κεφάλαιο

13

Συναρτήσεις ΙΙ – Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:

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

Προσδοκώμενα αποτελέσματα:

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

Έννοιες κλειδιά:

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

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

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

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

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

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

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

Έτσι, το πρόγραμμα θα μας δώσει:

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

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

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

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

13.1 Ένα Παλιό Πρόβλημα Ξανά

Στην §2.7 είχαμε λύσει το εξής πρόβλημα:

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

Είχαμε κάνει ένα σχέδιο για τη λύση:

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

και γράψαμε το πρόγραμμα.

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

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

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

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

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

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

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

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

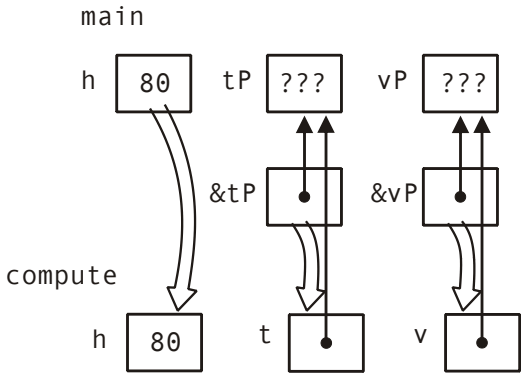
```
int main()
{
   const double  g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας

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

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

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

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



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

13.2 Επιστροφή Τιμών από τη Συνάρτηση I

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

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

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

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

και την καλούμε ως εξής:

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

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

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

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

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

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

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

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

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

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

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

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

Και να πώς γίνεται τελικώς το πρόγραμμά μας:

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

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

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

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

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

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

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

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

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

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

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

13.3 Επιστροφή Τιμών από τη Συνάρτηση II

Θα ξεκινήσουμε με έναν κανόνα, που θα εξηγηθεί αργότερα:

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

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

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

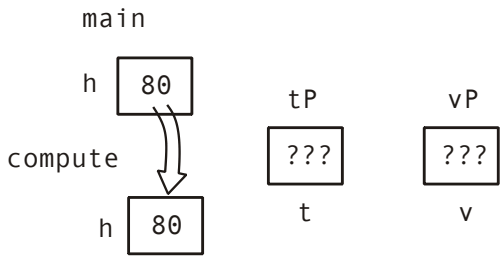
Ξαναγράφουμε ολόκληρο το πρόγραμμα της προηγούμενης παραγράφου χρησιμοποιώντας, αντί για βέλη, παραμέτρους αναφοράς:

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

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

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

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



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

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

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

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

Πρόσεξε την επικεφαλίδα της *compute()*:

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

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

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

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

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

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

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

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

Αλλά τώρα προσοχή:

Δηλαδή, αν έχεις τη

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

και τις:

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

μπορείς να βάλεις:

```
inputH( d );
```

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

Δεν μπορείς να βάλεις:

```
inputH( i );
```

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

Δεν μπορείς να βάλεις:

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

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

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

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

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

13.3.1 Παράμετρος `unsigned`; (ξανά)

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

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

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

13.4 Τύποι Αναφοράς

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

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

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

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

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

θα πάρουμε:

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

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

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

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

Αποτέλεσμα:

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

Τι διαφορές έχουμε;

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

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

Παράδειγμα

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

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

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

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

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

Πρόσεξε ότι:

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

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

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

μπορούμε να γράψουμε:

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

που θα μας δώσουν:

```
33   33
```

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

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

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

θα δώσουν:

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

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

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

και η παράξενη εκχώριση απαγορεύεται!

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

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

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

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

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

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

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

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

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

   return mxP;
} // maxNdx
```

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

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

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

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

13.5 Η Εντολή `return` (ξανά)

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

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

Παράδειγμα

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

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

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

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

13.6 Εμβέλεια και Χρόνος Ζωής Μεταβλητών

Όπως λέγαμε και στο Κεφ. 7,

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

Τώρα θα σταθούμε σε δύο σημεία:

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

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

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

```
using namespace std;

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

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

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

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

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

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

Όπως βλέπεις:

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

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

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

```
   int x;

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

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

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

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

```
2.5  4
```

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

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

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

Σε ένα πρόγραμμα C++:

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

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

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

Αν προσπαθήσεις να το μεταγλωττίσεις θα πάρεις μήνυμα:

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

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

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

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

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

που δίνει:

Πλαίσιο 13.1

Κανόνας Εμβέλειας

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

Το ίδιο πράγμα με άλλα λόγια:

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

Κανόνας Διάρκειας Ζωής

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

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

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

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

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

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

μας δίνει:

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

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

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

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

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

Όλα αυτά υπάρχουν μαζεμένα στο Πλ. 13.1.

Υπάρχουν και δύο ειδικές περιπτώσεις:

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

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

13.6.1 * Στατικές Μεταβλητές

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

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

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

Ας πούμε ότι έχουμε και την:

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

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

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

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

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

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

και να τι παίρνουμε:

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

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

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

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

13.6.2 Καθολικά Αντικείμενα και Τεκμηρίωση

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

- Επαληθευθούν
- Τροποποιηθούν.

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

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

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

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

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

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

13.7 * Οι Συναρτήσεις στις Αποδείξεις (ξανά)

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

Ξεκινούμε με δύο παραδείγματα:

Παράδειγμα 1

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

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

Αν τώρα βάλουμε σε μια συνάρτηση την εντολή:

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

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

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

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

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

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

Παράδειγμα 2

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

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

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

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

Επομένως στη συνάρτηση θα πρέπει να αποδείξουμε:

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

☜☜☜

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

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

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

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

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

13.8 Ορμαθοί C και Αριθμοί (ξανά)

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

Ας πούμε ότι δίνουμε:

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

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

Ας πούμε ότι έχουμε δηλώσει:

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

Οι παρακάτω εντολές:

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

```
            else  cout << d << endl;
```

θα δώσουν:

```
1.23457e+010
1.23  ab45678
```

Πώς είναι δηλωμένη η *strtod()* (στο **cstdlib**); Έτσι:

```
double strtod( const char* s, char** endptr );
```

Δεν έχεις πρόβλημα να καταλάβεις την πρώτη παράμετρο: Βέλος προς πίνακα χαρακτήρων. Η δεύτερη όμως; Μέσω της δεύτερης η συνάρτηση θα επιστρέψει ένα βέλος, είτε 0 (**NULL**) είτε προς τον πρώτο παράνομο χαρακτήρα. Στην §13.2 είδαμε πώς επιστρέφει η C τιμές μέσω παραμέτρων: αν η τιμή είναι τύπου *T*, βάζουμε μια παράμετρο τύπου *T** και εκεί –όταν καλούμε τη συνάρτηση– αντιστοιχίζουμε μια διεύθυνση της μνήμης όπου θα αποθηκευτεί η επιστρεφόμενη τιμή. Στην περίπτωσή μας θα επιστραφεί μια τιμή τύπου **char***· θα πρέπει λοιπόν να βάλουμε μια παράμετρο τύπου **(char*)***. Όταν καλούμε τη συνάρτηση θέλουμε να αποθηκεύσουμε θέλουμε η επιστρεφόμενη τιμή να αποθηκευτεί στην *p* (που είναι τύπου **char***). Δίνουμε λοιπόν την διεύθυνση της *p*, δηλαδή **&p**.

Αν κατάλαβες αυτό το σημείο, τα υπόλοιπα είναι απλά.

Σαν τη *strtod()* είναι και οι:

```
float strtof( const char* s, char** endptr );
long double strtold( const char* s, char** endptr );
```

Μπορείς να βλέπεις την *atof()* (§10.13.1) ως

```
double atof( const char* s )
{  return strtod( s, NULL );  }
```

Παρομοίως δουλεύει και η *strtol()*, που είναι δηλωμένη ως εξής:

```
long strtol( const char* s, char** endptr, int radix )
```

Όπως βλέπεις, αυτή έχει και μια τρίτη παράμετρο όπου πρέπει να βάλουμε τη βάση του αριθμητικού συστήματος στο οποίο είναι γραμμένος ο ακέραιος που παριστανει ο ορμαθός. Π.χ. οι:

```
   l = strtol( "11", &p,  2 );   cout << l << endl;
   l = strtol( "11", &p,  8 );   cout << l << endl;
   l = strtol( "11", &p, 10 );   cout << l << endl;
   l = strtol( "11", &p, 16 );   cout << l << endl;
```

δίνουν:

```
3
9
11
17
```

(3 = 2 + 1, 9 = 8 + 1, 11 = 10 + 1, 17 = 16 + 1)

Ακόμη, οι:

```
   long int l( strtol(s1, &p, 10) );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
   l = strtol( s2, &p, 10 );
   if ( p != 0 )  cout << l << "  " << p << endl;
            else  cout << l << endl;
```

δίνουν:

```
12345
1  .23456789e10
```

Τα παραπάνω σου δείχνουν και τον τρόπο χρήσης των δύο συναρτήσεων: αφού τις καλέσεις ελέγχεις το *p*:

- Αν είναι 0 όλα πήγαν καλά, δηλαδή ολόκληρος ο ορμαθός μετατράπηκε στην αριθμητική τιμή που σου επίστρεψε τη συνάρτηση και μπορείς να τη χρησιμοποιήσεις.

- Αν δεν είναι 0 τότε δεν έγινε πλήρης μετατροπή διότι υπήρχε ένας τουλάχιστον παράνομος χαρακτήρας. Το *p* σου δείχνει τον πρώτο (`*p`).

Όλα καλά εκτός από την περίπτωση που έχουμε **υπερχείλιση** (overflow)! Τι γίνεται στην περίπτωση αυτή; Σου έρχεται ένα μήνυμα από την τιμή που επιστρέφει η συνάρτηση και την τιμή μιας καθολικής μεταβλητής με το όνομα "`errno`". Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cerrno>`".

- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μεγαλύτερη από `DBL_MAX` αυτή επιστρέφει τιμή `HUGE_VAL` (ορίζεται στο `cmath`) και βάζει στην *errno* τιμή `ERANGE` (ορίζεται στο `cerrno`).
- Αν από τη *strtod()* ζητηθεί να διαβάσει τιμή μικρότερη από `-DBL_MAX` αυτή επιστρέφει τιμή `-HUGE_VAL` και βάζει στην *errno* τιμή `ERANGE`.[4]

Αν λοιπόν μετά τις

```
errno = 0;
d = strtod( s2, &p );
```

ισχύει η συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

δεν έχει νόημα να χρησιμοποιήσεις την τιμή της *d*.

Παρομοίως, αν μετά τις

```
errno = 0;
l = strtol(s2, &p, 10)
```

η συνθήκη

```
(l == LONG_MAX || l == LONG_MIN) && errno == ERANGE
```

έχει τιμή `true` μην χρησιμοποιήσεις την τιμής *l*.

Για την *errno* θα τα ξαναπούμε.

Παρόμοιες με τη *strtol()* είναι και οι

```
long long int strtoll( const char* s, char** endptr, int base );
unsigned long int strtoul( const char* s, char** endptr,
                                                    int base );
unsigned long long int strtoull( const char* s, char** endptr,
                                                    int base );
```

ενώ οι *atol()* (§10.13.1), *atoi()*, *atoll()* είναι ισοδύναμες με

```
long int atol( const char* s )
{  return strtol( s, NULL, 10 );  }
int atoi( const char* s )
{  return int( strtol(s, NULL, 10) );  }
long long int atoll( const char* s )
{  return strtoll( s, NULL, 10 );  }
```

13.9 Πώς Επιλέγουμε το Είδος της Συνάρτησης

Μάθαμε λοιπόν για συναρτήσεις με τύπο και χωρίς τύπο, για παραμέτρους βέλη, τιμής και αναφοράς, για καθολικές μεταβλητές... Πώς τα χρησιμοποιούμε όλα αυτά;

Παρακάτω δίνουμε μερικούς κανόνες για το πώς επιλέγουμε το είδος της συνάρτησης. Θα πρέπει να τηρείς αυτούς τους κανόνες μέχρι να γίνεις μεγάλος(-η) προγραμματιστής(-στρια)· τότε θα ξέρεις πότε και πώς να τους παραβιάζεις.

Πριν από όλα λοιπόν θα βάλουμε τον εξής κανόνα:

♦ ***Όποτε μπορούμε να γράψουμε συνάρτηση με τύπο θα γράφουμε συνάρτηση με τύπο.***

[4] Σε περίπτωση υπερχείλισης η *strtof* επιστρέφει *HUGE_VALF* ή *–HUGE_VALF* ενώ η *strtold* θα δώσει *HUGE_VALL* ή *–HUGE_VALL*.

Αυτός ο κανόνας μπορεί να θεωρηθεί ως ειδική περίπτωση ενός γενικότερου κανόνα που λέει:

- ***Κάθε συνάρτηση θα πρέπει να έχει τον ελάχιστο καλά καθορισμένο υπολογιστικό στόχο***

Δηλαδή, όχι «αφού γράφουμε μια συνάρτηση να τα βάλουμε όλα μέσα!» Στον κανόνα αυτόν θα επανέλθουμε αργότερα με παραδείγματα.

1ος Κανόνας:

- ***Κάθε συνάρτηση επικοινωνεί –με τη συνάρτηση που την καλεί– μόνο με το όνομά της και με τις παραμέτρους της.***

Με άλλα λόγια: *μη γράφεις συναρτήσεις που να επικοινωνούν μέσω καθολικών μεταβλητών*.

Όταν πρόκειται να γράψεις μια συνάρτηση θα πρέπει, πριν από όλα, να καταγράψεις:

- τις τιμές που θα εισάγονται σε αυτήν (εισερχόμενες),
- τις τιμές που θα εξάγονται από αυτήν (εξερχόμενες),
- τις τιμές που θα μεταβάλλονται από αυτήν (διπλής κατεύθυνσης).

Με βάση αυτήν την καταγραφή αποφασίζεις το είδος της συνάρτησης που θα γράψεις:

2ος Κανόνας:

- ***Αν η συνάρτηση δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή τότε γράφεις συνάρτηση με τύπο.***

Στην περίπτωση αυτήν η εξερχόμενη τιμή θα αποδίδεται με την κλήση (όνομα) της συνάρτησης και τύπος της συνάρτησης είναι ο τύπος της εξερχόμενης τιμής. Είναι φανερό ότι οποιαδήποτε συνάρτηση με τύπο μπορείς να την υλοποιήσεις και ως συνάρτηση χωρίς τύπο με μια εξερχόμενη τιμή. Αλλά η συνάρτηση με τύπο είναι σαφώς πιο εύχρηστη και επομένως προτιμότερη.

3ος Κανόνας:

- ***Αν η συνάρτηση έχει μια ή περισσότερες μεταβαλλόμενες τιμές ή περισσότερες από μία εξερχόμενες τιμές τότε γράφεις συνάρτηση χωρίς τύπο*** `(void)`.

Στην περίπτωση αυτή οι μεταβαλλόμενες τιμές επιστρέφουν μέσω παραμέτρων. Ο κανόνας αυτός είναι συμπλήρωση του 2ου κανόνα· μια διαφορετική διατύπωση είναι η εξής:

- ***Μη γράφεις συνάρτηση με τύπο με παραμέτρους αναφοράς που να υλοποιούν επικοινωνία για μεταβαλλόμενες τιμές ή εξερχόμενες τιμές.***

13.9.1 Περί Παραμέτρων

Ας έλθουμε τώρα στις παραμέτρους. Άλλες γλώσσες προγραμματισμού επιτρέπουν προσδιορισμό των παραμέτρων σχετικό με τη χρήση τους. Η Ada, για παράδειγμα, έχει παραμέτρους, όπως τις δώσαμε πιο πάνω:

- **in** για εισαγωγή δεδομένων στη συνάρτηση,
- **out** για εξαγωγή τιμών από τη συνάρτηση και
- **in out** για τιμές που μεταβάλλονται από τη συνάρτηση.

Αξίζει να επισημάνουμε εδώ ότι η Ada στα υποπρογράμματα που αντιστοιχούν στις συναρτήσεις με τύπο της C++ επιτρέπει παραμέτρους **in** μόνον![5]

[5] Σου θυμίζει αυτό κάτι από τους κανόνες μας;...

Η C έχει ένα είδος παραμέτρων: *παραμέτρους τιμής*. Με αυτές περνάς δεδομένα προς τη συνάρτηση. Αν τώρα θέλεις να πάρεις τιμές από τη συνάρτηση μέσω παραμέτρων εισάγεις προς αυτήν ένα βέλος που δείχνει τη διεύθυνση όπου περιμένεις το «εξαγόμενο».

Η C++ έκανε ένα βήμα σε σχέση με τη C αλλά όχι μεγάλο: μετονόμασε το βέλος σε *αναφορά* και μας έδωσε την ευκολία να το χειριζόμαστε χωρίς να κάνουμε αποπαραπομπή, που γίνεται αυτομάτως. Παρ' όλα αυτά ο προσδιορισμός των παραμέτρων έχει σχέση με την υλοποίηση:

1. **Παράμετροι τιμής** (value parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση χωρίς να επιστρέφουν τις οποιεσδήποτε αλλαγές στη συνάρτηση που καλεί. Με αυτές υλοποιούμε παραμέτρους για εισερχόμενες τιμές (**in**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο τιμής μπορεί να είναι γενικά μια παράσταση.
2. **Παράμετροι αναφοράς** (reference parameters) που εισάγουν τιμές προς την καλούμενη συνάρτηση και γνωστοποιούν στη συνάρτηση που καλεί τις όποιες αλλαγές έγιναν στην καλούμενη συνάρτηση. Με αυτές υλοποιούμε παραμέτρους για εξερχόμενες (**out**) και μεταβαλλόμενες τιμές (**in out**). Μια πραγματική παράμετρος που αντιστοιχεί σε τυπική παράμετρο αναφοράς μπορεί να είναι μεταβλητή ή στοιχείο πίνακα ή πίνακας (αν η τυπική παράμετρος είναι πίνακας).
3. **Παράμετροι-πίνακες** που είναι μεν παράμετροι τιμής αλλά είναι παράμετροι-βέλη. Επομένως, όπως έχουμε δει στην §9.4, είναι παράμετροι **in out**. Τις βάζουμε χωριστά διότι μπορείς να τις χειρίζεσαι μέσα στη συνάρτηση ως πίνακες χωρίς αποπαραπομπές. Αν θέλεις να έχεις έναν πίνακα ως παράμετρο **in** προτάσσεις τον περιορισμό `const`.

Προς το παρόν τα μόνα «μεγάλα» αντικείμενα που έχουμε δει είναι οι πίνακες και ξέρουμε πώς να τους χειριστούμε. Στη συνέχεια, όταν μάθουμε για κλάσεις, θα δούμε και άλλα. Αν θέλουμε να περάσουμε ένα μεγάλο αντικείμενο (ας πούμε τύπου *T*) ως παράμετρο **in**, η παράμετρος τιμής δεν είναι καλή λύση· η διαδικασία της αντιγραφής καθυστερεί την εκτέλεση του προγράμματος και φορτώνει μια περιοχή της μνήμης (τη στοίβα) που είναι μάλλον περιορισμένη. Στην περίπτωση αυτήν προτιμούμε να χρησιμοποιήσουμε παράμετρο αναφοράς (*T*`&`) με τον περιορισμό `const` (`const` *T*`&`).

Οι προγραμματιστές που έγραφαν παλιότερα C προτιμούν να χρησιμοποιούν αντί για παραμέτρους αναφοράς **παραμέτρους βέλη** (pointers). Ένας λόγος για την προτίμησή τους είναι ότι όταν διαβάζει κάποιος μια *κλήση συνάρτησης* καταλαβαίνει αμέσως ποιες παράμετροι είναι εισερχόμενες και ποιες διπλής κατεύθυνσης.

13.9.2 Παράμετρος – Ρεύμα

Είπαμε πριν (§11.6) ότι οι εντολές εισόδου/εξόδου είναι εντολές-παραστάσεις (πράξεις). Ας προσπαθήσουμε τώρα να το καταλάβουμε.

Κατ' αρχάς να υπενθυμίσουμε ότι, όπως μάθαμε στο Μέρος Α (§8.2), ένα ρεύμα έχει έναν **ενταμιευτή** (buffer), δηλαδή μια περιοχή μνήμης, όπου γίνεται ενδιάμεση αποθήκευση των δεδομένων του αρχείου που διαβάζουμε (ή γράφουμε). Φυσικά, χρειαζόμαστε και έναν δείκτη που δείχνει σε ποια θέση του ενταμιευτή διαβάζουμε, άλλους δείκτες που μας λένε ποια περιοχή του αρχείου έχουμε στον ενταμιευτή κλπ. Κάθε πράξη εισόδου/εξόδου μεταβάλλει την κατάσταση του ενταμιευτή και ολόκληρου του ρεύματος και η επόμενη πράξη εισόδου/εξόδου γίνεται πάνω σε αυτήν τη νέα κατάσταση.

Δες πώς αντανακλώνται αυτά τα πράγματα στο πρόγραμμα με ένα παράδειγμα σε γνωστά πράγματα. Ας πούμε ότι έχουμε την εντολή (δήλωσε τις *x*, *r* όπως θέλεις και δώσε όποιες τιμές θέλεις):

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

Δοκιμάζουμε να τη γράψουμε με διαφορετικό τρόπο:

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

Βλέπουμε ότι γίνεται δεκτή από τον μεταγλωττιστή και βγάζει το ίδιο αποτέλεσμα με την αρχική.

Για να δούμε όμως τι γίνεται εδώ: Ξέρουμε ότι η "`cout << " x = "`" σημαίνει «*στείλε στο ρεύμα cout την τιμή της παράστασης* `" x = "`.» Ωραία! Τότε όμως τι σημαίνει η:

`(cout << " x = ") << x`

Στείλε στο ρεύμα `(cout << " x = ")` *την τιμή της x;*

Ναι, ακριβώς αυτό σημαίνει. *Η "*`cout << " x = "`*" είναι μια πράξη που αποτέλεσμά της είναι το ρεύμα* `cout`.

Όπως καταλαβαίνεις αυτό είναι απαραίτητο για να μπορούμε να γράφουμε σε μια εντολή εξόδου πολλές εξερχόμενες τιμές (παραστάσεις).

Και κάτι ακόμη: Το «*Στείλε στο ρεύμα* `(cout << " x = ")` *την τιμή της x*» πρέπει να διατυπωθεί ακριβέστερα: *Στείλε στο ρεύμα "*`cout << " x = "`*" όπως έχει γίνει μετά την έξοδο της* `" x = "` *, την τιμή της x.*

Παίρνοντας υπόψη μας τα παραπάνω ας σκεφτούμε την εξής περίπτωση: Έχουμε δηλώσει στη **main**

```
ofstream tout( "afile.txt" );
```

και στη συνέχεια καλούμε μια συνάρτηση με παράμετρο *tout*:

```
afunction( tout, . . . );
```

Αν η *tout* περάσει ως παράμετρος τιμής, στο εσωτερικό της συνάρτησης θα δουλεύουμε με ένα αντίγραφο του ρεύματος ενώ η κατάσταση του *tout* της **main** δεν μεταβάλλεται. Όταν ο έλεγχος της εκτέλεσης επιστρέψει στη **main** το *tout* δεν θα μάθει αυτά που έγιναν στην *afunction*!

Επομένως; Η λύση είναι μία:

- ***Μια παράμετρος συνάρτησης που είναι ρεύμα προς/από αρχείο (ή συσκευή) θα πρέπει να είναι υποχρεωτικώς παράμετρος* in out *(αναφοράς).***

  Στην περίπτωσή μας θα πρέπει να έχουμε:

```
void afunction( ofstream& tout, . . . )  { . . . }
```

  Και κάτι ακόμη:

- Αν θέλεις να καλείς τη συνάρτησή σου ώστε να γράφει είτε σε αρχείο είτε στο *cout* βάζε παράμετρο τύπου `ostream` και όχι `ofstream`.
- Παρομοίως, αν θέλεις να περνάς σε μια συνάρτηση είτε το *cin* είτε κάποιο ρεύμα κλάσης `ifstream` βάζε παράμετρο κλάσης *istream*.

Γιατί; Ξαναδιάβασε την §8.1 και πάρε υπόψη σου ότι –όπως θα μάθουμε αργότερα– όπου μπορεί να εμφανιστεί αντικείμενο μιας κλάσης μπορεί να εμφανιστεί και αντικείμενο των κληρονόμων της.

Με βάση αυτά που είπαμε, μια συνάρτηση για είσοδο ή έξοδο στοιχείων έχει πάντοτε μια παράμετρο για μεταβαλλόμενη τιμή: το ρεύμα. Θα πρέπει λοιπόν να είναι συνάρτηση `void`.

13.9.3 Παραδείγματα

Ας δούμε μερικά παραδείγματα εφαρμογής των παραπάνω κανόνων ξεκινώντας από αυτά της προηγούμενης παραγράφου.

Παράδειγμα 1

Θέλουμε μια συνάρτηση *minmax* που θα τροφοδοτείται με δύο ακέραιους *x*, *y* και θα μας επιστρέφει τον ελάχιστο και τον μέγιστο από τους δύο.

Η συνάρτησή μας:

- θα παίρνει (**in**) δύο τιμές, τις *x*, *y* και

- θα επιστρέφει (**out**) δύο τιμές, τις *mnxy*, *mxxy*.

Αφού η συνάρτησή μας έχει δύο εξερχόμενες τιμές, σύμφωνα με τον Κανόνα 3 θα πρέπει να είναι συνάρτηση χωρίς τύπο.

Τι παραμέτρους θα έχει;

- δύο παραμέτρους τιμής, τις *x*, *y* και
- δύο παραμέτρους αναφοράς, τις *mnxy*, *mxxy*.

Να λοιπόν η συνάρτηση της:

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

Ας πούμε τώρα ότι δεν μας ζητούν να γράψουμε τη συγκεκριμένη συνάρτηση αλλά να επιλέξουμε τι είδους συνάρτηση ή συναρτήσεις θα γραφούν.

Πρέπει να επιλέξουμε μεταξύ της *minmax()* και των:

```
int min( int x, int y )
{
   int fv( x );
   if ( x > y )  fv = y;
   return fv;
} // min
int max( int x, int y )
{
   int fv( x );
   if ( y > x )  fv = y;
   return fv;
} // max
```

Η *minmax()* προφανώς παραβιάζει τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά αν το πρόγραμμά μας χρειάζεται να υπολογιζει κατ' επανάληψη και το ελάχιστο και το μέγιστο τότε κάθε φορά έχουμε:

- Η *minmax()* θα καλείται ως εξής:

  ```
  minmax( a, b, mnab, mxab );
  ```

  και θα έχουμε:
  - μια κλήση συνάρτησης,
  - μια σύγκριση,
  - δύο εκχωρήσεις.
- Οι *min()* και *max()* θα καλούνται ως εξής:

  ```
  mnab = min( a, b );  mxab = max( a, b );
  ```

  και θα έχουμε:
  - δύο κλήσεις συναρτήσεων,
  - δύο συγκρίσεις,
  - δύο δηλώσεις μεταβλητών με αρχική τιμή (ορισμούς),
  - μια εκχώρηση,
  - δύο επιστροφές τιμών.

Προφανώς, για την περίπτωση αυτή, η *minmax()* είναι προτιμότερη.

Παράδειγμα 2

Θέλουμε μια συνάρτηση (με όνομα *swap*) που θα παίρνει δύο μεταβλητές τύπου `int` και θα αντιμεταθέτει τις τιμές τους.

Στην περίπτωση αυτή θέλουμε μια συνάρτηση που θα *μεταβάλλει* τις τιμές δύο μεταβλητών (**in out**). Κατά τον 3ο Κανόνα θα γράψουμε μια συνάρτηση χωρίς τύπο.

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

Παράδειγμα 3

Θέλουμε μια συνάρτηση, με όνομα *cntInt()*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου **double** και το πλήθος *n* των στοιχείων του και θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή.

Η *cntInt()*:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και
- με «*το πλήθος n των στοιχείων του*»,
- «*θα επιστρέφει το πλήθος των στοιχείων του a που έχουν ακέραιη τιμή*».

Δηλαδή, τροφοδοτείται με έναν πίνακα –βέλος και πλήθος στοιχείων– (**in**) και επιστρέφει μία ακέραιη τιμή.

Αφού η συνάρτησή μας «*δεν έχει μεταβαλλόμενες τιμές και έχει μια μόνον εξερχόμενη τιμή*» θα γράψουμε *συνάρτηση με τύπο*. Γράφουμε λοιπόν:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
```

Η εξερχόμενη τιμή θα επιστρέφει μέσω του ονόματος της συνάρτησης στην κλήση της.

Ολόκληρη η συνάρτηση[6]:

```
int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
{
   int m( 0 );

   for ( int k(0); k < n; ++k )
      if (a[k] == static_cast<long int>(a[k]))  ++m;
   return m;
} // cntInt
```

Παράδειγμα 4

Θέλουμε μια συνάρτηση, με όνομα *to1Dgt*, που θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου **double**, και το πλήθος *n* των στοιχείων του και θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους.

Η συνάρτηση:

- «*θα τροφοδοτείται με έναν μονοδιάστατο πίνακα a με στοιχεία τύπου* **double**» και «*θα αποκόπτει από τις τιμές όλων των στοιχείων του όλα τα ψηφία μετά το 1ο ψηφίο του κλασματικού μέρους*»· δηλαδή οι τιμές των στοιχείων του πίνακα θα *μεταβάλλονται* από τη συνάρτηση.

  Ακόμη, η συνάρτηση

- «[*θα τροφοδοτείται*] *με το πλήθος n των στοιχείων του* [*πίνακα*]». Η τιμή της *n* είναι εισερχόμενη στη συνάρτηση.

Αφού η συνάρτησή μας «*έχει μεταβαλλόμενες τιμές*» (πίνακας a) θα πρέπει να γράψουμε συνάρτηση χωρίς τύπο (**void**). Γράφουμε λοιπόν:

```
void to1Dgt( double a[],     // μεταβαλλόμενη παράμετρος
```

[6] Προφανώς, ο έλεγχος `a[k] == static_cast<long int>(a[k])` για ακέραιη τιμή δεν δουλεύει όταν η a[*k*] έχει πολύ μεγάλη τιμή.

```
                int n )          // εισερχόμενη παράμετρος
```

Τα στοιχεία του a έχουν τις μεταβαλλόμενες τιμές.

Ολόκληρη η συνάρτηση:

```
void to1Dgt( double a[], int n )
{
   for (int k(0); k < n; ++k )
      a[k] = (static_cast<long int>(a[k]*10))/10.0;
} // to1Dgt
```

&

Παράδειγμα 5

Στο Παράδ. 2 της §7.7 γράψαμε δύο συναρτήσεις την *gcd()* –που υπολογίζει τον ΜΚΔ δύο ακεραίων– και την *lcm()* που υπολογίζει το ΕΚΠ. Η *lcm()* καλεί την *gcd()*.

Αν έχουμε να επιλέξουμε ποιες και τι είδους συναρτήσεις θα γράψουμε:

- Σε πλήρη συμφωνία με όλους τους κανόνες μας θα γράψουμε την

```
unsigned int gcd( int x, int y )
```

- Η *lcm()* που γράψαμε μπορεί να γραφεί ως εξής:

```
void gcdLcm( int x, int y, int& gcdP, int& lcmP )
{                                 // ΕΚΠ(x,y)*ΜΚΔ(x,y) = x*y
   if ( (x == 0 && y == 0) || x < 0 || y < 0 )
   {
      cout << " η gcdLcm κλήθηκε με "
           << x << ", " << y << endl;
      exit( EXIT_FAILURE );
   }
// (x != 0 || y != 0) && x >= 0 && y >= 0
   gcdP = gcd(x, y);
   lcmP = x * y / gcdP;
} // gcdLcm
```

Εδώ, προφανώς, παραβιάζουμε τον κανόνα του ελάχιστου υπολογιστικού στόχου. Αλλά δεν κάνουμε τους ίδιους υπολογισμούς δύο φορές.

Πρόσεξε ότι ο ΜΚΔ υπολογίζεται σε ένα μέρος μόνον: στη *gcd()*. Για βελτιώσεις ή προβλήματα που μπορεί να παρουσιασθούν παρεμβαίνουμε μόνον εκεί.

- Και την *lcm()*, θα την πετάξουμε; Ε, μη την πετάξεις. Αν έχεις κάποια εφαρμογή που θέλει μόνον το ΕΚΠ χρησιμοποίησέ την.

Μια παρόμοια περίπτωση είναι και αυτή του Παραδ. 2 της §9.4: Οι συναρτήσεις

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

υπολογίζουν τη μέση τιμή και την τυπική απόκλιση των `x[`*from*`]..x[`*upto*`]` αντιστοίχως. Η *stdDev()* καλεί τη *vectorAvrg()*.

Εκεί όμως έχουμε και κάτι άλλο: μπορούμε να γράψουμε μια:

```
void simpleStat ( const double x[], int n, int from, int upto,
                  double& avrgP, double& stdDevP )
```

που υπολογίζει και τα δύο μεγέθη με πιο αποδοτικό τρόπο.

Το τι γράφουμε, τι όχι και πότε τα χρησιμοποιούμε τα αφήνουμε ως άσκηση για σένα.

&

Παράδειγμα 6

Στο Παράδ. 4 της §12.4 (πολλαπλασιασμός πινάκων) γράφουμε δυο φορές τις ίδιες εντολές για να διαβάσουμε τους δυο πίνακες και τρεις φορές τις ίδιες εντολές για να γράψουμε τους τρεις πίνακες. Δεν θα μπορούσαμε να ευκολύνουμε τη ζωή μας με δύο συναρτήσεις; Φυσικά!

Κατ' αρχάς, αφού οι δύο συναρτήσεις κάνουν είσοδο/έξοδο στοιχείων θα πρέπει να είναι `void`:

```
void input2DAr( istream& tin, . . .
```

```
void output2DAr( ostream& tout, . . .
```

Και οι δύο συναρτήσεις έχουν ως παράμετρο τον πίνακα: η πρώτη ως παράμετρο **out** και η δεύτερη ως παράμετρο **in**. Για τη δεύτερη τα πράγματα είναι απλά:

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

Για την πρώτη πώς θα είναι οι παράμετροι; Έτσι:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
```

Οι διαστάσεις των πινάκων δεν διαβάζονται από το αρχείο· είναι γνωστές στο πρόγραμμα που καλεί τη συνάρτηση γι' αυτό και δεν βάζουμε "`int& nRow, int& nCol`". Αργότερα θα δούμε και περιπτώσεις όπου τα πάντα είναι **out**.

Ολόκληρη η *input2DAr*:

```
void input2DAr( istream& tin, int* a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
         tin >> a[r*nCol+c];
//          tin >> a[r][c];
   }
} // input2DAr
```

Για τούς πίνακες:

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

την καλούμε ως εξής:

```
ifstream atx( "arr.txt" );
input2DAr( atx, &a[0][0], l, m );
input2DAr( atx, &b[0][0], m, n );
atx.close();
```

Αφήνουμε ως άσκηση την *output2DAr*.

Παρατήρηση: ▶

Ωραίες οι συνταγές, αλλά εδώ βλέπουμε συναρτήσεις της πάγιας βιβλιοθήκης της γλώσσας που δεν συμμορφώνονται με αυτές.

- Στην προηγούμενη παράγραφο οι *strtod()* και *strtol()* επιστρέφουν τιμή (`double` ή `long int` αντιστοίχως) αλλά βγάζουν και ένα βέλος μέσω της παραμέτρου *endptr*.
- Στο προηγούμενο κεφάλαιο προσπαθώντας να μιμηθούμε τη λειτουργία των *strlen()* και *strcat()* είδαμε ότι η δεύτερη:
  - αλλάζει την πρώτη παράμετρο,
  - επιστρέφει και τιμή-βέλος.

Τι συμβαίνει με αυτές τις παρανομίες;

Οι απαντήσεις στα ερωτήματα αυτά είναι οι εξής:

- Οι συναρτήσεις αυτές έχουν σχεδιασθεί πολύ παλιά (έρχονται από τη C). Ο τρόπος που σκέφτονταν τότε οι προγραμματιστές ήταν αρκετά διαφορετικός και το βασικό κριτήριο για να πάρουν τις αποφάσεις τους είχε να κάνει με τους περιορισμούς που έβαζαν οι ΗΥ εκείνου του καιρού. Τώρα τα πράγματα είναι αρκετά διαφορετικά.
- Και εσυ, στο μέλλον, πιθανότατα, θα παραβείς αυτούς τους κανόνες. Αλλά να το κάνεις όταν θα έχεις γίνει προγραμματιστής(-στρια) σαν τον D.M. Ritchie ή τον B.W. Kernighan, γιατί τότε θα ξέρεις πότε και πώς να τους παραβείς. Μέχρι τότε, το καλύτερο που έχεις να κάνεις είναι να τους τηρείς.◀

13.10 Υποδείγματα Συναρτήσεων

Έστω ότι θέλουμε να γράψουμε ένα πρόγραμμα που i) θα διαβάζει από το πληκτρολόγιο τις τιμές των στοιχείων ενός μονοδιάστατου πίνακα (με 5 στοιχεία), ii) θα βρίσκει το πλήθος των στοιχείων του με ακέραιη τιμή, iii) θα αποκόπτει τις τιμές τους στο 1ο δεκαδικό ψηφίο και iv) θα τις γράφει στην οθόνη.

Μπορούμε να το γράψουμε χρησιμοποιώντας τις συναρτήσεις που είδαμε στα παραδ. 3 και 4 της προηγούμενης παραγράφου:

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

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main
```

Οι συναρτήσεις έχουν γραφεί έτσι ώστε να μην προκαλούν προβλήματα κατά τη μεταγλώττιση. Δηλαδή, γράψαμε τελευταία τη **main** που έχει τις αναφορές στις δύο συναρτήσεις, ώστε όταν ο μεταγλωττιστής βρει τα ονόματα των συναρτήσεων να τις ξέρει ήδη.

Όταν γράφεις ένα μεγάλο πρόγραμμα κάτι τέτοιο δεν είναι πάντοτε εφικτό ούτε βολικό. Πολύ συχνά το πρόγραμμά μας χρησιμοποιεί συναρτήσεις που δεν γράφονται μαζί με αυτό αλλά υπάρχουν σε διάφορες βιβλιοθήκες και συνδέονται στο πρόγραμμα μετά τη μεταγλώττισή του. Όπως λέγαμε και στην §7.4, η C++ μας επιτρέπει να γράφουμε τις συναρτήσεις σε οποιαδήποτε σειρά (ή και να μην τις γράφουμε καθόλου) αρκεί να βάζουμε πιο πριν **υποδείγματα** (prototypes) των συναρτήσεων.

Το υπόδειγμα μιας συνάρτησης περιέχει τον τύπο του αποτελέσματος, το όνομα της συνάρτησης και –μέσα σε παρενθέσεις– τους τύπους των τυπικών παραμέτρων. Πάντως μπορείς να χρησιμοποιήσεις και την επικεφαλίδα της τερματιζόμενη με ένα ";". Για τις συναρτήσεις του παραπάνω παραδείγματος θα μπορούσαμε να έχουμε:

```
   int cntInt( const double[], int );
   void to1Dgt( double[], int );
```

ή

```
   int cntInt( const double*, int );
   void to1Dgt( double*, int );
```

ή

```
   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );
```

Έτσι, το πρόγραμμά μας θα μπορούσε να γραφεί ως εξής:

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

   int cntInt( const double a[], int n );
   void to1Dgt( double a[], int n );

int main()
{
   double q[5];

   for ( int k(0); k < 5; ++k )  cin >> q[k];
   cout << cntInt( q, 5 ) << endl;
   to1Dgt( q, 5 );
   for ( int k(0); k < 5; ++k )  cout << q[k] << "  ";
   cout << endl;
} // main

   int cntInt( const double a[], int n )//εισερχόμενες παράμετροι
   {
      int m( 0 );

      for ( int k(0); k < n; ++k )
         if (a[k] == static_cast<long int>(a[k]))  ++m;
      return m;
   } // cntInt

   void to1Dgt( double a[], int n )
   {
      for (int k(0); k < n; ++k )
         a[k] = (static_cast<long int>(a[k]*10))/10.0;
   } // to1Dgt
```

Πολύ συχνά, κυρίως όταν έχουμε μεγάλα προγράμματα, οι επικεφαλίδες μαζί με διάφορες δηλώσεις μπαίνουν σε ένα ξεχωριστό αρχείο που περιλαμβάνεται (**#include**) στην αρχή του αρχείου του προγράμματος. Π.χ. αν το πρόγραμμά μας βρίσκεται στο αρχείο **truncarr.cpp** θα βάλουμε, κατά τη συνήθεια που επικρατεί, τις επικεφαλίδες σε ένα αρχείο με όνομα: **truncarr.h** και η αρχή του προγράμματός μας θα είναι:

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

#include "truncarr.h"

using namespace std;

int main()
   . . .
```

13.11 Ένα Δύσκολο Πρόβλημα!

Και τώρα θα γράψουμε πρόγραμμα! Μάθαμε αρκετά ώστε να μπορούμε να αντιμετωπίσουμε ένα πρόβλημα που θα μπορούσε να υπάρχει στον πραγματικό κόσμο.

Ένα συνεργείο έκανε μέτρηση ροής οχημάτων σε κάποιο δρόμο. Σε αρχείο, text – με το όνομα στο δίσκο **autoflow.txt***– καταγράφηκαν οι τιμές ροής σε οχήματα/min κάθε λεπτό. Το πλήθος των τιμών στο αρχείο είναι άγνωστο, αλλά σίγουρα θετικό.*

Οι πρώτες τρεις γραμμές του αρχείου έχουν την «ταυτότητα» της μέτρησης ως εξής:

```
Υπεύθυνος:\t<όνομα>\t<επώνυμο>
Σημείο Μετρήσεων:\t<οδός-αριθμός>\t<περιοχή>\t<δήμος>
Αρχή:\tdd.mm.yyyy\thh:mm
```

Στις υπόλοιπες γραμμές δίνονται οι τιμές από τη μέτρηση, μια σε κάθε γραμμή. Κάθε τιμή είναι γραμμένη στις πέντε πρώτες θέσεις. Πέρα από τη δέκατη θέση μπορεί να υπάρχουν σχόλια που περιγράφουν συμβάντα κατά τη διάρκεια της μέτρησης. Να ένα παράδειγμα:

```
Υπεύθυνος:   Ανδρέας      Νικολόπουλος
Σημείο Μετρήσεων: Τζαβέλλα 48    Νεάπολις     Αθήνα
Αρχή:  15.06.2009   05:00
   26
   30
    8
   28
. . .
   17
   19
    4
   12     / Αρχή λειτουργίας σηματοδότησης
   28
   17
. . .
   17
   31     / Βλάβη σηματοδότη Τζαβέλλα & Γιωργάκου
   23
   24
. . .
```

Να γραφεί πρόγραμμα που θα διαβάζει το αρχείο και θα υπολογίζει:

1. Το πλήθος τιμών του αρχείου καθώς και τη διάρκεια των μετρήσεων σε h και min.

2. Το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων.

3. Τη μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min.

Όλα αυτά θα γράφονται στην τελική έκθεση, που θα γραφεί σε αρχείο με το όνομα `report.txt`*. Στις πρώτες τρεις γραμμές του αρχείου θα αντιγραφούν οι τρεις πρώτες γραμμές του* `autoflow.txt`*.*

Ένα από τα ζητούμενα της δουλειάς είναι και η μελέτη της μεταβολής της ροής οχημάτων. Ως πρώτο βήμα μας ζητείται να δημιουργήσουμε ένα άλλο αρχείο με τις μεταβολές ροής ανά min.

Σαν πολλά δε ζητάει; Μπα, τα περισσότερα τα ξέρουμε! Ας ξεκινήσουμε με τα αρχεία. Οι προδιαγραφές λένε ότι θα διαβάζουμε ένα αρχείο και θα γράφουμε δύο. Όλα τα αρχεία θα είναι text.

Το πρώτο που θα κάνουμε είναι να διαμορφώσουμε τις προδιαγραφές. Αν $numberPerMin_k$, k: $1..n$ είναι η ακολουθία τιμών που διαβάζουμε από το `autoflow.txt` τότε στο ένα από τα αρχεία θα γράψουμε τις τιμές μιας νέας ακολουθίας, της

$$difference_k = numberPerMin_k - numberPerMin_{k-1},\ k: 2..n$$

«*Το πλήθος τιμών του αρχείου*» είναι το *n*. Αφού οι μετρήσεις λαμβάνονται ανά *min*, η «*διάρκεια των μετρήσεων*» σε *min* είναι *n*. Για να τη μετατρέψουμε «*σε h και min*» παίρνουμε το πηλίκο και το υπόλοιπο της *n* δια 60:

$$hours = n \ / \ 60, \quad minutes = n \ \% \ 60$$

Αφού κάθε $numberPerMin_k$ που διαβάζουμε είναι το πλήθος των αυτοκινήτων που περνούν σε ένα *min*, «*το πλήθος οχημάτων που μετρήθηκαν σε ολόκληρη τη διάρκεια των μετρήσεων*» είναι προφανώς το άθροισμα των τιμών που διαβάζονται:

$$numberOfCars = \sum_{k=1}^{n} numberPerMin_k .$$

Η «*μέση τιμή της ροής οχημάτων κατά τη διάρκεια των μετρήσεων, σε οχήματα/min*» είναι η

$$averageFlow = \sum_{k=1}^{n} numberPerMin_k / n$$

Αυτή μπορεί να υπολογισθεί μόνον αν $n > 0$, αν δηλαδή το αρχείο που μας δίνεται έχει μια τουλάχιστον μέτρηση.

Να λοιπόν οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Απαίτηση: Στο αρχείο `differences.txt` γράφονται τα μέλη της ακολουθίας $difference_k$, k: `2..`n

Στο αρχείο `report.txt` γράφονται:

- οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των *n*, *hours* (`==` n `/ 60`), *minutes* (`==` n `%` 60), *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (`==` *numberOfCars* / n).

Ας κάνουμε τώρα ένα σχέδιο για το πρόγραμμά μας.

Στην αρχή θα πρέπει να ανοιχθούν οπωσδήποτε τα αρχεία `autoflow.txt`, για διάβασμα και το `report.txt` για να γράψουμε τις τρεις πρώτες γραμμές. Στο `differences.txt` θα γράψουμε αν από το `autoflow.txt` διαβάσουμε τουλάχιστον δύο τιμές ροής. Θα πρέπει λοιπόν να περιμένουμε και να το ανοίξουμε μόνον αν βρούμε δεύτερη τιμή; Όχι! Αφού μας ζητείται αρχείο διαφορών θα το δημιουργήσουμε ακόμη και αν το αφήσουμε κενό.

Στη συνέχεια θα αντιγράψουμε στο `report.txt` τις τρεις πρώτες γραμμές.

Μετά θα διαβάζουμε από το `autoflow.txt`, θα υπολογίζουμε και θα γράφουμε στο `differences.txt` μέχρι να τελειώσει το αρχείο. Αλλά εδώ χρειάζεται προσοχή: για την πρώτη τιμή δεν μπορούμε να υπολογίσουμε διαφορά· επομένως χρειάζεται ειδικό χειρισμό.

Όσο διαβάζουμε και υπολογίζουμε και γράφουμε διαφορές θα πρέπει να υπολογίζουμε το *n* (να μετρούμε) και το *numberOfCars* (να αθροίζουμε) ώστε να υπολογίσουμε και τη *averageFlow*.

Τέλος, υπολογίζουμε τη μέση τιμή, γράφουμε όσα χρειάζεται στο `report.txt` και κλείνουμε τα αρχεία.

Να λοιπόν ένα πρώτο σχέδιο του προγράμματός μας:

```
Άνοιξε τα ρεύματα των αρχείων
Επεξεργασία
Κλείσε τα ρεύματα
```

Αλλά, σαν να ξεχάσαμε κάτι εδώ: Θα προχωρήσουμε στις επεξεργασίες αν ανοίξουν τα ρεύματα, αλλιώς τι να επεξεργαστούμε; Το σχέδιο πρέπει να αλλάξει λίγο:

```
Άνοιξε τα ρεύματα των αρχείων
if ( άνοιξαν τα ρεύματα )
{
   Επεξεργασία
   Κλείσε τα ρεύματα
}
```

Στην §0.5 λέγαμε ότι «*το αρχικό πρόβλημα διασπάται σε δύο ή περισσότερα υποπροβλήματα*» από τα οποία το καθένα «*έχει προϋπόθεση την απαίτηση του προηγούμενου*». Ας βάλουμε λοιπόν και εδώ τις προδιαγραφές στα προβλήματα που προκύπτουν από τη διάσπαση:

Προϋπόθεση: $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

```
Άνοιξε τα ρεύματα των αρχείων
```

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση προηγούμενης και προϋπόθεση επόμενης: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τρεις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής n και το πλήθος των οχημάτων *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$).

Υπολογίστηκαν τα *hours* (== n / 60), *minutes* (== n % 60).

Αν $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση) υπολογίστηκε η *averageFlow* (== *numberOfCars* / n).

Γράφηκαν στο `report.txt` τα n, *hours*, *minutes*, *numberOfCars*, *averageFlow*.

`Κλείσε τα ρεύματα`

Απαίτηση: Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n

Στο αρχείο `report.txt` έχουν γραφεί:

- οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,
- οι τιμές των n, *hours* (== n / 60), *minutes* (== n % 60), *numberOfCars* (== $\sum_{k=1}^{n} numberPerMin_k$), *averageFlow* (== *numberOfCars* / n).

Κάπως έτσι γίνεται η ανάλυση των προδιαγραφών μέσα στο σχέδιό μας. Πρόσεξε το "$n > 0$" που μεταφέρεται από βήμα σε βήμα μέχρι το «`Τελικοί υπολογισμοί`» όπου και χρειάζεται για να μπορέσουμε να υπολογίσουμε τη μέση ροή.

13.11.1 «Άνοιξε τα ρεύματα των αρχείων»

Ξεκινούμε από την "`Άνοιξε τα ρεύματα των αρχείων`" και το πρώτο που πρέπει να δούμε είναι το τι μπορεί να συμβεί και να μην ανοίξουμε τα ρεύματα. Θα πεις «Για ποια ρεύματα συζητούμε; Το μόνο πρόβλημα που μπορεί να έχουμε είναι: το ρεύμα από το `autoflow.txt`, να μη βρει το αρχείο!». Ας δούμε μιαν άλλη λεπτομέρεια. Μέχρι τώρα ξέρουμε ότι αν απόπειραθούμε να δημιουργήσουμε ένα αρχείο που ήδη υπάρχει στον δίσκο, αυτομάτως σβήνεται το παλιό και δημιουργείται το καινούριο. Ε, λοιπόν: δεν θέλουμε –το πρόγραμμα που θα γράψουμε– να σβήσει, χωρίς προειδοποίηση, κάποιο αρχείο που ήδη υπάρχει με όνομα `report.txt` ή `differences.txt`.

Θα ανοίξουμε αυτά τα αρχεία με την παρακάτω συνάρτηση:

```
void openWrNoReplace( ofstream& newStream, string flNm, bool& ok )
{
   ifstream test( flNm.c_str() );           // άνοιξε για διάβασμα
   if ( test.fail() )                 // ok, δεν υπάρχει το αρχείο
   {
      newStream.open( flNm.c_str() );        // άνοιξε για γράψιμο
      ok = !newStream.fail();
   }
   else // υπάρχει το αρχείο, κλείσε το και μην το πειράξεις
   {
      test.close();
      ok = false;
   }
// ok = (δεν υπήρχε το αρχείο) && (άνοιξε για γράψιμο)
} // openWrNoReplace
```

Ας πούμε ότι έχουμε δηλώσει:

```
ofstream report;
bool     ok;
```

και στη συνέχεια το ανοίγουμε ως εξής:

```
openWrNoReplace( "report.txt", report, bool& ok );
```

Τι θα γίνει; Η συνάρτηση θα προσπαθήσει να δημιουργήσει ένα τοπικό ρεύμα (*test*) *για διάβασμα* από το αρχείο `report.txt`.

- Αν το *test* δημιουργηθεί τότε το κλείνουμε και δεν το πειράζουμε.
- Αποτυχία της απόπειρας (`test.fail()`) σημαίνει ότι το αρχείο δεν υπάρχει και έτσι προσπαθούμε να το ανοίξουμε για γράψιμο.

Με αυτόν τον τρόπο θα ανοίξουμε τα δύο εξερχόμενα ρεύματα.

Τώρα μπορούμε να προχωρήσουμε στο επόμενο επίπεδο ανάλυσης. Η "`Άνοιξε τα ρεύματα των αρχείων`" αναλύεται ως εξής:

```
Άνοιξε το autoflow για διάβασμα
if (δεν άνοιξε το autoflow)
{
   Μην προχωρείς
}
else    // το autoflow ανοικτό
{
   Άνοιξε το differences για γράψιμο
   if (δεν άνοιξε το differences)
   {
      κλείσε το autoflow
      Μην προχωρείς
   }
   else    // τα autoflow, differences ανοικτά
   {
      Άνοιξε το report για γράψιμο
      if (δεν άνοιξε το report)
      {
         κλείσε τα autoflow, differences
         Μην προχωρείς
      }
      else  //τα autoflow, differences, report ανοικτά
      {
         Προχώρησε στην επεξεργασία
      } // if (δεν άνοιξε το report)...
   } // if (δεν άνοιξε το differences)...
} // if (δεν άνοιξε το autoflow)...
```

Όλα αυτά θα τα βάλουμε σε μια συνάρτηση, ας την πούμε *openFiles*. Να δούμε τι είδους θα είναι αυτή η συνάρτηση και τι παραμέτρους θα έχει.

Η *openFiles* θα ανοίγει τρία ρεύματα από και προς αρχεία ή –με άλλα λόγια– θα δίνει τιμές σε τρία ρεύματα. Θα έχει λοιπόν τρεις *εξερχόμενες* παραμέτρους· θα είναι λοιπόν συνάρτηση *χωρίς τύπο* (`void`). Ναι, αλλά εκείνα τα «`Μην προχωρείς`» και «`Προχώρησε στην επεξεργασία`» πώς θα τα βγάζει προς τα έξω; Χρειαζόμαστε άλλη μια εξερχόμενη παράμετρο ας την πούμε *ok* που θα περνάει προς τα έξω αυτά τα μηνύματα. Αν πάρουμε υπόψη μας ότι τα δύο μηνύματα είναι αμοιβαίως αποκλειόμενα μπορούμε να δηλώσουμε την *ok* τύπου `bool`:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
                ofstream& report, string reprtFlNm,
                bool& ok )
```

Παίρνουμε ολόκληρη την *openFiles* μεταφράζοντας σε C++ το σχέδιο που δώσαμε πιο πάνω:

```
void openFiles( ifstream& autoflow, string autoFlNm,
                ofstream& differences, string difFlNm,
```

```
                ofstream& report, string reprtFlNm,
                bool& ok )
{
   ok = true;
   autoflow.open( "autoflow.txt" );
   if ( autoflow.fail() )
      ok = false;
   else  // το autoflow ανοικτό
   {
      openWrNoReplace( "differences.txt", differences, ok );
      if ( !ok )
         autoflow.close();
      else    // τα autoflow, differences ανοικτά
      {
         openWrNoReplace( "report.txt", report, ok );
         if ( !ok )
         {
            autoflow.close();
            differences.close();
         } // if (δεν άνοιξε το report)...
      } // if (δεν άνοιξε το differences)...
   } // if (δεν άνοιξε το autoflow)...
} // openFiles
```

Εδώ, ας κάνουμε και μια σύγκριση με τις προδιαγραφές μας: Η τιμή της *ok* είναι η τιμή της «*Το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο.*»

Τώρα, ας αρχίσουμε να γράφουμε τη **main**, μεταφράζοντας σε C++ το σχέδιό μας:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      Επεξεργάσου
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.2 «Επεξεργασία»

Ας έρθουμε τώρα στην «`Επεξεργασία`» που έχει προδιαγραφές:

Προϋπόθεση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο, $n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

`Επεξεργασία`

Απαίτηση: Το `autoflow.txt` ανοικτό για διάβασμα, τα `differences.txt` και `report.txt` ανοικτά για γράψιμο.

Στο αρχείο `differences.txt` έχουν γραφεί τα μέλη της ακολουθίας $difference_k$, k: 2..n.

Στο αρχείο `report.txt` έχουν γραφεί οι τέσσερις πρώτες γραμμές του αρχείου `autoflow.txt`,

Εχουν μετρηθεί: το πλήθος τιμών ροής *n* και το πλήθος των οχημάτων *numberOfCars* (`==` $\sum_{k=1}^{n} numberPerMn_k$).

$n > 0$ (το αρχείο `autoflow.txt` έχει μια τουλάχιστον μέτρηση)

Η συνθήκη «*το* `autoflow.txt` *ανοικτό για διάβασμα, τα* `differences.txt` *και* `report.txt` *ανοικτά για γράψιμο*» υπάρχει και στην προϋπόθεση και στην απαίτηση, παραμένει δηλαδή *αναλλοίωτη* από την "`Επεξεργασία`". Δηλαδή, δεν θα βάλουμε πουθενά εντολή "*close*". Αλλά, δεν ξέρεις τι άλλο μπορεί να συμβεί...

Η "$n > 0$" υπάρχει και στην προϋπόθεση και στην απαίτηση· και αυτή αναλλοίωτη; Για να κάνει το πρόγραμμα όλα όσα ζητούνται θα πρέπει να ισχύει αυτή η συνθήκη[7]. Αλλά αυτό δεν το ξέρουμε με βεβαιότητα παρά μόνον αφού προχωρήσουμε στο διάβασμα του αρχείου `autoflow.txt`. Να λοιπόν μια πιθανότητα να υπάρχει κάποιο πρόβλημα:

- Να βρούμε ένα αρχείο με λιγότερες από τέσσερις γραμμές· στην περίπτωση αυτή το *n* είναι 0. Μπορεί οι προδιαγραφές των στοιχείων εισόδου να λένε ότι έχει τουλάχιστον μια τιμή, αλλά ένα καλό πρόγραμμα θα ελέγξει αν πληρούνται οι προδιαγραφές των στοιχείων εισόδου! Θα πρέπει λοιπόν να είμαστε προσεκτικοί όταν έλθει η ώρα να υπολογίσουμε τη μέση τιμή της ροής.

Υπάρχει πιθανότητα να βρούμε άλλο λάθος κατά την επεξεργασία; Φυσικά υπάρχει:

- Αφού το `autoflow.txt` είναι text υπάρχει μεγαλύτερη πιθανότητα να υπάρχουν λάθη μέσα στο αρχείο. Πιθανές προσπάθειες παρεμβάσεων με κάποιον κειμενογράφο μπορεί να έχουν αφήσει διάφορα σφάλματα. Κάποιος έλεγχος εγκυρότητας των στοιχείων είναι αναγκαίος· θέλοντας όμως να κρατήσουμε το παράδειγμά μας απλό δεν θα τον κάνουμε προς το παρόν.

Οι τέσσερις πρώτες γραμμές, ο «τίτλος», θα πρέπει να διαβαστούν –και να αντιγραφούν– ως κείμενο. Από εκεί και πέρα τα ξέρουμε (σχεδόν): Μηδενίζουμε την *n* και τη *numberOfCars* κλπ. Για να είναι τα πράγματα εντάξει θα πρέπει να φτάσουμε να διαβάσουμε μέχρι και μια, τουλάχιστον, τιμή ροής.

```
Αντίγραψε τίτλο
if (εντάξει)
{
   Μηδένισε τους μετρητές
   Διάβασε και επεξεργάσου τις τιμές
   Τελικοί υπολογισμοί
}
```

Όπως κάναμε και πιο πριν, θα πρέπει να διατυπώσουμε προδιαγραφές για τη νέα διάσπαση. Αυτό σου το αφήνουμε ως άσκηση. Στη συνέχεια θα κάνουμε το ίδιο αλλά με αρκετά «ελεύθερο» τρόπο.

Η αντιγραφή του «τίτλου» μπορεί να γίνει όπως ξέρουμε (αντιγραφή αρχείου text), αλλά θα πρέπει να αντιγράψουμε τέσσερις γραμμές μόνον.

Θα γράψουμε μια συνάρτηση *copyTitle* που θα της περνούμε τα ρεύματα *autoflow* και *report* και αυτή θα αντιγράφει από το πρώτο στο δεύτερο τρεις γραμμές. Μέσω μιας παραμέτρου (`bool ok`) θα γνωστοποιεί αν τα κατάφερε.

Τι είδους θα είναι η συνάρτηση; Αφού η συνάρτηση θα εκτελεί αντιγραφή από το ένα αρχείο στο άλλο (είσοδο και έξοδο στοιχείων) η συνάρτηση θα είναι χωρίς τύπο:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
```

Πρόσεξε ότι τα ρεύματα περνούν (πάντοτε) ως παράμετροι αναφοράς.

---

[7] Και για να υπάρχει μια τουλάχιστον διαφορά θα πρέπει να έχουμε "$n \geq 2$".

Θα μπορούσαμε, όπως είπαμε, να γράψουμε τη συνάρτηση όπως μάθαμε στην §8.11, αντιγράφοντας χαρακτήρα προς χαρακτήρα αλλά καλύτερα να χρησιμοποιήσουμε τη *getline()* που μάθαμε στην §10.4. Αφού δηλώσουμε:

```
   int  lineCount;   // μετρητής γραμμών
```

αντιγράφουμε και μετρούμε ως εξής:

```
   lineCount = 0;
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
```

Στο τέλος, αν αντιγράψαμε τέσσερις γραμμές όλα πήγαν εντάξει:

```
   ok = ( lineCount == 3 );
```

Να ολόκληρη η συνάρτηση:

```
void copyTitle( ifstream& autoflow, ofstream& report, bool& ok )
{
   int lineCount( 0 );   // μετρητής γραμμών
   while ( !autoflow.eof() && lineCount < 3 )
   {
      string aLine;
      getline( autoflow, aLine, '\n' );
      report << aLine << endl;
      ++lineCount;
   } // while (... lineCount < 3)
   ok = (lineCount == 3);
} // copyTitle
```

Ας έρθουμε τώρα στη "**Μηδένισε τους μετρητές**". Τι σημαίνει; Δύο εντολές:

```
   n = 0;  numberOfCars = 0;
```

Θα μπορούσαμε να τις βάλουμε έτσι και ούτε γάτα ούτε ζημιά.

Εμείς όμως θα τις βάλουμε σε μια συνάρτηση με όνομα *initialize*. Γιατί; Σε κάθε πρόγραμμα, συνήθως, υπάρχει μια συνάρτηση (ή και περισσότερες) που διαμορφώνει μια αρχική κατάσταση (κάνει κάποια αναλλοίωτη να ισχύει): δίνει αρχικές τιμές σε μεταβλητές, ανοίγει αρχεία κλπ. Θα τη δεις συνήθως με το όνομα *initialize* ή κάτι παρόμοιο[8]. Καλό είναι να αρχίσεις να συνηθίζεις κάτι τέτοιες πάγιες πρακτικές.

Τι τύπο θα δηλώσουμε για τις *n* και *numberOfCars*; Θα πεις «θέλει και ρώτημα; `int` βέβαια!» Για την *n* δεν υπάρχει πρόβλημα. Για την *numberOfCars* όμως, αν έχουμε `INT_MAX == 32767` και μετρήσεις από έναν πολυσύχναστο δρόμο για μεγάλο χρονικό διάστημα, ο `int` δεν είναι αρκετός και καλύτερα να βάλουμε `long`.

```
void initialize( int& n, long& numberOfCars )
{
   n = 0;
   numberOfCars = 0;
} // initialize
```

Και τώρα ερχόμαστε στο «ψητό»: "**Διάβασε και επεξεργάσου τις τιμές**". «Σιγά το ψητό» θα πεις «αθροίσματα και μέσες τιμές υπολογίζουμε συνέχεια.» Σωστό! Αρκεί να προσέξουμε την πρώτη τιμή.

Ας δούμε τώρα πώς θα γίνεται η επεξεργασία. Αφού θέλουμε να υπολογίσουμε μέση τιμή αντιγράφουμε ένα γνωστό «πατρόν» επεξεργασίας από το *Μέση Τιμή 5* (§8.5):

```
   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
```

[8] Στην πραγματικότητα η *openFiles* είναι μια τέτοια συνάρτηση για ολόκληρο το πρόγραμμα. Η *initialize()* είναι η αντίστοιχη για την επεξεργασία.

```
a.open( "exp4.txt" );
a >> x;
while ( !a.eof() )
{
   n = n + 1;               // Αυτά γίνονται για
   sum = sum + x;           // όλους τους αριθμούς
   if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
   {
      selSum = selSum + x;       // Αυτά γίνονται μόνο για
      selN = selN + 1;           // τους επιλεγόμενους αριθμούς
   } // if
   a >> x;
} // while
```

Εδώ:

- το ρεύμα που διαβάζουμε (*autoflow*) είναι ήδη ανοικτό,
- αντί για την *sum* έχουμε την *numberOfCars*,
- αντί για την *x* έχουμε την *numberPerMin*,
- μετά την ανάγνωση μιας τιμής *numberPerMin*, θα πρέπει να πηγαίνουμε στην αρχή της επόμενης γραμμής,
- δεν έχουμε επιλεκτική επεξεργασία.

Άρα παίρνουμε:

```
initialize( n, numberOfCars );
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

έχοντας δηλώσει:

```
string aLine;
char*  p;
```

Πρόσεξε πώς γίνεται η ανάγνωση μιας τιμής από το *autoflow*: αφού σε κάθε γραμμή υπάρχει μια τιμή στην αρχή της, διαβάζουμε ολόκληρη τη γραμμή:

```
getline( autoflow, aLine, '\n' );
```

και μετά, με τη *strtol*, μετατρέπουμε σε **long int** τα ψηφία που υπάρχουν στην αρχή της. Διαστήματα που μπορεί να υπάρχουν πριν από τα ψηφία αγνοούνται από τη *strtol*. Αυτή θα σταματήσει τη μετατροπή όταν βρει τον πρώτο χαρακτήρα που δεν είναι ψηφίο. Την *p* δεν τη χρησιμοποιούμε; Όχι, αλλά θα σημειώσουμε ότι εκεί βρίσκεται ένα από τα μεγαλύτερο πλεονεκτήματα αυτού του τρόπου ανάγνωσης· μπορούμε να χρησιμοποιήσουμε την πληροφορία που μας δίνει για να κάνουμε έλεγχο εγκυρότητας των στοιχείων.

Χρειαζόμαστε όμως ακόμη κάτι: τον υπολογισμό των διαφορών. Ας πούμε ότι κρατούμε την προηγούμενη τιμή της *numberPerMin* στην *previous*. Τότε θα έχουμε:

```
getline( autoflow, aLine, '\n' );
numberPerMin = strtol( aLine.c_str(), &p, 10 );
while ( !autoflow.eof() )
{
   ++n;
   numberOfCars = numberOfCars + numberPerMin;
   difference = numberPerMin - previous;
   differences << difference << endl;
   previous = numberPerMin;
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );
} // while
```

Γιατί βγάλαμε την κλήση προς την *initialize*; Διότι έχουμε και την χωριστή επεξεργασία της πρώτης τιμής που πρέπει να μεσολαβήσει.

Την πρώτη τιμή:

- θα τη μετρήσουμε,
- θα την προσθέσουμε,
- *δεν θα υπολογίσουμε διαφορά*, αφού δεν έχουμε προηγούμενη,
- *ούτε θα γράψουμε κάτι στο αρχείο διαφορών*, αλλά
- θα την βάλουμε στην *previous*, για να χρησιμοποιηθεί από τη δεύτερη τιμή.

Να λοιπόν τι πρέπει να γίνει πριν από τη **while**:

```
   getline( autoflow, aLine, '\n' );
   numberPerMin = strtol( aLine.c_str(), &p, 10 );    // 1η τιμή
   if ( !autoflow.eof() )
   {
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      previous = numberPerMin;
      getline( autoflow, aLine, '\n' );
      numberPerMin = strtol( aLine.c_str(), &p, 10 ); // 2η τιμή
      while ( !autoflow.eof() )
      {
         . . .
      } // while
   } // if (!autoflow.eof())
```

Και τώρα μπορούμε να γράψουμε την *processData* αφού αποφασίσουμε για το είδος και τις παραμέτρους της συνάρτησης. Η συνάρτηση διαβάζει από αρχείο και γράφει σε αρχείο. Θα είναι λοιπόν χωρίς τύπο. Οι παράμετροί του θα είναι τα ρεύματα των δύο αρχείων και οι *n* και *numberOfCars* (εξερχόμενες):

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
```

Ολόκληρη η *processData*:

```
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfCars )
{
   int numberPerMin;          // μια τιμή ροής από το autoflow.txt
   string aLine;
   char*  p;
// επεξεργασία 1ης τιμής
   getline( autoflow, aLine, '\n' );
   if ( !autoflow.eof() )
   {
      numberPerMin = strtol( aLine.c_str(), &p, 10 );   // 1η τιμή
      ++n;
      numberOfCars = numberOfCars + numberPerMin;
      int previous( numberPerMin );     // η προηγούμενη τιμή ροής
      getline( autoflow, aLine, '\n' );
      while ( !autoflow.eof() )
      {
         numberPerMin = strtol( aLine.c_str(), &p, 10 );// 2η τιμή
         ++n;
         numberOfCars = numberOfCars + numberPerMin;
         int difference( numberPerMin - previous );
                                 // η διαφορά των δύο προγούμενων
         differences << difference << endl;
         previous = numberPerMin;
         getline( autoflow, aLine, '\n' );
      } // while
   } // if (!autoflow.eof())
} // processData
```

Εδώ πρόσεξε: Οι προδιαγραφές μας λένε ότι τα αρχεία θα είναι τελικώς ανοικτά. Τι γίνεται όμως με το ρεύμα *autoflow*; Σταματάμε να ασχολούμαστε με αυτό όταν βρούμε `autoflow.eof()`. Όπως ξέρουμε όμως, στην περίπτωση αυτή, το ρεύμα δεν κλείνει βέβαια αλλά έχουμε αναστολή της λειτουργίας του.

Μετά την επεξεργασία των τιμών έχουμε τους τελικούς υπολογισμούς: από τις *n* και *numberOfCars* υπολογίζουμε

- τις ώρες και τα λεπτά καθώς και
- τη μέση τιμή ροής αν $n > 0$. Εδώ ακριβώς γίνεται ο έλεγχος προϋπόθεσης.

Όλα αυτά γράφονται στο `report.txt`:

```
void finish( ofstream& report, int n, long numberOfCars )
{
   int    hours( n / 60 ),    // διάρκεια μέτρησης σε h . . .
          minutes( n % 60 );  // . . . και min
// υπολογισμοί και γράψιμο στο report.txt
   report << endl;
   report << " Η μέτρηση κράτησε ";
   report.width(2);  report << hours << " h και ";
   report.width(2);  report << minutes << " min." << endl;
   report << " Διάβασα " << n << " τιμές." << endl;
   report << " Μέτρησα " << numberOfCars << " οχήματα συνολικά"
          << endl;
   if ( n > 0 )
   {
      double averageFlow( static_cast<double>(numberOfCars) / n );
                                        // μέση τιμή ροής οχημάτων
      report << " Μέση Τιμή Ροής: ";
      report.setf( ios::fixed, ios::floatfield );
      report.precision(1);
      report.width(5);
      report << averageFlow << " οχήματα/min" << endl;
   }
} // finish
```

Πρόσεξε ότι υπολογίζουμε και γράφουμε τη μέση τιμή ροής αφού προηγουμένως εξασφαλίσουμε ότι $n > 0$.

Τώρα, μπορούμε να γράψουμε την *processing()*. Θα είναι και αυτή `void` και θα μας δίνει μέσω μιας παραμέτρου (`bool ok`) την πληροφορία για το τι κατάφερε να κάνει:

```
void processing( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   int n;             // πλήθος τιμών ροής στο autoflow.txt,
   long numberOfCars; // το άθροισμα των τιμών ροής.
   bool ok;

   copyTitle( autoflow, report, ok );
   if ( ok )
   {
      initialize( n, numberOfCars );
      processData( autoflow, differences, n, numberOfCars );
      finish( report, n, numberOfCars );
   } // if (ok)...
} // processing
```

Προχωρώντας στο γράψιμο της **main**, έχουμε:

```
int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
```

```
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      Κλείσε τα ρεύματα
   }
   else  {
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
   } // if (ok)...
} // main
```

13.11.3 «Κλείσε τα Ρεύματα»

Η *closeFiles* κλείνει τα τρία ρεύματα με τα οποία δουλέψαμε:

```
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok )
{
   autoflow.close();

   differences.close();
   if ( differences.fail() )
   {
      cout << "Τις χάσαμε τις διαφορές!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = true;

   report.close();
   if ( report.fail() )
   {
      cout << "Γράψε την έκθεση με το χέρι!" << endl
           << "Δεν μπορώ να κλείσω το αρχείο" << endl;
      ok = false;
   }
   else
      ok = ok && true; // άχρηστη
} // closeFiles
```

Μπορεί να αποτύχει το κλείσιμο των αρχείων; Βεβαιότατα! Αν πάρουμε υπόψη μας ότι με το κλείσιμο αντιγράφεται στο αρχείο το όποιο περιεχόμενο του ενταμιευτή, σκέψου την εξής περίπτωση: ας πούμε ότι γράφεις σε αφαιρούμενο δίσκο –π.χ. σε δισκέτα– και βιάζεσαι να την αφαιρέσεις πριν εκτελεσθεί η *close*.

Παρατήρηση: ▶
Βεβαίως εδώ τελειώνουμε όλες τις γραμμές με "`endl`"· έτσι ο ενταμιευτής είναι σχεδόν μονίμως άδειος.◀

13.11.4 Ολόκληρο το Πρόγραμμα

Ολόκληρο το πρόγραμμα θα είναι κάπως έτσι:

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

using namespace std;

void openWrNoReplace( string flNm, ofstream& newStream,
                      bool& ok );
```

```
void openFiles( ifstream& autoflow,
                ofstream& differences, ofstream& report,
                bool& ok );
void process( ifstream& autoflow,
              ofstream& differences, ofstream& report,
              bool& ok );
void copyTitle( ifstream& autoflow, ofstream& report,
                bool& ok );
void initialize( int& n, long& numberOfVehicles );
void processData( ifstream& autoflow, ofstream& differences,
                  int& n, long& numberOfVehicles );
void finish( ofstream& report,
             int n, long numberOfVehicles );
void closeFiles( ifstream& autoflow,
                 ofstream& differences, ofstream& report,
                 bool& ok );

int main()
{
   ifstream autoflow;    // ρεύμα από το αρχείο autoflow.txt
   ofstream differences; // ρεύμα προς το αρχείο differences.txt
   ofstream report;      // ρεύμα προς το αρχείο report.dta
   string   autoFlNm( "autoflow.txt" ),
            difFlNm( "differences.txt" ),
            reprtFlNm( "report.txt" );
   bool     ok;          // τιμή true αν όλα τα ρεύματα άνοιξαν

   openFiles( autoflow, autoFlNm, differences, difFlNm,
              report, reprtFlNm, ok );
   if ( ok )  // όλα τα ρεύματα άνοιξαν επιτυχώς
   {
      processing( autoflow, differences, report, ok );
      if ( !ok )
         cout << "Δεν βρήκα ούτε μια τιμή ροής" << endl;
      closeFiles( autoflow, differences, report, ok );
      if ( ok ) // γράψιμο και κλείσιμο επιτυχώς
         cout << "Τέλος καλό, όλα καλά..." << endl;
   }
   else
      cout << " Δεν ανοίγουν τα αρχεία! Γειά χαρά!" << endl;
} // main
```

13.12 Δυο Λόγια για το Παράδειγμά μας

Έχοντας δει αρκετά στοιχεία προγραμματισμού, θελήσαμε να σου δώσουμε ένα παράδειγμα ανάπτυξης ενός μη-τετριμμένου προγράμματος. Η ανάπτυξη βασίστηκε στην αρχή «διαίρει και βασίλευε» (divide and conquer). Δηλαδή, ενώ το αρχικό πρόβλημα έμοιαζε κάπως δύσκολο, το διασπάσαμε σε μικρότερα και μικρότερα, μέχρι που φτάσαμε σε προβλήματα με σχετικά απλή λύση. Όπως είχαμε πει στην §0.5, αυτή η διαδικασία λέγεται και **βήμα προς βήμα ανάλυση** (step by step refinement). Τώρα την είδαμε στην πράξη.

Ξεκινήσαμε από το αρχικό πρόβλημα και πήγαμε προς τις λεπτομέρειες, **από πάνω προς τα κάτω** (top - down). Σε αυτήν την πορεία, δημιουργούσαμε δικές μας εντολές, π.χ.: *openFiles()*, *initialize()*, *finish()* κλπ, που στη συνέχεια τις υλοποιούσαμε ως συναρτήσεις. Ένας τρόπος απεικόνισης αυτής της διαδικασίας είναι το **ιεραρχικό διάγραμμα** που βλέπεις στο Σχ. 13-3. Σε αυτό βλέπεις τα διαδοχικά επίπεδα ανάλυσης, ενώ οι συναρτήσεις που καλούνται από κάθε συνάρτηση φαίνονται σαν κλαδιά της. Το διάγραμμα αυτό είναι ένας τρόπος καταγραφής των αλλεξαρτήσεων των συναρτήσεων (για τις οποίες μιλούσαμε στην §13.6.2.)

Στο διάγραμμα αυτό δεν παρατίθενται συνήθως συναρτήσεις όπως η *sqrt()*, η *strcpy()* και άλλες τέτοιες από τις βιβλιοθήκες της C++ ή άλλες βιβλιοθήκες συναρτήσεων. Έτσι, πρέπει να βλέπεις και την *openWrNoReplace()*. Αυτή θα τη βάλουμε αργότερα σε μια δική μας βιβλιοθήκη.

Ένα άλλο σημείο που πρέπει να προσέξεις, είναι το πώς αποφασίσαμε το ποιες συναρτήσεις θα γράψουμε. Ούτε για μια στιγμή δεν είχαμε κάποια αμφιβολία. Οι συναρτήσεις «ξεπήδησαν», μπορούμε να πούμε, αυθόρμητα. Αυτό *δεν σημαίνει ότι η ανάλυση είναι μοναδική*.

Ενδιαφέρον έχει και το λογικό δέσιμο των συναρτήσεων. Κάθε μια προετοιμάζει την κατάσταση για την επόμενη (ή τις επόμενες) –όπως π.χ. η *openFiles()* για όλες τις επόμενες, η *initialize()* για την *processData()*– και κάθε μια συνεχίζει τη δουλειά από εκεί που την άφησε η προηγούμενη –όπως π.χ. οι *copyTitle()*, *processData()*.

Αυτή είναι η μέθοδος του **δομημένου προγραμματισμού** (structured programming) και είναι η μόνη μέθοδος που μπορείς να χρησιμοποιήσεις προς το παρόν. Αργότερα θα δεις και άλλες μεθόδους σχεδίασης λογισμικού και (γενικότερα) συστημάτων. Αλλά, παντού θα βλέπεις αυτές εδώ τις αρχές. Θα δεις ακόμη ότι έχει γίνει πολλή δουλειά στο δέσιμο αυτής της μεθόδου με τη μαθηματική λογική.

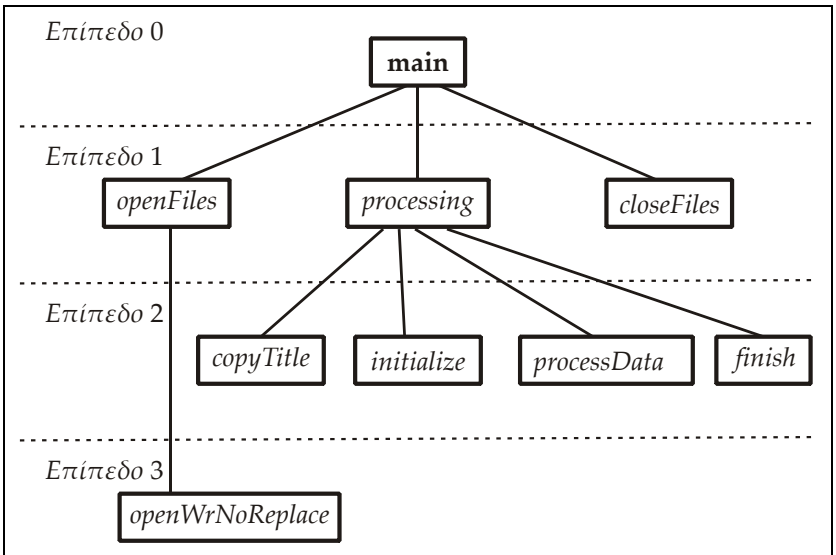
Πρόσεξε ακόμη τη σειρά υλοποίησης των συναρτήσεων. Αρχίσαμε από το τελευταίο επίπεδο, υλοποιώντας συναρτήσεις που δεν καλούν άλλες και προχωρήσαμε προς τα πάνω. Τελευταία υλοποιήθηκε η **main**. Αυτή η διαδικασία λέγεται **υλοποίηση από κάτω προς τα πάνω** (bottom-up implementation) και συνήθως πάει μαζί με τη **σχεδίαση από πάνω προς τα κάτω** (top-down design, bottom-up implementation).

Παρατηρήσεις: ▶

Κατά τ' άλλα, το πρόγραμμα που γράψαμε θέλει κι άλλη δουλειά, κυρίως στον τομέα της ασφάλειας.

1. Ένα πράγμα που δεν μας απασχόλησε καθόλου είναι ο **έλεγχος εγκυρότητας** των στοιχείων (data validation) που διαβάσαμε. Για παράδειγμα, δεν είναι δυνατόν να έχουμε *αρνητικές* τιμές ροής αλλά και μια τιμή 5000 *οχήματα/min* είναι απαράδεκτη. Ας πούμε ότι ο έλεγχος εγκυρότητας των στοιχείων έχει γίνει πιο πριν, από άλλο πρόγραμμα.

2. Ας έλθουμε στο άνοιγμα των αρχείων: Το πρόγραμμά μας είναι εξαιρετικώς ανελαστικό· ένα καλύτερο πρόγραμμα, κάθε φορά που θα εύρισκε πρόβλημα με κάποιο αρχείο, θα έπρεπε να ζητάει νέο όνομα αρχείου. ◀



Σχ. 13-3 Ιεραρχικό διάγραμμα της εφαρμογής. Η *openWrNoReplace* (επομένως και το Επίπεδο 3) θα ήταν καλό να μην υπάρχει στο διάγραμμα.

Ασκήσεις

Α Ομάδα

Για κάθε συνάρτηση που ζητείται στις Ασκ. 13-1..13-11 θα πρέπει να δικαιολογήσεις το είδος της συνάρτησης (με τύπο ή `void`*) και το είδος και τον τύπο κάθε παραμέτρου. Γράψε πρόγραμμα που θα δοκιμάζει τη συνάρτηση.*

13-1 Γράψε τη συνάρτηση *output2Dar()*, όπως την προδιαγράψαμε στο Παράδ. 6 της §13.9.3.

13-2 (συνέχεια της Άσκ. 12-5) Γράψε συνάρτηση που θα τροφοδοτείται με έναν δισδιάστατο πίνακα και δύο φυσικούς *c*1, *c*2 και θα αντιμεταθέτει τις τιμές των στοιχείων των στηλών *c*1, *c*2.

13-3 (συνέχεια της Άσκ. 12-12) Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και έναν φυσικό *k* και θα αντιμεταθέτει τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης του πίνακα.

13-4 Γράψε συνάρτηση με όνομα *h3*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με τρεις πραγματικές τιμές, ας τις πούμε χ_1, χ_2, χ_3, και έναν ακέραιο *n* και θα επιστρέφει τη διαφορά $\chi_1 - \chi_2$ αν *n* == 0, τη $\chi_2 - \chi_3$ αν *n* == 1 και τη $\chi_3 - \chi_1$ αν *n* == 2.

13-5 Μια συνάρτηση ορίζεται στο {1,2,3}×ℝ×ℝ ως εξής:

$$g(n, x, y) = \begin{cases} x + y, & n = 1 \\ x\eta\mu y, & n = 2 \\ x - 1/y, (y \neq 0) & n = 3 \end{cases}$$

Πώς θα υλοποιήσουμε τη *g()* στη C++;

13-6 Θέλουμε μια συνάρτηση, με όνομα *q1*, που θα τροφοδοτείται μέσω των παραμέτρων της, με τρεις πραγματικούς αριθμούς, ας τους πούμε *x*, *y*, *z*, και θα υπολογίζει και θα μας επιστρέφει δύο τιμές, των παραστάσεων p_1 και p_2, όπου:

αν $z > 0$ τότε $p_1 = \dfrac{x^z + y^z}{z}$ και $p_2 = z^{y+x}$,

αν $z < 0$ τότε $p_1 = \dfrac{x^{-z} - y^{-z}}{z}$ και $p_2 = (-z)^{y-x}$.

Δεν επιτρέπεται να κληθεί η *q1()* με τιμή της παραμέτρου *z* ίση με 0 (μηδέν).

13-7 Γράψε συνάρτηση που θα τροφοδοτείται, μέσω των παραμέτρων της, με δύο μονοδιάστατους πίνακες x, y –με το ίδιο πλήθος *n* στοιχείων τύπου `double`–και με έναν πραγματικό *w*. Η συνάρτηση θα κάνει το εξής: θα αλλάζει τις τιμές των στοιχείων των x και y ως εξής: αν αρχικώς x_k == *a* και y_k == *b* τότε τελικά θα πρέπει να έχουμε: x_k == *a* – *wb* και y_k == *a* + *wb*, για όλα τα *k* από 0 μέχρι *n* – 1.

Β Ομάδα

13-8 Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει τον **ανάστροφό** του (τον πίνακα που έχει ως γραμμές τις στήλες του αρχικού). Γράψε συνάρτηση που θα κάνει το ίδιο για τυχαίο δισδιάστατο πίνακα (όχι κατ' ανάγκη τετραγωνικό).

13-9 Κάθε τετραγωνικός πίνακας M (*n*×*n*) μπορεί να γραφεί ως άθροισμα δύο τετραγωνικών πινάκων *n*×*n*· ενός συμμετρικού S και ενός αντισυμμετρικού A. Τα στοιχεία τους:

$$s_{rc} = \frac{m_{rc} + m_{cr}}{2} (= s_{cr}) \text{ και } a_{rc} = \frac{m_{rc} - m_{cr}}{2} (= -a_{cr})$$

Γράψε συνάρτηση που θα τροφοδοτείται με έναν τετραγωνικό πίνακα και θα μας επιστρέφει το συμμετρικό και το αντισυμμετρικό μέρος του.

13-10 Με βάση το προγράμματα που έγραψες για τις Άσκ. 5-7, 5-8 γράψε μια συνάρτηση, με όνομα *trinomial()*, που:

- Θα τροφοδοτείται με τους (πραγματικούς) συντελεστές *a*, *b*, *c* ενός τριωνύμου $ax^2 + bx + c$.
- Θα υπολογίζει και θα επιστρέφει τέσσερις πραγματικές τιμές *reX1*, *imX1*, *reX2*, *imX2*, τα πραγματικά και φανταστικά μέρη των ριζών της εξίσωσης $ax^2 + bx + c = 0$.
- Θα επιστρέφει και μια ακέραιη τιμή, *n*, που θα είναι:
  - −1, αν η εξίσωση είναι αδύνατη,
  - 0 αν δεν έχει πραγματικές ρίζες,
  - 1, αν είναι πρώτου βαθμού με μία ρίζα (στη *reX1*),
  - 2, αν έχει δύο πραγματικές ρίζες (οι *imX1*, *imX2* θα είναι 0),
  - *INT_MAX*, αν είναι αόριστη.

Γράψε πρόγραμμα που θα διαβάζει τρεις πραγματικούς *a*, *b*, *c* από το πληκτρολόγιο και, καλώντας την *trinomial()*, θα λύνει την $ax^2 + bx + c = 0$.

13-11 Γράψε μια συνάρτηση, με όνομα *addPrev*, που θα τροφοδοτείται με ένα μονοδιάστατο πίνακα a με στοιχεία τύπου `double` και θα αντικαθιστά την τιμή του κάθε στοιχείου (εκτός από το a[0]) με το άθροισμά της παλιάς και της τιμής του προηγούμενου στοιχείου, δηλαδή: $a[k]_{\text{νέα}} = a[k]_{\text{παλιά}} + a[k-1]_{\text{παλιά}}$, k: 1..n−1. (Προσοχή! Θα πας από την αρχή προς το τέλος ή από το τέλος προς την αρχή;).

13-12 Κάνε αυτό που σου αφήσαμε ως άσκηση στην §13.11.2: διατύπωσε προδιαγραφές για τα επί μέρους βήματα της διάσπασης του βήματος "`Επεξεργασία`" του αρχικού προγράμματος. Μετά έλεγξε τις συναρτήσεις που γράψαμε (δηλαδή: κάνε μια άτυπη επαλήθευση). Συμμορφώνονται με τις προδιαγραφές;

13-13 α) Γράψε συνάρτηση, ας την πούμε *toSec*, που θα παίρνει *h* (0..23), *min*, *sec* μιας χρονικής στιγμής και θα τα μετατρέπει σε *sec* από τα τελευταία μεσάνυκτα.

β) Γράψε συνάρτηση, ας την πούμε *toHms*, αντίστροφη της προηγούμενης: θα παίρνει δευτερόλεπτα από τα τελευταία μεσάνυκτα και θα μας δίνει *h*, *min*, *sec*.

γ) Γράψε συνάρτηση *readHms()* που θα διαβάζει από το πληκτρολόγιο, θα ελέγχει και θα περνάει στο πρόγραμμά μας μια χρονική στιγμή που δίνεται στη μορφή *h*, *min*, *sec*.

Γράψε πρόγραμμα που θα χρησιμοποιεί τις παραπάνω συναρτήσεις για να

- διαβάσει από το πληκτρολόγιο δύο τριάδες ακεραίων h_1, m_1, s_1 και h_2, m_2, s_2 που είναι ώρα, πρώτα λεπτά και δευτερόλεπτα δύο χρονικών στιγμών,
- να ελέγξει αν είναι σωστές: $0 \le h_1, h_2 < 24$, $0 \le m_1, m_2 < 60$, $0 \le s_1, s_2 < 60$,
- να υπολογίσει και θα μας δώσει τη χρονική διαφορά μεταξύ των δύο χρονικών στιγμών σε ώρες, λεπτά και δευτερόλεπτα στη μορφή: *hh*:*mm*:*ss*. Π.χ.: Αν δοθούν: ***15 59 0*** και ***16 1 12*** γράφει: `00:02:12`.

Γ Ομάδα

13-14 Στο παιχνίδι "Φιδάκι" (snakes and ladders) ο κάθε παίκτης έχει να διανύσει έναν δρόμο με 100 βήματα αριθμημένα από 1 μέχρι 100. Ο κάθε παίκτης προχωράει όσα βήματα δείξει το ζάρι, που ρίχνει κάθε φορά που έρχεται η σειρά του. Για να τερματίσει ένας παίκτης, θα πρέπει να φέρει ζαριά που να τον φέρει ακριβώς στο 100. Αν φέρει μεγαλύτερη επιστρέφει πίσω όσα βήματα περισσεύουν (π.χ. αν είναι στο 96 και φέρει 6 θα προχωρήσει μέχρι το 100 με τα τέσσερα και θα επιστρέψει στο 98 με τα δύο που περίσσεψαν). Μια παρτίδα του παιχνιδιού τελειώνει όταν τερματίσει ένας παίκτης.

- Σε όρισμένα βήματα υπάρχουν φίδια· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα το φίδι τον «καταπίνει» και τον βγάζει στο βήμα που είναι η ουρά του (μερικά βήματα πιο πίσω).
- Σε άλλα βήματα υπάρχουν σκάλες· αν ένας παίκτης φτάσει σε ένα τέτοιο βήμα «ανεβαίνει» τη σκάλα που τον βγάζει στο βήμα που είναι η άλλη άκρη της (μερικά βήματα πιο μπροστά).

Στους παρακάτω πίνακες βλέπεις πού αρχίζουν και πού καταλήγουν οι σκάλες και τα φίδια:

Σκάλες		**Φίδια**	
αρχή	τέλος	αρχή	τέλος
5	12	10	2
17	30	25	15
40	66	37	20
60	78	50	25
75	96	88	64

Γράψε πρόγραμμα που θα προσομοιώνει μια παρτίδα του παιχνιδιού με τέσσερις παίκτες.

13-15 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο έναν φυσικό αριθμό n και θα μας δίνει όλους τους πρώτους φυσικούς που είναι ≤ n. Ο αλγόριθμος που θα χρησιμοποιήσεις είναι το *Κόσκινο του Ερατοσθένη*, που δουλεύει ως εξής:

1. Γράψε όλους τους φυσικούς από 1 μέχρι n
2. Βάλε $p = 2$
3. while ($p^2 \le n$) {

3.1 Διάγραψε όλα τα πολλαπλάσια του p

3.2 Βάλε p = επόμενος αριθμός που δεν έχεις διαγράψει

}

Παράδειγμα: (n = 22)

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
p = 2
2 3   5   7   9    11    13    15    17    19    21
p = 3
2 3   5   7        11    13          17    19
```