2. Η συνάρτησή σου θα πρέπει να έχει μια τουλάχιστον σταθερή παράμετρο. Είναι προφανές ότι οι σταθερές παράμετροι μπαίνουν στην αρχή.

3. Αν έχεις περισσότερες από μία σταθερές παραμέτρους, στην κλήση της *va_start() θα βάλεις ως δεύτερη παράμετρο την τελευταία σταθερή παράμετρο* που πρέπει να μην είναι παράμετρος αναφοράς.

4. Οι μη σταθερές παράμετροι θα πρέπει να είναι του ίδιου τύπου ή να εμφανίζουν κάποιο σταθερό σχήμα επανάληψης (π.χ. μια τύπου `char` μια τύπου `int`). Μόνον έτσι μπορείς να χρησιμοποιήσεις επαναληπτική εντολή (και έχει νόημα αυτό το είδος της συνάρτησης).

5. Οι πραγματικές παράμετροι θα πρέπει να συμφωνούν πλήρως με τους τύπους που βάζεις στην κλήση της *va_arg()*. Αν, στο παράδειγμά μας βάζαμε ως 3η παράμετρο όχι 2.0 αλλά 2 θα είχαμε πρόβλημα.

14.12 Συνοψίζοντας...

Κυρίως σε μαθηματικές εφαρμογές (αλλά όχι μόνον), παρουσιάζεται η ανάγκη γράψουμε συναρτήσεις με παράμετρο συνάρτηση. Η C++ λύνει αυτό το πρόβλημα με παράμετρο-βέλος-προς-συνάρτηση.

Πολύ συχνά παρουσιάζεται η ανάγκη να αλλάξουμε τύπους κάποιων μεταβλητών. Είναι πολύ σημαντικό να μην χρειάζεται να αλλάξουμε και τα ονόματα συναρτήσεων ή να

αλλάξουμε πράξεις με τελεστές σε κλήσεις συναρτήσεων. Η *επιφόρτωση* συναρτήσεων και τελεστών είναι η τεχνική με την οποία λύνουμε αυτά τα προβλήματα.

Αν οι συναρτήσεις που επιφορτώνονται διαφέρουν μόνο σε τύπους δεδομένων και δώσουμε στον μεταγλωττιστή ένα σχέδιο, το *περίγραμμα* (template), αυτός θα τις γράψει μόνος του.

Ας πούμε τώρα ότι για να λύσεις ένα συγκεκριμένο πρόβλημα γράφεις κάποια ή κάποιες συναρτήσεις που καταλαβαίνεις ότι θα σου είναι χρήσιμη/ες γενικότερα. Τα περιγράμματα συναρτήσεων και οι συναρτησιακές παράμετροι είναι δύο εργαλεία που σου επιτρέπουν να δώσεις την κατάλληλη παραμετρική μορφή ώστε να τα χρησιμοποιείς με ευκολία και στο μέλλον.

Πιθανότατα μεγαλύτερης σπουδαιότητας είναι ένα άλλο εργαλείο που είδαμε στο κεφάλαιο αυτό: οι *εξαιρέσεις*. Μαθαίνοντας να τις χρησιμοποιείς θα δεις ότι θα αλλάξει συνολικώς ο τρόπος που γράφεις τα προγράμματά σου: θα γίνονται πιο ευέλικτα, πιο λειτουργικά και –καθώς θα μαθαίνεις πώς να χειρίζεσαι τις εξαιρέσεις– πιο καλοσχεδιασμένα και με λιγότερα λάθη. Οι δομές (κλάσεις) εξαιρέσεων θα είναι ένα πολύ σοβαρό εργαλείο για την ανάπτυξη αλλά και τη συντήρηση του λογισμικού που γράφεις.

Τρία θέματα που ασχοληθήκαμε ακόμη ήταν

- Οι *συναρτήσεις* **inline**: με αυτές μπορείς να επιταχύνεις κάπως την εκτέλεση του προγράμματός σου.
- Η *αναδρομή*: ένα πολύ καλό εργαλείο που έχει όμως και μερικές παγίδες. Τώρα μπορείς να το χρησιμοποιείς «μετά λόγου γνώσεως».
- Το *ακαθόριστο πλήθος παραμέτρων*: Βάλαμε αυτήν την παράγραφο μόνο για να καταλαβαίνεις συναρτήσεις που έχουν γραφεί έτσι. *Μη χρησιμοποιείς αυτήν τη δυνατότητα.* Αντιθέτως, *η δυνατότητα να παραλείπεις παραμέτρους που έχουν ερήμην καθορισμένη τιμή είναι μια χαρά.*

Ασκήσεις

Α Ομάδα

14-1 (Γενίκευση της Ασκ.9-5) Γράψε μια

```
double vectorSumIf( const double x[], int n, bool (*predic)(double) )
```

που θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων x[*k*] του x για τα οποία ισχύει η *predic*(x[*k*]).

Β Ομάδα

14-2 Θέλουμε συνάρτηση με όνομα *pluMin()* που

α) όταν καλείται με δύο ορίσματα *a*1, *a*2, τύπου `int`, θα αυξάνει την τιμή του πρώτου ορίσματος, *a*1, κατά 1 και θα μειώνει την τιμή του *a*2 κατά 1.

β) όταν καλείται με τέσσερα ορίσματα *a*1, *a*2, *b*1, *b*2 τύπου `int`, θα προσθέτει στην τιμή του *a*1 την τιμή του *b*1 και θα αφαιρεί από την τιμή του *a*2 την τιμή του *b*2.

Τι συνάρτηση θα γράψουμε, με τύπο ή χωρίς τύπο; Δικαιολόγησε την απάντησή σου.

Για κάθε μια παράμετρο δικαιολόγησε όλα τα χαρακτηριστικά της (τιμής ή αναφοράς, τύπο κλπ).

Γράψε τη συνάρτηση και δώσε παράδειγμα χρήσης με 2 και 4 ορίσματα. Στην κάθε περίπτωση θα βάζεις μέσα σε σχόλια τις τιμές των μεταβλητών (που χρησιμοποιούνται) πριν και μετά την κλήση.

Η συνάρτηση που έγραψες μπορεί να κληθεί με 3 ορίσματα; Δικαιολόγησε την απάντησή σου· αν είναι θετική δώσε παράδειγμα όπως παραπάνω.

14-3 (Να την απαντήσεις εσύ και *όχι ο μεταγλωττιστής C++ που χρησιμοποιείς*.) Έστω ότι έχουμε:[23]

```
template<typename T1, typename T2>
void f( T1, T2 );                       // 1
template<typename T> void f( T );       // 2
template<typename T> void f( T, T );    // 3
template<typename T> void f( T* );      // 4
template<typename T> void f( T*, T );   // 5
template<typename T> void f( T, T* );   // 6
template<typename T> void f( int, T* ); // 7
template<> void f<int>( int );          // 8
void f( int, double );                  // 9
void f( int );                          // 10
```

Αν

```
int             i;
double          d;
float           ff;
```

ποια από τις παραπάνω θα κληθεί για την κάθε μια από τις:

```
f( i );         // a
f<int>( i );    // b
f( i, i );      // c
f( i, ff );     // d
f( i, d );      // e
f( i, &d );     // f
f( &d, d );     // g
f( &d );        // h
f( d, &i );     // i
f( &i, &i );    // j
```

Γ Ομάδα

14-4 Όπως είναι γνωστό, το ορισμένο ολοκλήρωμα: $\int_a^b f(x)dx$ υπολογίζει το εμβαδό που περικλείεται ανάμεσα στον άξονα *x'x* και στην καμπύλη *y* = *f* (*x*), από *x* = *a* μέχρι *b*. Μια αριθμητική μέθοδος για να προσεγγίσουμε την τιμή του ολοκληρώματος είναι αυτή του **μέσου** (midpoint):

Διαιρούμε το διάστημα [*a*,*b*] σε *N* ίσα τμήματα που το καθένα έχει μήκος $h = (b - a)/N$. Κατασκευάζουμε *N* παραλληλόγραμμα, που το καθένα έχει βάση *h* και ύψος $f(x_k)$, όπου $x_k = a + (k-0.5)h$, είναι το μέσο του *k* διαστήματος. Το άθροισμα των εμβαδών των παραλληλογράμμων μας δίνει μια προσέγγιση του ολοκληρώματος.

$$\int_a^b f(x)dx \approx h\sum_{k=1}^{N} f(x_k)$$

Γράψε μια συνάρτηση *midPoint* που θα τροφοδοτείται με τη συνάρτηση που έχουμε να ολοκληρώσουμε, τα άκρα του διαστήματος της ολοκλήρωσης και το *N* και θα υπολογίζει και θα επιστρέφει την προσέγγιση της τιμής του ολοκληρώματος με τη μέθοδο του μέσου. Τέλος θα έχει μία ακόμη παράμετρο, τη *errCode*· αν η *midPoint* κληθεί με *N* < 1, δεν θα γίνονται υπολογισμοί και η *errCode* θα επιστρέφει τιμή 1, αλλιώς, αν όλα πάνε καλά, θα επιστρέφει τιμή 0.

14-5 Στην §14.3, λέγαμε: Αν m_0 είναι το μέσο του διαστήματος [a_0,b_0], ελέγχουμε την $f(m_0)$.

```
Αν f(a0)f(m0) ≤ 0 τότε
   ψάχνουμε για τη λύση στο διάστημα [a0,m0]
```

[23] Από το (Sutter 1998).

```
    αλλιώς
       ψάχνουμε για τη λύση στο διάστημα [m0,b0]
```

και γράψαμε σε C++:

```
    a = a0;  b = b0;
    m = (a + b) / 2;
    if (f(a)*f(m) <= 0.0)  b = m;
                   else  a = m;
```

Αλλά, το «ψάχνουμε για τη λύση στο διάστημα [a_0,m_0]» μπορεί να μεταφρασθεί και ως εξής: «κάλεσε τον εαυτό σου για να ψάξει λύση στο διάστημα [a_0, m_0]». Παρομοίως μπορεί να μεταφρασθεί και το «ψάχνουμε για τη λύση στο διάστημα [m_0,b_0]». Με βάση αυτές τις παρατηρήσεις γράψε μια *αναδρομική* μορφή της *bisection*.

14-6 Με βάση το πρόγραμμα για τη συγχώνευση ταξινομημένων πινάκων (§9.4) γράψε περίγραμμα συνάρτησης:

```
template<typename T> void merge( T v[], T w[], T z[],
                                 int pv, int tv, int pw, int tw,
                                 int pz, int tz,
                                 bool& ok )
```

που θα συγχωνεύει τα τμήματα v[*pv*]...v[*tv*] και w[*pw*]...w[*tw*] των πινάκων v, w στο z[*pz*]...z[*tz*]. Προφανώς θα πρέπει να έχουμε $0 \le pv \le tv$, $0 \le pw \le tw$ και $tz - pz + 1 \ge (tv - pv + 1) + (tw - pw + 1)$. Αν δεν ισχύουν οι παραπάνω, η *ok* επιστρέφει `false`. Τα τμήματα v[*pv*]...v[*tv*] και w[*pw*] ... w[*tw*] πρέπει να είναι ταξινομημένα κατ' αύξουσα τάξη. Την ίδια ταξινόμηση θα έχει και το z[*pz*]... z[*tz*].

14-7 Στο τέλος της §9.6 δίναμε μια άλλη ιδέα για ταξινόμηση:

- Κόβουμε τον πίνακά μας στη μέση και έτσι έχουμε δυο πίνακες με μήκος $\frac{1}{2}N$ ο καθένας.
- Ταξινομούμε τον κάθε ένα από αυτούς, σε χρόνο $\lambda(\frac{1}{2}N)^2 = \frac{1}{4}\lambda N^2$. Συνολικά: $\frac{1}{2}\lambda N^2$.
- Συγχωνεύουμε τους δυο πίνακες σε έναν, σε χρόνο περίπου κN, που για μεγάλα N, είναι αμελητέος μπροστά στο $\frac{1}{2}\lambda N^2$.

  Με τις διαδικασίες που γράψαμε, αυτό θα μπορούσε να γίνει ως εξής:

```
middle = (from + upto) / 2;
ταξινόμησε το τμήμα v[from]… v[middle]
ταξινόμησε το τμήμα v[middle+1]… v[upto]
συγχώνευσε στον πίνακα z τα δύο τμήματα
```

Βρήκαμε δηλαδή έναν τρόπο να υποδιπλασιάσουμε το χρόνο ταξινόμησης. Τι πληρώσαμε; Διπλασιάσαμε τις απαιτήσεις σε μνήμη (πίνακας z). Ύστερα απ' αυτό, δεν μπορούμε να μη σκεφτούμε: «*γιατί να μην ταξινομήσουμε τα δυο μισά με τον ίδιο τρόπο;*» Γιατί όχι; Αυτή ακριβώς είναι η ιδέα για την *ταξινόμηση με συγχώνευση* (merge sort) που ταξινομεί έναν πίνακα με N στοιχεία σε χρόνο ανάλογο του $N\log(N)$, που φυσικά είναι καλύτερος από N^2.

Γράψε και δοκίμασε τη *mergeSort()* για πίνακες με στοιχεία τύπου *string*. Μετάτρεψε τη συνάρτηση που έγραψες σε περίγραμμα.