κεφάλαιο

6

Επαναλήψεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Να κάνεις το δεύτερο βήμα στη χρήση συνθηκών για τον έλεγχο εκτέλεσης ενός προγράμματος: Να προγραμματίζεις επαναλαμβανόμενη εκτέλεση (ομάδων) εντολών.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να γράφεις προγράμματά που θα έχουν *επαναληπτικούς* υπολογισμούς. Με αυτά αρχίζεις να εκμεταλλεύεσαι τη μεγάλη δύναμη του υπολογιστή: Πολύ μεγάλη ταχύτητα στην εκτέλεση επαναληπτικών αλγορίθμων.

Έννοιες κλειδιά:

- *εντολές επανάληψης - βρόχοι*
- *εντολή* `while`
- *εντολή* `for`
- *αναλλοίωτη επανάληψης*
- *τιμή-φρουρός*
- *τερματισμός επανάληψης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Στον προγραμματισμό παρουσιάζεται συχνά η ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Φυσικά, αυτό που μας ενδιαφέρει είναι να γράψουμε τις εντολές μια φορά μόνον. Η C++, όπως και οι άλλες γλώσσες προγραμματισμού, μας δίνει αυτήν τη δυνατότητα.

6.1 Επαναλήψεις

Ας ξεκινήσουμε με ένα παράδειγμα.

Έστω ότι θέλουμε να υπολογίσουμε και εκτυπώσουμε το άθροισμα και τη μέση αριθμητική τιμή *n* (π.χ.10) πραγματικών αριθμών –ας τους πούμε t_1, t_2,... t_n– που δίνονται στην είσοδο του υπολογιστή.

Με όσα ξέρουμε μέχρι τώρα, θα έπρεπε να δηλώσουμε *n* μεταβλητές *x*1, *x*2, ... *x*10 και να δώσουμε τις εντολές:

```
cin >> x1;  cin >> x2; ... cin >> x10;
sum = x1 + x2 + ... + x10;
```

Βέβαια, ένα τέτοιο πρόγραμμα θα ήταν τρομακτικό, αν μάλιστα το *n* δεν είναι 10, αλλά είναι 100 ή 1000 ή 10000.

Ας προσπαθήσουμε να το γράψουμε αλλιώς. Κοίταξε την παρακάτω εναλλακτική λύση:

```
sum = 0;                    // sum == 0
cin >> x;  sum = sum + x;   // (x == t1) && (sum == Σ(j:1..1)tj)
cin >> x;  sum = sum + x;   // (x == t2) && (sum == Σ(j:1..2)tj)
  . . .
cin >> x;  sum = sum + x;   // (x == tN) && (sum == Σ(j:1..n)tj)
```

Τι κάνει το (κάθε) *m*-οστό ζευγάρι εντολών

```
cin >> x;  sum = sum + x;
```

Διαβάζει από το πληκτρολόγιο τη *m*-οστή τιμή (t_m), την αποθηκεύει στη θέση της μνήμης *x* και την προσθέτει στη μεταβλητή *sum*. Όταν εκτελεσθεί το επόμενο ζευγάρι *η προηγούμενη τιμή που υπάρχει στη x θα σβηστεί*. Αυτό δεν μας δημιουργεί πρόβλημα, αφού από τις προδιαγραφές του προγράμματος φαίνεται ότι ο μόνος προορισμός κάθε τιμής που διαβάζεται είναι να προστεθεί στο άθροισμα.

Με τη *sum* τι γίνεται; Αρχικώς της δίνουμε την τιμή 0. Μετά το πρώτο ζευγάρι εντολών η τιμή της είναι η πρώτη τιμή που διαβάστηκε. Μετά το δεύτερο ζευγάρι η τιμή της *sum* είναι το άθροισμα των δύο πρώτων τιμών, μετά το *m*-οστό ζευγάρι η *sum* έχει ως τιμή το μερικό άθροισμα των *m* πρώτων τιμών. Μετά το *n*-οστό ζευγάρι η *sum* έχει ως τιμή το άθροισμα των *n* τιμών που πληκτρολογήθηκαν.

Όλα αυτά φαίνονται και στις συνθήκες που βάζουμε μετά το κάθε ζευγάρι εντολών. Με το `Σ(j: 1..m)tj` εννοούμε το: $\sum_{j=1}^{m} t_j$.

Ωραία λοιπόν! Φαίνεται ότι και αυτές οι εντολές λύνουν το πρόβλημα. Τι κερδίσαμε που τις γράψαμε; Η λύση του προβλήματός μας διατυπώθηκε ως επανάληψη μιας ακολουθίας εντολών: οι εντολές "`cin >> x;  sum = sum + x;`" εκτελούνται *n* φορές. Μπορούμε να πούμε στον υπολογιστή να τις εκτελέσει *n* φορές με αρκετά συνοπτικό τρόπο:

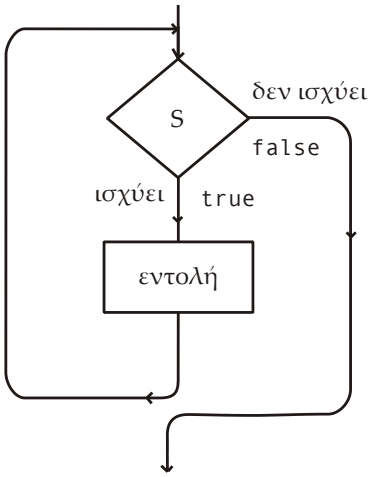
```
sum = 0;  m = 1;
όσο (m <= n) να εκτελείς την εντολή:
{
    cin >> x;  sum = sum + x;
 // (x == tm) && (sum == Σ(j:1..m)tj)
    m = m + 1;
}
```

Αυτό που εννοούμε εδώ είναι το εξής: όσο βρίσκει το $m \le n$ να εκτελεί τις τρεις εντολές που «πακετάραμε» σε μια σύνθετη εντολή. Η *m* είναι ένας **μετρητής** (counter) –συνήθως μεταβλητή τύπου (`unsigned`) `int`. Η τελευταία επαναλαμβανόμενη εντολή αυξάνει κάθε φορά την τιμή της *m* κατά 1· έτσι, αφού ξεκινάει από 1, όταν θα πάψει να ισχύει η συνθήκη "`m <= n`" η σύνθετη εντολή θα έχει εκτελεσθεί *n* φορές. Ακόμη, πρόσεξε ότι, αφού η *m* ξεκινάει από 1 και αυξάνεται κατά 1, είναι σίγουρο ότι κάποτε το *m* θα γίνει μεγαλύτερο από το *n*.

Το ίδιο πράγμα γράφεται στην C++:

```
sum = 0;  m = 1;
while ( m <= n )  {  cin >> x;  sum = sum + x;
                  // (x == tm) && (sum == Σ(j:1..m)tj)
                     m = m + 1;
                  }
```

Το **while** (που σημαίνει: όσο, για όσο, εφ' όσον) είναι λεξικό σύμβολο της C++. Η εντολή **while** είναι μια **επαναληπτική εντολή** (repetitive statement) και στο Πλ. 6.1 μπορείς να δεις την περιγραφή της. Η λειτουργία της εντολής **while** δίνεται επίσης από το λογικό διάγραμμα του Σχ. 6-1. Βλέποντάς το, μπορείς να κατάλάβεις γιατί οι επαναληπτικές εντολές λέγονται και **εντολές ανακύκλωσης** ή **βρόχοι** (loops).



Σχ. 6-1 Λογικό διάγραμμα της **while**.

Η συνθήκη $(x = t_m)$ **&&** $(sum = \sum_{j=1}^{m} t_j)$ δεν παύει να ισχύει, στο σημείο του προγράμματος που τη γράψαμε, όσο εκτελούνται ξανά και ξανά οι επαναλαμβανόμενες εντολές· λέμε λοιπόν ότι είναι **αναλλοίωτη** (invariant) της επανάληψης. Για αναλλοίωτες θα συζητήσουμε πιο εκτεταμένα παρακάτω.

Πρόσεξε το εξής: η **while**, όπως και η **if**, περιμένει μετά τη συνθήκη *μια* εντολή· αν θέλουμε να βάλουμε περισσότερες, όπως είδες ήδη στο παράδειγμά μας, τις «συσκευάζουμε» με ένα άγκιστρο (**{**) στην αρχή και ένα άγκιστρο (**}**) στο τέλος σε μια *σύνθετη εντολή*.

Άλλα παραδείγματα εντολών **while**:

```
while ( a > b )  a = a - b;
```

```
while ( n > 0 )
{
   a = a + n*n;
   n = n - 1;
}
```

Ας επιστρέψουμε τώρα στο πρόβλημά μας που διατυπώθηκε στο παράδειγμα. Το παρακάτω πρόγραμμα –Μέση Τιμή 1– είναι μια λύση με χρήση της εντολής **while**:

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

   int m;       // Μετρητής των επαναλήψεων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  m = 1;
   while ( m <= n )
   {
      cout << "Δώσε έναν αριθμό: ";  cin >> x;          // x = tm
      sum = sum + x;           // (x == tm) && (sum = Σ(j:1..m)tj)
      m = m + 1;           // Ετοιμαζόμαστε για τον επόμενο αριθμό
   } // while
   avrg = sum / n;
   cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
}
```

Πλαίσιο 6.1

Η Εντολή while

"`while`", "`(`", *συνθήκη*, "`)`", *εντολή*

Συστατικά:

- το `while` είναι λεξη-κλειδί,
- η *συνθήκη* είναι μια λογική παράσταση,
- η *εντολή* είναι μια οποιαδήποτε εντολή της C++ και λέγεται *περιοχή της επανάληψης*.

Εκτέλεση της εντολής **while**:

Υπολογίζεται πρώτα η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο `while`.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη.

Υπολογίζεται ξανά η λογική τιμή της συνθήκης, που δίνεται μετά το λεξικό σύμβολο **while**.

Αν η τιμή αυτή είναι `true` (αληθής), τότε

εκτελείται η εντολή που ακολουθεί τη συνθήκη κ.ο.κ.

Αν η τιμή της συνθήκης βρεθεί `false` τότε η εντολή που ακολουθεί τη συνθήκη δεν εκτελείται και η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί τη **while**.

Αν η τιμή της συνθήκης είναι `false` την πρώτη φορά που θα ελεγχθεί, η εντολή που ακολουθεί τη συνθήκη δεν θα εκτελεστεί ποτέ!

Οι εντολές "`sum = 0; m = 1`", που δίνονται πριν από την εντολή **while**, εκτελούνται βέβαια μόνο μια φορά και αποτελούν το *μέρος προετοιμασίας* της επανάληψης, δηλ. το μέρος όπου καθορίζονται οι αρχικές τιμές μεταβλητών, που χρησιμοποιούνται μέσα στην περιοχή της **while**.

Στο πρόγραμμά μας το *n*, που καθορίζει το πλήθος των επαναλήψεων της **while**, είναι σταθερά (εδώ *n*=10), και επομένως η επαναληπτική διαδικασία δεν μπορεί να σταματήσει αν δεν διαβαστούν *n* αριθμοί. Μια τέτοια επιλογή όμως, που γίνεται *όταν γράφουμε το πρόγραμμα*, οδηγεί σε ένα πρόγραμμα εξαιρετικώς ανελαστικό. Πιο βολικές είναι οι λύσεις που μας επιτρέπουν να καθορίζουμε το πλήθος των αρχικών δεδομένων (στο παράδειγμά μας τον αριθμό *n*), *όταν αρχίζει η εκτέλεση του προγράμματος*. Έτσι δεν χρειάζεται αλλαγή (εδώ του ορισμού *n* = 100) και νέα μεταγλώττιση του προγράμματος κάθε φορά που αλλάζει το πλήθος των στοιχείων εισόδου. Σε ένα τέτοιο πρόγραμμα –ας το πούμε Μέση Τιμή 2– το *n* είναι *μεταβλητή* και η τιμή της πληκτρολογείται από το χρήστη πριν αρχίσει να πληκτρολογεί τους άλλους αριθμούς:

```
int n;       // Τώρα το πλήθος είναι μεταβλητή
   . . .
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
   . . .
```

Τώρα όμως προσοχή: Θα πρέπει να έχουμε ως προϋπόθεση *n* > 0 και να την ελέγχουμε μόλις διαβάσουμε την τιμή της *n*:

```
cout << " Δώσε το ΠΛΗΘΟΣ των στοιχείων: ";
cin >> n;       // Τώρα γίνεται γνωστό το πλήθος
if ( n > 0 )
{
   sum = 0;  m = 1;
     . . .
}
```

Πλαίσιο 6.2α
Υπολογισμός Αθροίσματος

```
sum = 0;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   sum = sum + x;
      . . .
} // while
// sum == Σx
```

Πλαίσιο 6.2β
Υπολογισμός Γινομένου

```
prod = 1;
while ( S )
{
      . . .
   υπολογισμός (ανάγνωση)
   της επόμενης τιμής x
   prod = prod * x;
      . . .
} // while
// prod == Πx
```

```
else
   cout << " Το πλήθος πρέπει να είναι θετικό" << endl;
```

Αφήνουμε σε σένα, για άσκηση, να γράψεις ολόκληρο το Μέση Τιμή 2 (Άσκ. 6-1).

Όπως θα δεις και στη συνέχεια, έτσι υπολογίζουμε τα αθροίσματα ακόμη και αν ο έλεγχος της επανάληψης είναι διαφορετικός ή αν οι τιμές δημιουργούνται από το πρόγραμμα και δεν διαβάζονται από το πληκτρολόγιο. Σου το δίνουμε λοιπόν σαν συνταγή στο Πλ. 6.2α. Παρόμοιος είναι και ο υπολογισμός γινομένου (Πλ. 6.2β).

Αυτή η μορφή επανάληψης –που είδαμε στα παραπάνω παραδείγματα– είναι η πιο απλή και λέγεται **μετρούμενη** (counted) επανάληψη. Θα ασχοληθούμε με αυτήν ξανά στη συνέχεια.

6.1.1 Άγνωστο Πλήθος Στοιχείων – Τιμή-Φρουρός

Συχνά το πλήθος των στοιχείων που θέλουμε να δώσουμε στον υπολογιστή δεν είναι γνωστό. Σκέψου, για παράδειγμα, μερικά φύλλα με μετρήσεις από κάποιο πείραμα. Κάθε φορά που τα μετράς βρίσκεις και διαφορετικό πλήθος. Δεν είναι καλύτερο να τα μετρήσει το πρόγραμμα;

Έστω ότι θέλουμε να βρούμε και να τυπώσουμε το άθροισμα, τη Μέση Τιμή και το πλήθος πραγματικών αριθμών που θα πληκτρολογηθούν στην είσοδο του Υπολογιστή. *Ενώ το πλήθος τους δεν είναι γνωστό*, ξέρουμε ότι οι αριθμοί μας είναι *μη-μηδενικοί*.

Το πώς θα υπολογίσουμε το άθροισμα και τη μέση τιμή το ξέρουμε· δεν ξέρουμε όμως *πότε θα σταματήσει η επανάληψη*. Ή αλλιώς: *πώς θα πούμε στον Υπολογιστή ότι τελείωσαν τα στοιχεία εισόδου* και να μην περιμένει άλλα.

Είναι φανερό ότι την ώρα που εκτελείται το πρόγραμμα και δίνουμε τα στοιχεία εισόδου θα πρέπει να δώσουμε στον ΗΥ το σύνθημα «τέλος στοιχείων» με κάποιο τρόπο. Αλλά ποιά είναι η δυνατότητά μας να δώσουμε κάτι στον ΗΥ; Καθορίζεται με απόλυτη ακρίβεια από την εντολή: "`cin >> x`" που εκτελείται ξανά και ξανά. Μπορούμε λοιπόν να δώσουμε μια τιμή στη *x* που να είναι συνθηματική και να σημαίνει «τέλος». Ας δούμε τώρα, τι τιμές μπορεί να πάρει η *x*: τιμές τύπου **`double`**. Και τώρα γεννιούνται τα ερωτήματα: Πώς δεν θα γίνει μπέρδεμα; Γιατί ο ΗΥ δεν θα πάρει να επεξεργαστεί και τη συνθηματική τιμή όπως όλες τις άλλες; Και αν θέλουμε να δώσουμε αυτήν την τιμή για επεξεργασία;

Στο παράδειγμά μας, θα χρησιμοποιήσουμε την πληροφορία ότι *τα στοιχεία είναι μη μηδενικά*. Σχεδιάζουμε λοιπόν το πρόγραμμά μας με βάση την εξής σύμβαση: *Όταν τελειώσουν τα στοιχεία που θέλουμε να δώσουμε θα πληκτρολογήσουμε την τιμή "`0`"*. Πρόσεξε ότι:

- Το "`0`" είναι δεκτή απάντηση στην εντολή: `cin >> x`.
- Από τις προδιαγραφές του προβλήματος ξέρουμε ότι αποκλείεται να δώσουμε το 0 για επεξεργασία.

Το "0" λέγεται **τιμή-φρουρός** (sentinel value) και αυτή η τεχνική χρησιμοποιείται ευρύτατα από τους προγραμματιστές. Μπορούμε λοιπόν να πούμε ότι η **while** θα ξεκινάει ως εξής:

```
while ( x != sentinel )
```

Τώρα όμως κάναμε μια σοβαρή αλλαγή στη λογική του προγράμματος που είχαμε στα προηγούμενα παραδείγματα: Όταν έρχεται η στιγμή να εκτελεσθεί η **while** πρέπει να έχουμε την πρώτη τιμή της *x*. Η πρώτη ανάγνωση θα πρέπει να γίνεται πριν από την **while**:

```
cin >> x;
while ( x != sentinel )
```

Αλλαγών συνέχεια: Μέχρι τώρα η περιοχή της **while** είχε τη μορφή:

```
{
    cin >> x;
    Επεξεργάσου την τιμή που διάβασες στη x
}
```

Τι θα γίνει αν την αφήσουμε έτσι; Θα διαβάσουμε τη δεύτερη τιμή πριν επεξεργαστούμε την πρώτη. Θα πρέπει λοιπόν να αντιστρέψουμε τη σειρά των εντολών:

```
cin >> x;
while ( x != sentinel )
{
   Επεξεργάσου την τιμή που διάβασες στη x;
   cin >> x;      // Διάβασε την επόμενη τιμή
}
```

Και η τελευταία αλλαγή: Ο μετρητής μας, κάθε στιγμή, θα πρέπει να δείχνει πόσες τιμές διαβάστηκαν για επεξεργασία. Θα πρέπει να ξεκινάει από 0 και όχι από 1. Το παρακάτω πρόγραμμα, Μέση Τιμή 3, είναι μια λύση του προβλήματος, με αυτήν την τεχνική.

```
// πρόγραμμα: Μέση Τιμή 3
#include <iostream>
using namespace std;
int main()
{
   const double sentinel = 0;

   int n;       // Μετρητής των επαναλήψεων. Η τελική
                // τιμή είναι το πλήθος των στοιχείων
   double x;    // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;  // Το μερικό (τρέχον) άθροισμα.
                // Στο τέλος έχει το ολικό άθροισμα.
   double avrg; // Μέση Αριθμητική Τιμή των x (<x>)

   sum = 0;  n = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;            // x == tn
      sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
}
```

Να παρατηρήσουμε ακόμη τα εξής:

1. Μπορεί να αναρωτιέσαι αν μπορούμε να γράψουμε το πρόγραμμα βάζοντας τις εντολές:

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

Πλαίσιο 6.3

Γενικό Σχήμα Επανάληψης με Τιμή-Φρουρό

```
Ετοίμασε ή διάβασε την πρώτη τιμή v
while ( η v δεν είναι φρουρός )
{
   Επεξεργάσου τη v;
   Ετοίμασε ή διάβασε την επόμενη τιμή v
} // while
```

μόνο μια φορά. Ναι, μπορούμε! Δοκίμασε να το κάνεις.

2. Το πρόγραμμα μπορεί να γίνει πιο ευέλικτο αν ο *sentinel* δεν είναι σταθερά αλλά *μεταβλητή που η τιμή της θα δίνεται από το πληκτρολόγιο πριν από τα στοιχεία εισόδου*. Τροποποίησε το πρόγραμμα προς αυτήν την κατεύθυνση.

3. Μπορείς, αντί για μια τιμή-φρουρό, να έχεις μια ολόκληρη περιοχή τιμών. Π.χ. αν στο παράδειγμά μας ξέραμε ότι οι τιμές μας είναι *θετικές* θα μπορούσαμε να γράψουμε το πρόγραμμά μας ως εξής:

```
cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
while ( x > 0 )
{
   n = n + 1;            // x == tn
   sum = sum + x;        // (x == tn) && (sum = Σ(j:1..n)tj)
   cout << " Δώσε θετικό αριθμό - <= 0 για ΤΕΛΟΣ: ";  cin >> x;
} //  while
```

4. Οι τιμές που ελέγχεις με φρουρό δεν είναι απαραίτητο να εισάγονται από το πληκτρολόγιο. Μπορεί να δημιουργούνται από το πρόγραμμα. Μπορεί, για παράδειγμα, να είναι όροι ακολουθίας που ξέρουμε το μαθηματικό της τύπο.

5. Δεν θα μπορούσαμε, αντί να ψάχνουμε για φρουρούς, να δηλώσουμε:

```
char resp;
```

να βάλουμε μια ερώτηση:

```
    cout << " Έχεις άλλες τιμές; (N/O): ");  cin >> resp;
```

και να ελέγχουμε την επανάληψη ως εξής:

```
while ( resp == 'N' )  { . . .
```

Και βέβαια θα μπορούσαμε[1]. (Πρόσεξε ότι στην περίπτωση αυτή δεν φτάνει ο έλεγχος για το `'N'`, αλλά πρέπει να ελέγχεις για ελληνικά, λατινικά, πεζά, κεφαλαία.)

Πάντως η τεχνική της τιμής-φρουρού (και η φιλοσοφία της) είναι πολύ χρήσιμη σε πολλές περιπτώσεις. Το γενικό σχήμα χρήσης το βλέπεις στο Πλ. 6.3.

6.1.2 Επιλεκτική Επεξεργασία

Το πρόβλημα που λύσαμε με τα τρία προγράμματα Μέση Τιμή μπορεί να το συναντήσεις αρκετά συχνά με δεδομένα διαφόρων ειδών, π.χ. μετρήσεις από κάποιο πείραμα, κάποια έρευνα αγοράς, κάποια σφυγμομέτρηση κλπ. Πολύ συχνά όμως, πέρα από τη γενική επεξεργασία που κάνουμε σε όλα τα στοιχεία, θέλουμε να κάνουμε ειδική επεξεργασία σε ένα υποσύνολο τιμών, που επιλέγονται με κάποια κριτήρια. Π.χ.

- Πόσες φορές μέτρησα στο πείραμά μου συχνότητα από 1000 *Hz* μέχρι 1500 *Hz*; Ποιά ήταν η μέση τιμή τους;
- Στη σφυγμομέτρησή μου, πόσοι από αυτούς που στις εκλογές ψήφισαν το κόμμα x, έχουν τελειώσει Πανεπιστήμιο;

[1] Μήπως στην περίπτωση αυτή κάθε τιμή ≠ `'N'` είναι φρουρός στην *resp*; Μήπως;...

- Στην έρευνα αγοράς που έκανα, πόσοι από αυτούς που έχουν πλυντήριο πιάτων κάνουν τα ψώνια τους σε μηνιαία βάση;

Στις περιπτώσεις αυτές το γενικό σχήμα επεξεργασίας είναι το εξής:

```
Κάνε τη γενική επεξεργασία στην τιμή x;
if (για τη x πληρούνται τα κριτήρια επιλογής)
   κάνε την ειδική επεξεργασία στην τιμή x
```

Ας κάνουμε κι εμείς κάτι παρόμοιο: θα τροποποιήσουμε το πρόγραμμα Μέση Τιμή 3 ώστε να μας δίνει επί πλέον:

- το πλήθος,
- το άθροισμα και
- τη μέση τιμή

όσων από τις τιμές που διαβάζει είναι θετικές και μέχρι (και) 10.

Η γενική επεξεργασία είναι αυτή που κάναμε και πιο πριν. Για την ειδική επεξεργασία χρειαζόμαστε: ένα μετρητή (*selN*) για το πλήθος των επιλεγόμενων τιμών, μια μεταβλητή (*selSum*) όπου θα «μαζεύουμε» το άθροισμα των επιλεγόμενων τιμών και μια μεταβλητή (*selAvrg*) για τη μέση τιμή τους. Η επιλογή και η ειδική επεξεργασία γίνεται με την εντολή:

```
      if ( 0 < x && x <= 10 )      // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;      // Αυτά γίνονται μόνο για
         selN = selN + 1;          // τους επιλεγόμενους αριθμούς
      } // if
```

Ολόκληρο το πρόγραμμα Μέση Τιμή 4:

```
// πρόγραμμα: Μέση Τιμή 4
#include <iostream>
using namespace std;
int main()
{
   const double sentinel( 0 );

   int n;          // Μετρητής όλων των στοιχείων
   int selN;       // Μετρητής επιλεγόμενων στοιχείων
   double x;       // Εδώ αποθηκεύουμε κάθε τιμή που διαβάζουμε
   double sum;     // Το άθροισμα όλων των στοιχείων
   double selSum;  // Άθροισμα επιλεγόμενων στοιχείων
   double avrg;    // Μέση Αριθμητική Τιμή όλων των στοιχείων
   double selAvrg; // Μέση Αριθμητική Τιμή επιλεγόμενων στοιχείων

   sum = 0;  n = 0;
   selSum = 0;  selN = 0;
   cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   while ( x != sentinel )
   {
      n = n + 1;               // Αυτά γίνονται για
      sum = sum + x;           // όλους τους αριθμούς
      if ( 0 < x && x <= 10 )  // 'Ελεγχος - Επιλογή
      {
         selSum = selSum + x;       // Αυτά γίνονται μόνο για
         selN = selN + 1;           // τους επιλεγόμενους αριθμούς
      } // if
      cout << " Δώσε αριθμό - 0 για ΤΕΛΟΣ: ";  cin >> x;
   } // while
   cout << " Διάβασα " << n << " αριθμούς" << endl;
   if ( n > 0 )
   {
      avrg = sum / n;
      cout << " ΑΘΡΟΙΣΜΑ = " << sum << "  <x> = " << avrg << endl;
   }
   cout << " Διάλεξα " << selN << " αριθμούς ε (0,10]" << endl;
   if ( selN > 0 )
   {
```

```
      selAvrg = selSum/selN;
      cout << " ΑΘΡΟΙΣΜΑ = " << selSum
           << "  <x> = " << selAvrg << endl;
   }
}
```

Εδώ βλέπεις μια **if** μέσα σε μια **while**.

6.2 * Αναλλοίωτες και Τερματισμός

Αντιγράφουμε από το παράδειγμα της §6.1:

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

Είναι σωστό αυτό το πρόγραμμα; Δηλαδή: αν διαβαστούν n (> 0) τιμές t_1 ... t_n, θα ισχύει, μετά την εκτέλεση της **while**, η *sum* == $\sum_{j=1}^{n} t_j$;

Η τιμή του μετρητή m ξεκινάει από 1 και αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή επανάληψης. Άρα την πρώτη φορά που δεν θα ισχύει η `m <= n` –και θα σταματήσει η εκτέλεση της **while**– θα έχουμε m == n + 1. Αν αποδείξουμε ότι τότε θα ισχύει η *I*: *sum* == $\sum_{j=1}^{m-1} t_j$ θα έχουμε αποδείξει αυτό που θέλαμε.

Θα το αποδείξουμε με τη **μέθοδο της μαθηματικής επαγωγής**[2].

Βασικό βήμα: Αρχικώς, πριν αρχίσει η εκτέλεση της **while**, έχουμε, από τις εντολές προετοιμασίας: (*sum* == 0) && (m == 1). «Ταιριάζει» αυτή με την *sum* == $\sum_{j=1}^{m-1} t_j$; Στην *sum* έχουμε το άθροισμα των m – 1 πρώτων μελών του συνόλου { t_1 ... t_N } που έχουν ήδη πληκτρολογηθεί. Αφού αρχικώς δεν έχει πληκτρολογηθεί κάποια τιμή, η *sum* πρέπει να έχει τιμή 0. Αυτό είναι και το νόημα του $\sum_{j=1}^{1-1} t_j = \sum_{j=1}^{0} t_j$. Μπορούμε λοιπόν να πούμε ότι αρχικώς, πριν από τη **while**, ισχύει η *I*.

Επαγωγικό βήμα: Θα πρέπει να αποδείξουμε ότι αν η *I* ισχύει για κάποιο m θα ισχύει και για την επόμενη τιμή της m (δηλαδή: m+1). Στο πρόγραμμά μας όμως πηγαίνουμε στην επόμενη τιμή της m αφού διαβάσουμε, προσθέσουμε και μετρήσουμε την επόμενη τιμή. Θα πρέπει λοιπόν να αποδείξουμε ότι: *αν η I:* (*sum* == $\sum_{j=1}^{m-1} t_j$) *ισχύει όταν αρχίσουν να εκτελούνται οι επαναλαμβανόμενες εντολές θα ισχύει και μετά το τέλος της εκτέλεσής τους*, δηλαδή:

```
// sum == Σ(j:1..m-1)tj
   cin >> x;
   sum = sum + x;
   m = m + 1;
// sum == Σ(j:1..m-1)tj
```

[2] Δες για παράδειγμα Σ. Ανδρεαδάκη κ.ά. «ΑΛΓΕΒΡΑ Β′ ΛΥΚΕΙΟΥ», ΟΕΔΒ 1994.

Αυτό ξέρουμε να το αποδείξουμε: Για να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$ μετά τη `m = m + 1` θα πρέπει πριν από αυτήν να ισχύει η $sum == \sum_{j=1}^{(m+1)-1} t_j$ ή αλλιώς: $sum == \sum_{j=1}^{m} t_j$. Έχουμε δηλαδή:

```
   cin >> x;
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Για να ισχύει η $sum == \sum_{j=1}^{m} t_j$ μετά την `sum = sum + x` θα πρέπει πριν από αυτήν να ισχύει η: $sum + x == \sum_{j=1}^{m} t_j$:

```
   cin >> x;
// sum + x == Σ(j:1..m)tⱼ == Σ(j:1..m-1)tⱼ + tₘ }
   sum = sum + x;
// sum == Σ(j:1..m)tⱼ)
   m = m + 1;
// sum == Σ(j:1..m-1)tⱼ
```

Αφού, όπως είπαμε, όταν η `cin >> x` εκτελείται για m-οστή φορά διαβάζεται η t_m, μπορούμε να πούμε ότι μετά την εκτέλεσή της θα έχουμε: $x == t_m$. Μαζί με το ότι $\sum_{j=1}^{m} t_j == \sum_{j=1}^{m-1} t_j + t_m$ καταλήγουμε στο ότι πριν από τη "`cin >> x`" θα πρέπει να ισχύει η $sum == \sum_{j=1}^{m-1} t_j$.

Αυτή όμως ισχύει, αφού είναι η προϋπόθεσή μας.

Τι σημαίνουν τα παραπάνω με απλά λόγια;

- Η I ισχύει αρχικώς (βασικό βήμα), για $m == 1$.
- Αφού αποδείξαμε (επαγωγικό βήμα) ότι αν η I ισχύει αρχικώς[3] και εκτελεσθούν οι επαναλαμβανόμενες εντολές, που αυξάνουν την τιμή της m, η I ισχύει και πάλι, η I ισχύει και για $m == 2$.
- Αφού η I ισχύει για $m == 2$, λόγω του επαγωγικού βήματος, θα ισχύει και για $m == 3$

κ.ο.κ. Γενικώς, *η I ισχύει για οποιαδήποτε τιμή πάρει η m με τις επαναλαμβανόμενες εντολές*. Βλέπουμε λοιπόν ότι η εκτέλεση των επαναλαμβανόμενων εντολών αφήνει την I αναλλοίωτη· γι' αυτό και ονομάζεται **αναλλοίωτη της επανάληψης** (repetition invariant).

Παρατηρήσεις ▶

1. Για να αποδείξουμε την ορθότητα μιας **while** θα *πρέπει να έχουμε την αναλλοίωτη*. Πού θα τη βρούμε; Στα προγράμματα που γράφουμε εμείς δεν είναι δύσκολο να τη έχουμε: είναι, στην πραγματικότητα, η λογική περιγραφή της μεθόδου που χρησιμοποιούμε για να λύσουμε το πρόβλημά μας. Αν μας δώσουν μια **while** και μας ζητήσουν να βρούμε την αναλλοίωτη τα πράγματα είναι δύσκολα! Αν όμως υπάρχουν προδιαγραφές τα πράγματα είναι καλύτερα. Διάβασε όμως παρακάτω...
2. Όταν κάνουμε αποδείξεις από το τέλος προς την αρχή φτάνουμε σε κάτι σαν:

```
   while ( S )  E;
// Q
```

Εδώ τι κάνουμε; Αν μπορέσουμε να φέρουμε την Q στη μορφή (`!`S) `&&` Q_I, προσπαθούμε να αποδείξουμε ότι η Q_I είναι αναλλοίωτη της **while**. Φυσικά, η Q_I θα πρέπει να ισχύει και πριν

[3] Οι εντολές "`sum = 0; m = 1;`", που δίνονται πριν από την εντολή **while**, σκοπό έχουν να εξασφαλίσουν ότι ισχύει η I πριν από τη **while**.

από τη **while**. Αυτή η «συνταγή» μπορεί να μη φαίνεται και τόσο εύκολη για ορισμένες περιπτώσεις, όπως π.χ. η μετρούμενη επανάληψη· θα τα πούμε παρακάτω.
3. Πρόσεξε ακόμη το εξής: αν έχουμε **while** (*S*) *E*, με αναλλοίωτη *I*, οι *E* θα εκτελεσθούν μόνον αν ισχύει η *S*. Δηλαδή, αυτό που έχεις να αποδείξεις είναι:

```
// I && S
   E
// I  ◀
```

Τώρα, μπορούμε να διατυπώσουμε και συμβολικώς τον συμπερασματικό κανόνα της **while**:

$$\frac{I \,\&\&\, S\,\{E\}\,I}{I\,\{\textbf{while}(S)\;E\}\,(!S)\,\&\&\,I}$$

Και όταν κάνουμε αποδείξεις από το τέλος προς την αρχή, ποια συνθήκη θα πρέπει να ισχύει πριν από τη **while**; Ποια άλλη; Η *αναλλοίωτη*!

Όταν έχουμε επαναληπτικές εντολές, για να αποδείξουμε ότι το πρόγραμμά μας είναι ***ολικώς*** σωστό, *θα πρέπει να αποδείξουμε ότι η εκτέλεση όλων των επαναληπτικών εντολών θα τερματισθεί*. Πώς γίνεται αυτό; Ας έρθουμε στο παράδειγμά μας: Θεωρούμε την ακολουθία των διαδοχικών τιμών της διαφοράς $n - m$ που είναι ακέραιος αριθμός.

- Αφού η **while** εκτελείται μόνο όταν $m \le n$ έχουμε πάντοτε $n - m \ge 0$, δηλαδή έχουμε μια ακολουθία *φυσικών* αριθμών.
- Η ακολουθία αυτή είναι γνησίως φθίνουσα, αφού κάθε τιμή της *m* είναι κατά 1 μεγαλύτερη από την προηγούμενη.

Τα μαθηματικά μας λένε ότι *δεν είναι δυνατόν να έχουμε γνησίως φθίνουσα ακολουθία*[4] *φυσικών αριθμών με άπειρο πλήθος όρων*. Επειδή δεν είναι δυνατόν η ακολουθία που δημιουργεί η **while** να παραβιάσει αυτό το θεώρημα, η εκτέλεσή της θα σταματήσει κάποτε.

Αυτός είναι ο τρόπος που συνήθως χρησιμοποιούμε για να αποδείξουμε τον τερματισμό. Μερικές φορές δεν είναι και τόσο εύκολο να βρούμε τη γνησίως φθίνουσα ακολουθία φυσικών αριθμών.

6.2.1 * Παραδείγματα

Παράδειγμα 1 ✍

Ας υποθέσουμε ότι η C++ δεν μας δίνει την ακέραιη διαίρεση και την "**%**". Θέλουμε ένα πρόγραμμα που θα διαβάζει δύο ακέραιους *d1*, *d2* –ο πρώτος (διαιρετέος) μη αρνητικός, ο δεύτερος (διαιρέτης) θετικός– και θα υπολογίζει και θα τυπώνει το πηλίκο και το υπόλοιπο της ακέραιης διαίρεσης *d1*:*d2*.

Πρώτα οι προδιαγραφές: Για το υπόλοιπο *y*, σίγουρα θυμάσαι ότι, θα πρέπει να έχουμε $0 \le y < d2$. Για το πηλίκο *p* τι θα έχουμε; Τι άλλο από αυτό που σου μάθανε ως δοκιμή της διαίρεσης: $d1 == p \cdot d2 + y$:
Προϋπόθεση: $(d1 \ge 0)$ **&&** $(d2 > 0)$
Απαίτηση: $(0 \le y < d2)$ **&&** $(d1 == p \cdot d2 + y)$

Ο αλγόριθμος θα πρέπει να σου είναι γνωστός από το Δημοτικό Σχολείο (από τότε που σου λέγανε ότι η διαίρεση είναι πολλές αφαιρέσεις): όσο ο *d2* είναι μικρότερος από ή ίσος με *d1* (ο *d2* χωράει στον *d1*) αφαιρούμε από τον *d1* τον *d2* και μετρούμε την αφαίρεση. Τελικώς το πλήθος των αφαιρέσεων θα είναι το πηλίκο ενώ ότι έχει μείνει από τις διαδοχικές αφαιρέσεις είναι το υπόλοιπο. Δηλαδή:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
```

[4] Όπου λέμε «ακολουθία» εννοούμε ακολουθία με *πεπερασμένο* πλήθος όρων.

```
   while ( d2 <= y )
   {
      y = y - d2;
      p = p + 1;
   } // while
// (0 ≤ y < d2) && (d1 == p*d2 + y)
```

Ας αποδείξουμε τώρα, από το τέλος προς την αρχή, την ορθότητα του προγράμματός μας.

Όπως είπαμε, όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!`*S* `&&` *I*. Στην περίπτωσή μας

- `!`*S* είναι η `!(d2 <= y)`, ή αλλιώς: $d2 > y$.
- *I* θα πρέπει να είναι αναλλοίωτη της επανάληψης.

Πώς θα βρούμε την αναλλοίωτη; Παρατηρούμε το εξής: Η απαίτηση που έχουμε για μετά τη **while** είναι: $(0 \le y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$ ή αλλιώς: $(0 \le y)$ `&&` $(y < d2)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αν από αυτήν ξεχωρίσουμε την $y < d2$, που είναι η `!`*S*, το υπόλοιπο θα πρέπει να είναι η αναλλοίωτη. Θα πρέπει λοιπόν να αποδείξουμε ότι η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ είναι η αναλλοίωτη της **while**.

1. *Βασικό βήμα*: Η αναλλοίωτη θα πρέπει να ισχύει πριν από τη **while**. Θα πρέπει δηλαδή να έχουμε:

```
// (d1 ≥ 0) && (d2 > 0)
   y = d1;
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ μετά την `p = 0` θα πρέπει να έχουμε πριν από αυτήν $(0 \le y)$ `&&` $(d1 == 0{\cdot}d2 + y)$ ή αλλιώς:

$$(0 \le y) \text{ \&\& } (d1 == y)$$

```
   y = d1;
// (0 ≤ y) && (d1 == y) }
   p = 0;
// (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == y)$ μετά την `y = d1` θα πρέπει να ισχύει πριν από αυτήν η: $(0 \le d1)$ `&&` $(d1 == d1)$ ή απλούστερα: $d1 \ge 0$ που συνάγεται από την προϋπόθεση.

2. *Επαγωγικό βήμα*: Θα αποδείξουμε ότι:

```
   // (d2 ≤ y) && (0 ≤ y) && (d1 == p*d2 + y)
      y = y - d2;
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να έχουμε μετά την `p = p + 1` την $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$ θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$:

```
      y = y - d2;
   // (0 ≤ y) && (d1 == (p+1)*d2 + y)
      p = p + 1;
   // (0 ≤ y) && (d1 == p*d2 + y)
```

Για να ισχύει η $(0 \le y)$ `&&` $(d1 == (p+1){\cdot}d2 + y)$ μετά την `y = y - d2` θα πρέπει να έχουμε πριν από αυτήν: $(0 \le y - d2)$ `&&` $(d1 == (p+1){\cdot}d2 + y - d2)$ ή αλλιώς: $(d2 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$. Αυτή όμως συνάγεται από την $(d2 \le y)$ `&&` $(0 \le y)$ `&&` $(d1 == p{\cdot}d2 + y)$.

Άρα το πρόγραμμά μας είναι *μερικώς σωστό*.

Να δούμε τώρα αν είναι και *ολικώς σωστό*, αν δηλαδή τερματίζεται η εκτέλεσή του. Ας θεωρήσουμε την ακολουθία των διαδοχικών τιμών της *y*. Αυτές είναι:

- ακέραιες,
- μη αρνητικές, αφού η `y = y - d2` εκτελείται μόνον αν $d2 \le y$ και
- κάθε μια είναι *μικρότερη από την προηγούμενή της* αφού έχουμε από την προϋπόθεσή μας ότι $d2 > 0$.

Έχουμε δηλαδή μια γνησίως φθίνουσα ακολουθία φυσικών αριθμών. Αυτή αποκλείεται να έχει άπειρο πλήθος όρων· άρα η εκτέλεση της **while** θα τερματισθεί κάποτε. Δηλαδή το πρόγραμμά μας *είναι ολικώς σωστό*. Καμάρωσέ το:

```
#include <iostream>
using namespace std;
int main()
{
   int d1, d2, p, y;

   cin >> d1 >> d2;
   if ( d1 >= 0 && d2 > 0 )
   { // (d1 ≥ 0) && (d2 > 0)
      y = d1;
      p = 0;
      while ( d2 <= y )  // I: (0 ≤ y) && (d1 == p*d2 + y)
      {
         y = y - d2;
         p = p + 1;
      } // while
   // (0 ≤ y < d2) && (d1 == p*d2 + y)
      cout << " Πηλίκο: " << p << "    Υπόλοιπο: " << y << endl;
      cout << " Η C++ δίνει: " << endl;
      cout << " Πηλίκο: " << (d1 / d2)
           << "    Υπόλοιπο: " << (d1 % d2) << endl;
   }
   else
   // false
      cout << " Λάθος! " << endl;
}
```

Παρατήρηση ▶

Τι θα πει διαίρεση δια 0 (μηδέν); Θα πει ότι η ακολουθία μας *δεν είναι γνησίως φθίνουσα* οπότε η απόδειξη του τερματισμού δεν γίνεται. Ούτε και τερματισμός της εκτέλεσης της **while** γίνεται! Να λοιπόν που μια από τις «κακοτοπιές» που προσέχουμε (διαίρεση δια 0) έχει σχέση με μια **while** που δεν τελειώνει ποτέ[5]! Έτσι, από περιέργεια, αξίζει τον κόπο να βγάλεις την **if**, που ελέγχει την προϋπόθεση, και να δώσεις $d2 = 0$. ◀

Παράδειγμα 2

Ας πούμε τώρα ότι η C++ *δεν* μας δίνει συνάρτηση για την τετραγωνική ρίζα θετικού αριθμού. Ας γράψουμε ένα πρόγραμμα που να την υπολογίζει. Πώς; Τα μαθηματικά[6] μας λένε ότι: αν $x_0 > 0$ και $x_0^2 > a$ τότε η ακολουθία:

$$\{ n: \mathbb{N}^+, x_n: \mathbb{R} \bullet x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}} \right) \}$$

συγκλίνει προς τη $\sqrt{a}$.[7]

Τι προδιαγραφές θα έχουμε; Ή αλλιώς: πόσο καλή προσέγγιση θα έχουμε; Ας πούμε ότι θέλουμε το σχετικό σφάλμα να είναι μικρότερο από κάποιο $\varepsilon > 0$:

$$\frac{\left| x - \sqrt{a} \right|}{\sqrt{a}} < \varepsilon \qquad (1)$$

---

[5] Φυσικά, ο υπολογιστής σου προσέχει ότι ο διαιρέτης είναι 0 πριν κάνει τη διαίρεση και διακόπτει την εκτέλεση του προγράμματος με το σχετικό μήνυμα λάθους.

[6] Η ακολουθία προκύπτει όταν προσπαθούμε να βρούμε μια προσέγγιση της λύσης της εξίσωσης $x^2 - A = 0$ με τη μέθοδο Newton - Raphson.

[7] Δες την απόδειξη στο `http://en.wikipedia.org/wiki/ Methods_of_computing_square_roots` ή στο (Κάππος 1962), ας πούμε.

Μπορούμε εύκολα να γράψουμε μια **while** που να υπολογίζει τους όρους αυτής της ακολουθίας:

```
x = αρχική τιμή;
while ( S )
   x = 0.5*(x + a/x);
```

Τι θα βάλουμε ως αρχική τιμή *x*; Κάποια τιμή που να μας δίνει: $(x > 0)$ **&&** $(x^2 > a)$. Να μια σκέψη:

```
if (a <= 1)  x = a + 1;  else  x = a;
```

αλλά θα πρέπει να αποδείξουμε ότι είναι σωστή:

```
// a > 0
   if (a <= 1)  x = a + 1;  else  x = a;
// (x > 0) && (x² > a) }
```

Για να ισχύει αυτό, αρκεί να ισχύουν οι:

```
// (a > 0) && (a <= 1)      // (a > 0) && (a > 1)
   x = a + 1;          και      x = a;
// (x > 0) && (x² > a)      // (x > 0) && (x² > a)
```

Η απόδειξη των παραπάνω είναι τετριμμένη.

Η *S* ποια θα είναι; Προφανώς η άρνηση της (1), δηλαδή η:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} \geq \varepsilon$$

ή:

$$\frac{\left|x-\sqrt{a}\right|}{\sqrt{a}} = \frac{\left|x-\sqrt{a}\right|\left|x+\sqrt{a}\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} = \frac{\left|x^2-a\right|}{\sqrt{a}\left|x+\sqrt{a}\right|} \approx \frac{\left|x^2-a\right|}{2a} \geq \varepsilon$$

(στο τελευταίο βήμα χρησιμοποιήσαμε την προσέγγιση: $x \approx \sqrt{a}$).

Τι θα πάρουμε ως ε; Για να σκεφτούμε λίγο παραπάνω την τελευταία σχέση που γράφεται:

$$\left|\frac{x^2}{a}-1\right| \geq 2\varepsilon$$

Το ιδανικό θα ήταν να μηδενίσουμε το αριστερό μέρος. Αλλά ξέρουμε ότι το αριστερό μέρος θα είναι μηδέν αν η διαφορά του x^2/a από το 1 είναι < ε`double` (**DBL_EPSILON**). Επομένως, το καλύτερο που μπορούμε να πάρουμε είναι όταν ε = ½**DBL_EPSILON**. Μπορούμε λοιπόν να γράψουμε:

```
if ( a <= 1 )  x = a + 1;  else  x = a;
while ( fabs(x*x - a ) >= DBL_EPSILON*a)
   x = 0.5*(x + a/x);
```

Δεν θα ψάξουμε να βρούμε την αναλλοίωτη; *Δεν μας χρειάζεται!* Όλοι οι υπολογισμοί αποβλέπουν στο να ανατρέψουμε τη συνθήκη της **while**.

Είναι σίγουρος ο τερματισμός αυτής της **while**; Αφού τα μαθηματικά μας εγγυώνται ότι η ακολουθία των διαδοχικών τιμών της *x* συγκλίνει στην $\sqrt{a}$ έχουμε ότι για κάθε $\delta > 0$ υπάρχει *N* τέτοιο ώστε για κάθε $n > N$ να έχουμε $|x_n - \sqrt{a}| < \delta$. Αν λοιπόν πάρουμε $\delta = \varepsilon\sqrt{a}$, τοτε από κάποιο *n* και μετά θα έχουμε: $|x_n - \sqrt{a}| < \varepsilon\sqrt{a}$, που όπως είδαμε παραπάνω ισοδυναμεί με: $|x^2 - a| < 2\varepsilon a$. Αυτή όμως είναι η **!**$(|x^2 - a| \geq 2\varepsilon a)$, δηλαδή η άρνηση της συνθήκης της **while**.

Αλλά, προσοχή: Η Ανάλυση μας εγγυάται τη σύγκλιση με την προϋπόθεση ότι $a > 0$. Αν $a < 0$ –μπορείς να το δεις με μερικές δοκιμές– η **while** (και το πρόγραμμα) δεν τελειώνει ποτέ! Να λοιπόν άλλη μια περίπτωση (μετά τη διαίρεση δια μηδέν) όπου αυτό που λέγαμε «κακοτοπιά» έχει να κάνει με τον τερματισμό μιας **while**. Η C++ προσεγγίζει την τετραγωνική ρίζα (*sqrt*) με τον τρόπο που είδαμε παραπάνω, αλλά αν δώσεις αρνητικό όρισμα

στην *sqrt*, αντί να κολλήσει σε μια αέναη επανάληψη, προτιμάει να κόψει την εκτέλεση του προγράμματος.

Εδώ είδαμε παράδειγμα προγράμματος στο οποίο τα μαθηματικά μας εγγυώνται και τη μερική και την ολική ορθότητα. Και όλα αυτά που γράφουμε τι είναι; *Προσεκτική υλοποίηση αυτών που μας λένε τα μαθηματικά!*

Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
#include <cmath>
using namespace std;
int main()
{
   double a, x;

   cin >> a;
   if ( a > 0 )
   {// a > 0
      if ( a <= 1 )  x = a + 1;  else  x = a;
      while ( fabs(x*x - a) >= DBL_EPSILON*a )
         x = 0.5*(x + a/x);
   // |x² - a| < εa
      cout.precision(16);
      cout << x << "  " << sqrt(a) << endl;
   }
   else
      cout << " μόνον θετικούς παρακαλώ" << endl;
}
```

Παράδειγμα 3

Πρόβλημα: Θέλουμε ένα πρόγραμμα που θα διαβάσει από το πληκτρολόγιο n (> 0) τιμές t_k, k = 1, 2,...n –ας πούμε τύπου `int`– και στο τέλος θα μας δώσει τη μεγαλύτερη από αυτές και τη σειρά (*kmax*) με την οποία πληκτρολογήθηκε. Πριν από τους αριθμούς θα διαβάζει την τιμή της n.

Σε μια θέση της μνήμης, *tmax*, κρατούμε τη μέγιστη τιμή από όσες έχουμε διαβάσει και σε μια άλλη, *kmax*, τη σειρά της. Να οι προδιαγραφές μας:

Προϋπόθεση: $n > 0$

Απαίτηση: $1 \le kmax \le n$ && $tmax == t_{kmax}$ && $\forall j: 1..n \bullet t_j \le tmax$

Ας δούμε πώς μπορούμε να λύσουμε το πρόβλημά μας.

Διαβάζουμε την πρώτη τιμή, που προφανώς είναι και η μέγιστη μέχρι στιγμής:

```
   cin >> t;  // t == t1 }
   k = 1;  tmax = t;  kmax = k;
```

Τώρα διαβάζουμε τη δεύτερη τιμή. Αν είναι μεγαλύτερη από την πρώτη, που τη θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη δεύτερη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη την πρώτη:

```
   cin >> t;   // t == t2
   k = k + 1;  // == 2
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Στη συνέχεια διαβάζουμε την τρίτη τιμή. Αν είναι μεγαλύτερη από αυτήν που θεωρούμε μέγιστη μέχρι τώρα, από εδώ και πέρα θα θεωρούμε μέγιστη τη τρίτη. Αλλιώς, καλά κάνουμε και θυμόμαστε ως μέγιστη αυτήν που είχαμε μέχρι τώρα:

```
   cin >> t;   // t == t3
   k = k + 1;  // == 3
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
```

Όπως βλέπεις, διαχειριζόμαστε τη 2η και την 3η τιμή με τον ίδιο ακριβώς τρόπο και με τον ίδιο τρόπο θα διαχειριστούμε και τις υπόλοιπες. Στην 1η έχουμε διαφορά. Μπορούμε

λοιπόν να ξεκινήσουμε, όπως είδαμε, με την πρώτη τιμή και να βάλουμε τα υπόλοιπα σε μια **while**:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if (t > tmax)  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

Όταν τελειώσει η εκτέλεση της **while** θα ισχύει η `!(k <= n)`, δηλαδή η $k > n$. Αφού όμως η k αυξάνεται κατά 1 κάθε φορά που εκτελείται η περιοχή της επανάληψης και αφού $n > 0$ θα έχουμε ακριβέστερα: $k == n + 1$. Από αυτό και από την απαίτηση προσπαθούμε να μαντέψουμε την αναλλοίωτη της επανάληψης: βάζουμε στην απαίτηση όπου n το k–1 και παίρνουμε:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j\text{: } 1..\ k\text{–}1 \bullet t_j \le tmax$$

Αν αποδείξουμε ότι αυτή είναι η αναλλοίωτη *I*, τότε σίγουρα μετά τη **while** θα έχουμε την απαίτηση. Η απόδειξη δεν είναι δύσκολη αλλά είναι αρκετά μακροσκελής και την αφήνουμε για άσκηση (6-16).

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  t, tmax;
   int  n, k, kmax;

   cin >> n;
   if ( n > 0 )
   {// n > 0
      cin >> t;     // t == t1
      k = 1;
      tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
   // 1 ≤ kmax ≤ n && tmax == tkmax && ∀j:1..n • tj ≤ tmax
      cout << " Μέγιστη τιμή: " << tmax
           << "  (" << kmax << "η) " << endl;
   }
   else
      cout << " το πλήθος πρέπει να είναι θετικό" << endl;
}
```

Στο Πλ. 6.4 βλέπεις το γενικό σχήμα προγράμματος για τον υπολογισμό μεγίστου. Για να υπολογίσεις το ελάχιστο άλλαξε τη συνθήκη στην **if**:

```
if ( t < tmin ) ...
```

Μερικές φορές, όπως θα δεις στη συνέχεια, δεν είναι εύκολο να επεξεργαστείς χωριστά την πρώτη τιμή. Γράφουμε λοιπόν τη **while** έτσι ώστε να επεξεργάζεται όλες τις τιμές (k από 1 μέχρι n). Τώρα όμως υπάρχει το εξής πρόβλημα: την πρώτη φορά που θα εκτελεσθεί η **if** ($t > tmax$) τι τιμή θα έχει η *tmax*;

Η *tmax* θα πρέπει να έχει τέτοια τιμή ώστε:

- να αλλάξει οπωσδήποτε –όποια και αν είναι η πρώτη τιμή της t– και έτσι να πάρει μια ρεαλιστική τιμή,
- να μην υπάρχει πιθανότητα να θεωρηθεί ως μια από τις τιμές που επεξεργαζόμαστε.

Δηλαδή, πρέπει να βάλουμε ως αρχική τιμή της *tmax* μια *απίθανα μικρή τιμή*, π.χ. το $-\infty$!

«Πλην άπειρο και πράσινα άλογα!» θα σκεφτείς, «Γίνονται τέτοια πράγματα;» Δεν γίνονται βέβαια, αλλά για κάθε πρόβλημα που έχεις να λύσεις, είναι πολύ πιθανό να μπορείς να βρεις μια απίθανα μικρή τιμή. Π.χ.

- αν οι τιμές που μελετάς είναι θετικές, μια απίθανα μικρή τιμή είναι το 0 ή το -1 (ή οποιοσδήποτε αρνητικός),
- αν ξέρεις ότι οι τιμές που επεξεργάζεσαι είναι μεταξύ 150 και 250, μια απίθανα μικρή τιμή μπορεί να είναι το 0 ή το 10 κλπ.

Στο παράδειγμα που δώσαμε παραπάνω, αν ξέραμε επιπλέον ότι *όλες οι τιμές που διαβάζουμε είναι θετικές*, θα μπορούσαμε να γράψουμε:

```
tmax = -1;
k = 1;
while ( k <= n )
{
   cin >> t;  // t = tk
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Φυσικά, αν ψάχνεις για ελάχιστη τιμή θα ξεκινήσεις από μια *απίθανα μεγάλη τιμή* ($+\infty$).

Πλαίσιο 6.4

Εύρεση Μέγιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmax = t;  kmax = k;
k = k + 1;  // == 2
while ( συνθήκη )
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t > tmax )  {  tmax = t;  kmax = k;  }
   k = k + 1;
} // while
```

Εύρεση Ελάχιστης Τιμής

```
Δημιούργησε ή διάβασε την πρώτη τιμή t1 στην t
k = 1;  tmin = t;  kmin = k;
k = k + 1;  // == 2
while (συνθήκη)
{
   Δημιούργησε ή διάβασε την k-οστή τιμή tk στην t
   if ( t < tmin )  {  tmin = t;  kmin = k;  }
   k = k + 1;
} // while
```

6.3 Η Μετρούμενη Επανάληψη

Η **μετρούμενη επανάληψη** είναι πολύ συνηθισμένη και απλή. Στην §6.1 είδαμε την απλούστερη μορφή της:

```
m = 1;
while ( m <= n )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m + 1;  }
```

Από όσα μάθαμε μέχρι τώρα, αν οι «άλλες επαναλαμβανόμενες εντολές» δεν αλλάζουν την τιμή της *m*, θα εκτελεσθούν:

- 0 φορές αν $n < 1$. Η τιμή της *m* δεν θα αλλάξει.
- *n* φορές αν $n \geq 1$. Στο τέλος η τιμή της *m* θα είναι $n + 1$.

Όταν οι «άλλες επαναλαμβανόμενες εντολές» εκτελούνται για *k*-οστή φορά έχουμε $m == k$.

Η *m* είναι η **μεταβλητή ελέγχου** (control variable) της επανάληψης.

Πολλοί προτιμούν μια άλλη μορφή για την ίδια δουλειά:

```
m = n;
while ( m >= 1 )  {  άλλες επαναλαμβανόμενες εντολές
                     m = m - 1;    }
```

ή

```
m = n;
while (m > 0)  {  άλλες επαναλαμβανόμενες εντολές
                  m = m - 1;    }
```

διότι η ακολουθία τιμών της *m* είναι ακριβώς αυτή που χρησιμοποιήσαμε για να αποδείξουμε τον τερματισμό.

Ας δούμε όμως την εξής γενίκευση: αν η(οι) εντολή (-ές) *E* δεν μεταβάλλουν τις τιμές των *m*, *Ma*, *Mt*, *b* (οποιουδήποτε αριθμητικού τύπου) και αν $b > 0$, τότε κατά την εκτέλεση των

```
m = Ma;
while ( m <= Mt )
{
   E;
   m = m + b;
}
```

η(οι) εντολή(-ές) *E* θα εκτελεσθεί(-ούν) *n* φορές όπου:

- $n = 0$ αν $Ma > Mt$,
- $n = Trunc[(Mt - Ma + b)/b]$ φορές αν $Ma \leq Mt$.

Κατά την *k*-οστή φορά που θα εκτελεσθεί(-ούν) η(οι) *E*, η *m* θα έχει τιμή $Ma + (k-1)\cdot b$. Το *b* λέγεται **βήμα** (step).

Ας δούμε ένα

Παράδειγμα

Το παρακάτω πρόγραμμα μας δίνει, για τα πρώτα 15 sec, την απόσταση που διανύει, ανά 0.5 *sec*, ένα κινητό που ξεκινάει από την ηρεμία και κινείται με σταθερή επιτάχυνση 10 m/sec^2.

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf(ios::fixed,ios::floatfield);
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
```

```
   step = 0.5; // sec
   t = 0.0;
   while ( t <= 15.0 )
   {
      x = 0.5*a*t*t;
      cout.width(6);  cout.precision(1);  cout << t << "     ";
      cout.width(7);  cout.precision(2);  cout << x << endl;
      t = t + step;
   } // while
}
```

Το πρόγραμμα αυτό θα μας δώσει τα αποτελέσματα σε μορφή πίνακα:

```
t          x
   sec        m
   0.0        0.00
   0.5        1.25
   1.0        5.00
  . . .      . . .
  14.5     1051.25
  15.0     1125.00
```

6.4 Η Εντολή for

Η C++ έχει μια εντολή, τη **for**, που παραδοσιακά χρησιμοποιείται κυρίως για να γράφουμε μετρούμενες επαναλήψεις. Π.χ. η:

```
   m = Ma;
   while ( m <= Mt )
   {
      E;
      m = m + b;      // b > 0
   }
```

που είδαμε στην προηγούμενη παράγραφο μπορεί να γραφεί:

```
   for ( m = Ma;  m <= Mt;  m = m + b )
   {
      E;
   }
```

ενώ η

```
   m = Ma;
   while ( m >= Mt )
   {
      E;
      m = m - b;      // b > 0
   }
```

μπορεί να γραφεί:

```
   for ( m = Ma;  m >= Mt;  m = m - b )
   {
      E;
   }
```

Η μεταβλητή ελέγχου *m* πρέπει να είναι αριθμητικού ή απαριθμητού τύπου. Συνηθέστερα χρησιμοποιούμε ακέραιο ή απαριθμητό τύπο.

Τα *Ma, Mt, b* μπορεί γενικώς να είναι παραστάσεις. Παρ' όλο που δεν απαγορεύεται, απόφευγε όταν χρησιμοποιείς τη **for** να γράφεις επαναλαμβανόμενες εντολές (*E*) που να τροποποιούν οποιαδήποτε από τις *m, Ma, Mt, b*.

- ***Είναι καλό οι τιμές των Ma, Mt, b να καθορίζονται όταν αρχίζει η εκτέλεση της for και να μη μεταβάλλονται οι τιμές τους στη διάρκεια της εκτέλεσης της for. Η τιμή της μεταβλητής ελέγχου θα πρέπει να μεταβάλλεται μόνο από την*** `m = m ± b`.

Και γιατί όλα αυτά; Με τους παραπάνω περιορισμούς, τα δύο σχήματα επανάληψης με **for** που γράψαμε πιο πάνω δεν χρειάζονται απόδειξη για τον τερματισμό. Χωρίς αυτούς τους περιορισμούς η απόδειξη ορθότητας μπορεί να γίνει αρκετά πολύπλοκη. Γράφουμε λοιπόν μια **while** (και όχι **for**) για να ξέρουμε τι μας περιμένει.

Όπως η **while** έτσι και η **for**, περιμένει *μια επαναλαμβανόμενη εντολή*· αν θέλουμε να βάλουμε περισσότερες, όπως κάναμε στην **if** και στη **while**, τις «συσκευάζουμε» με ένα άγκιστρο (`{`) στην αρχή και ένα άγκιστρο (`}`) στο τέλος σε μια *σύνθετη εντολή*.

Δες πώς θα μπορούσε να γραφεί η επανάληψη του Μέση Τιμή 1 (ή 2) με χρήση της **for**:

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

Ας δούμε μερικά παραδείγματα ακόμη:

Παράδειγμα 1

Η μεταβλητή ελέγχου της **for** μπορεί να είναι και τύπου `bool`. Ξαναγράφουμε το πρόγραμμα της §4.3 που δημιουργεί τους αληθοπίνακες:

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;
   int   iP, iQ;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q " << endl;
   for ( iP = 0;  iP <= 1;  iP = iP+1 )
   {
      P = static_cast<bool>(iP);
      for (iQ = 0;  iQ <= 1;  iQ = iQ+1)
      {
         Q = static_cast<bool>(iQ);
         cout << P << "   " << Q << "   " << !P << "   "
              << (P && Q) << "    " << (P || Q) << "    "
              << (P <= Q) << "    " << (P == Q) << "    "
              << (P != Q) << endl;
      } // for (iQ...
   } // for (iP
}
```

Παράδειγμα 2

Ο πιο συνηθισμένος τύπος για τη μεταβλητή ελέγχου είναι ο `int`. Ας δούμε το εξής πρόβλημα: Να γραφεί πρόγραμμα που θα διαβάζει την τιμή του n και θα υπολογίζει και θα τυπώνει το άθροισμα:

$$1^1 + 2^2 + \ldots + n^n$$

Εδώ βλέπουμε ότι έχουμε (α) μετρούμενη επανάληψη (1.. n) και (β) υπολογισμό αθροίσματος. Σύμφωνα με όσα μάθαμε (Πλ. 6.2α) γράφουμε:

```
   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      υπολόγισε τον m-οστό όρο mm;
      sum = sum + mm;
   }
```

Αυτό το πρόγραμμα θα μοιάζει με αυτό της Μέσης Τιμής, με τη διαφορά ότι ο όρος που θα προσθέτουμε στο *sum* δεν προέρχεται από ανάγνωση από το πληκτρολόγιο αλλά από υπολογισμό που γίνεται μέσα στο πρόγραμμα.

Για τον υπολογισμό του m^m θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση **pow**. Αλλά εδώ θα κάνουμε τον υπολογισμό με πολλούς πολλαπλασιασμούς:

$$m^m = \underbrace{m \cdot \ldots \cdot m}_{m\varphi o\rho\varepsilon\varsigma}$$

Το πώς υπολογίζουμε ένα γινόμενο το είδαμε στο Πλ. 6.2β. Υπολογίζουμε λοιπόν το m^m ως εξής:

```
product = 1;  k = 1;          product = 1;
while ( k <= m )          ή  for ( k=1; k <= m; k=k+1 )
{
   product = product*m;          product = product*m;
   k = k + 1;
} // while (k ...
```

Να λοιπόν το πρόγραμμά μας:

```
#include <iostream>
using namespace std;
int main()
{
   int  n, sum;
   int  m, k, product;

   cin >> n;
   sum = 0;
   for ( m = 1;  m <= n;  m = m + 1 )
   {
      product = 1;
      for ( k = 1;  k <= m;  k = k + 1 )
      {
         product = product*m;
      } // for (k...
      sum = sum + product;
   } // for (m...
   cout << " n = " << n << "  Άθροισμα Σειράς = "
        << sum << endl;
} // main
```

Παράδειγμα 3

Ας έρθουμε τώρα στο πρόγραμμα της §6.4: Εδώ η μεταβλητή ελέγχου είναι τύπου **double**. Μπορούμε να γράψουμε την επανάληψη με μια **for** ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
   double a;    // επιτάχυνση
   double x;    // διάστημα
   double t;    // χρόνος
   double step;

   cout.setf( ios::fixed,ios::floatfield );  cout.precision( 8 );
   cout << " t          x" << endl;
   cout << "   sec        m" << endl;
   a = 10.0;   // m/sec²
   step = 0.5; // sec
   for ( t = 0.0;  t <= 15.0;  t += step )
   {
```

```
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
   } // for
}
```

Πάντως, όπως θα μάθεις αργότερα, είναι καλύτερο να σκεφτείς ότι οι επαναλαμβανόμενες εντολές εκτελούνται 31 φορές και να γράψεις:

```
   int    k;
     . . .
   t = 0.0;
   for ( k = 1;  t <= 31;  k++ )
   {
      x = 0.5*a*t*t;
      cout.width( 6 ); cout.precision( 1 ); cout << t << "     ";
      cout.width( 7 ); cout.precision( 2 ); cout << x << endl;
      t = t + step;
   } // for
```

Η **for** της C++ έχει πολύ περισσότερες δυνατότητες. Προς το παρόν είδαμε –και θα χρησιμοποιούμε– μόνον αυτές που μας χρειάζονται.

6.5 Λαθάκια και Σοβαρά Λάθη

Τα λάθη στις επαναληπτικές εντολές μπορεί να είναι πολύ πιο «δραματικά» και αυτό έχει να κάνει με τον τερματισμό. Το δίδαγμα από εδώ είναι:

- ***Σε κάθε εντολή while πρέπει να φροντίζεις ώστε κάποτε, κατά τη διάρκεια της εκτέλεσης, η συνθήκη της εντολής να γίνεται*** `false`**.**

Τα ίδια ισχύουν και για τη **for**, αλλά οι περιορισμοί που έχουμε βάλει στη χρήση της είναι πολύ ισχυρότεροι.

Ένας πολύ απλός τρόπος να παραβείς το παραπάνω δίδαγμα είναι να βάλεις ένα `";"` μετά την παρένθεση της συνθήκης της **while**:

```
while ( S );
```

Στην περίπτωση αυτή ζητάς την εκτέλεση της *κενής εντολής* όσο ισχύει η συνθήκη της **while**. Αν λοιπόν η συνθήκη ισχύει αρχικώς και αρχίσει η εκτέλεση της **while**, η κενή εντολή δεν μπορεί να την ανατρέψει. Αν δεν το πιστεύεις δοκίμασέ το. Αλλά να θυμάσαι:

- ***Δεν βάζουμε ποτέ "*** `;` ***" μετά την παρένθεση της συνθήκης της while.***

Όπως βλέπεις, ένα «τόσο δα λαθάκι» μπορεί να έχει σοβαρά επακόλουθα.

6.6 Τι (Πρέπει να) Έμαθες

Θα πρέπει να μπορείς να γράψεις προγράμματα στα οποία υπάρχει ανάγκη να εκτελεσθούν πολλές φορές οι ίδιες εντολές

- είτε για να κάνουμε την ίδια επεξεργασία σε πολλά στοιχεία εισόδου
- είτε για να υπολογίσουμε μια τιμή με επαναληπτικό αλγόριθμο.

Θα πρέπει να μπορείς να γράψεις τέτοια προγράμματα είτε ξέρεις το πλήθος των επαναλήψεων πριν αρχίσει η εκτέλεσή τους (μετρούμενη επανάληψη) είτε θα τις συνεχίσεις μέχρι να ισχύσει κάποια συνθήκη (π.χ. να διαβαστεί η τιμή-φρουρός).

Θα πρέπει να έμαθες ακόμη να κάνεις μερικές πολύ συνηθισμένες επαναληπτικές δουλειές:

- υπολογισμό αθροίσματος ή γινομένου,

- υπολογισμό μεγίστου ή ελαχίστου

για τιμές που δημιουργεί το πρόγραμμα ή για τιμές που διαβάζει,

Τέλος, θα πρέπει να μπορείς να αποδείξεις την ορθότητα (απλών τουλάχιστον) προγραμμάτων με επαναλήψεις.

Ασκήσεις

Α Ομάδα

6-1 Γράψε το πρόγραμμα Μέση Τιμή 2 που περιγράψαμε στην §6.1. Θα διαφέρει από το Μέση Τιμή 1 στο ότι το *n* είναι μεταβλητή που η τιμή της διαβάζεται πριν από τις τιμές που θα αθροίσουμε.

6-2 Με βάση το πρόγραμμα για το άθροισμα, γράψε πρόγραμμα που θα υπολογίζει και θα εκτυπώνει το γινόμενο *n* πραγματικών αριθμών που δίνονται από το πληκτρολόγιο. Πριν από τις τιμές θα δίνεται η τιμή του *n*. Απόδειξε ότι το πρόγραμμά σου είναι ολικώς σωστό.

6-3 Ξαναδιάβασε το πρόγραμμα για το λογαριασμό της ΔΕΗ στην §6.4. Τροποποίησέ το έτσι ώστε να βγάζει πολλούς λογαριασμούς, διαβάζοντας από το πληκτρολόγιο τις καταναλώσεις. Το πρόγραμμα θα σταματάει αν του δώσουμε *αρνητική* κατανάλωση (κάθε αρνητική τιμή είναι τιμή - φρουρός).

Β Ομάδα

6-4 Τροποποίησε το πρόγραμμα Μέση Τιμή 3 της ώστε να έχει μια μόνον φορά την `cin >> x`. Σου αρέσει όπως έγινε;

6-5 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει την τιμή του *n*-οστού όρου της ακολουθίας, που καθορίζεται από τον τύπο:

α) $a_0 = 0,\ a_{k+1} = a_k k + k,$ για $k \geq 1$

β) $b_0 = b_1 = 0,\ b_k = b_{k-1} + b_{k-2} + k,$ για $k \geq 2$

6-6 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο πραγματικούς αριθμούς – θετικούς ή αρνητικούς– και θα σταματάει μόλις διαβάσει 10 αρνητικούς. Στο τέλος θα μας λέει:

- πόσους αριθμούς διάβασε συνολικώς και
- το άθροισμα των θετικών αριθμών που διάβασε.

6-7 Μετά την ανακοίνωση των αποτελεσμάτων στις εξετάσεις δύο μαθημάτων, έγινε φανερό ότι στο δεύτερο από αυτά έγινε «σφαγή». Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 40 δυάδες πραγματικών αριθμών που αντιπροσωπεύουν τους βαθμούς 40 σπουδαστών στα δύο μαθήματα. Στο τέλος θα πρέπει να μας λέει:

- πόσοι σπουδαστές είχαν στο δεύτερο μάθημα μεγαλύτερο βαθμό από ότι στο πρώτο,
- τους μέσους όρους των βαθμών στα δύο μαθήματα.

6-8 Απόδειξε ότι:

```
// n > 0
   k = 1;  sum = 0;
   while ( k <= n )    // I:  (sum = (k - 1)*k/2)
   {  sum = sum + k;  k = k + 1;  }
// sum = (n + 1)*n/2
```

Γ Ομάδα

6-9 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο 150 πραγματικούς αριθμούς. Το πρόγραμμα θα πρέπει να υπολογίσει και στο τέλος να γράψει:

- το άθροισμα των θετικών,
- πόσοι είχαν απόλυτη τιμή μεγαλύτερη από 10,
- ποιος ήταν ο μέγιστος *αρνητικός* από τους αριθμούς που δόθηκαν (δηλαδή, αρνητικός με τη ελάχιστη απόλυτη τιμή).

6-10 Ξαναδιάβασε το πρόγραμμα της ελεύθερης πτώσης, της §3.4 και κάνε τις εξής παραλλαγές:

1. Το πρόγραμμα να τυπώνει ανά 1 sec, από τότε που αφέθηκε και μέχρι να κτυπήσει στο έδαφος
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - το ύψος που βρίσκεται το σώμα,
   - την ταχύτητα που έχει το σώμα.
2. Το πρόγραμμα να τυπώνει ανά 10 *m* διανυθέντος διαστήματος, από τότε που αφέθηκε (διάστημα 0) και μέχρι να κτυπήσει στο έδαφος (διάστημα *h*)
   - το διάστημα που διανύθηκε από την αρχή της πτώσης,
   - το ύψος που βρίσκεται το σώμα,
   - τον χρόνο από την αρχή της πτώσης (χρόνος 0),
   - την ταχύτητα που έχει το σώμα.
3. Το πρόγραμμα δεν θα διαβάζει το αρχικό ύψος αλλά θα υπολογίζει και θα τυπώνει τα *tP*, *vP* για *h* = 100 *m*, 200 *m*, ..., 1000 *m*.

6-11 *Μέθοδος των ελαχίστων τετραγώνων.* Αν μας δοθούν *n* σημεία του επιπέδου *xy*, (x_k, y_k), *k* = 1..*n*, η «καλύτερη ευθεία», $y = \alpha x + \beta$, που περνάει από αυτά, είναι αυτή που έχει:

$$\alpha = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - \left(\sum x\right)^2} \quad \text{και} \quad \beta = \frac{\sum y \sum x^2 - \sum x \sum xy}{n\sum x^2 - \left(\sum x\right)^2}$$

όπου:

$$\sum x = \sum_{k=1}^{n} x_k \,,\ \sum y = \sum_{k=1}^{n} y_k \,,\ \sum x = \sum_{k=1}^{n} n_k^2 \,,\ \sum xy = \sum_{k=1}^{n} x_k y_k$$

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο την τιμή του *n* και στη συνέχεια τα (x_k, y_k), *k* = 1..*n* και θα υπολογίζει τα α και β.

6-12 Απόδειξε ότι αν:

```
   int  x, z, u;
```

τότε:

```
// x ≥ 0
   z = 0;  u = 1;
   while ( u <= x )     // I: (z² ≤ x) && (u = (z+1)²)
   {
      z = z + 1;
      u = u + z + z + 1;
   }
// z² ≤ x < (z+1)²
```

Απόδειξε πρώτα ότι η $(z^2 \le x)$ **&&** $(u = (z+1)^2)$ είναι αναλλοίωτη της **while**.

Όπως καταλαβαίνεις, το *z* είναι ακέραια προσέγγιση στην $\sqrt{x}$. Συμπλήρωσε το παραπάνω πρόγραμμα έτσι ώστε να διαβάζει από το πληκτρολόγιο την τιμή της *x* και εφόσον είναι σύμφωνη με την προϋπόθεση να υπολογίζει και να τυπώνει την «ακέραια τετραγωνική ρίζα» της *x* με τον παραπάνω τρόπο και με την `static_cast <int>(sqrt(x))`.

6-13 Απόδειξε ότι αν:

```
   int  x, c;
```

τότε:

```
// x ≥ 0
   c = x;
   while ( c*c*c > x )   // a: x < (c + 1)³
      c = c - 1;
// c³ ≤ x < (c + 1)³
```

6-14 Έστω ότι έχουμε την ακολουθία: $a_k = k!/p^k \mid k = 1,2,...$ όπου p φυσικός > 1. Όποια και να είναι η τιμή της p, η ακολουθία είναι τελικώς αύξουσα, αλλά στη αρχή –στους πρώτους όρους– βλέπεις οτι οι τιμές είναι φθίνουσες. Γράψε πρόγραμμα που

- θα διαβάζει την τιμή του p από το πληκτρολόγιο,
- θα υπολογίζει και θα τυπώνει τους $2p$ πρώτους όρους της ακολουθίας,
- θα μας δίνει την τιμή και την τάξη του ελάχιστου από αυτούς.

6-15 Θέλουμε πρόγραμμα που θα παρακολουθεί έναν παίκτη του μπάσκετ κατά τη διάρκεια ενός παιγνιδιού. Το πρόγραμμα θα παίρνει τα εξής στοιχεία:
`'1'` κάθε φορά που ο παίκτης επιτυγχάνει καλάθι με ελεύθερη βολή,
`'2'` κάθε φορά που ο παίκτης επιτυγχάνει δίποντο,
`'3'` κάθε φορά που ο παίκτης επιτυγχάνει τρίποντο και
`'4'` κάθε φορά που ο παίκτης κάνει φάουλ.
Πληκτρολογώντας `'0'` δείχνουμε στο πρόγραμμα ότι τελείωσε το παιχνίδι. Όταν ο παίκτης συμπληρώσει πέντε φάουλ, θα πρέπει να βγαίνει το μήνυμα: `"ΒΓΑΙΝΕΙ ΕΞΩ ΜΕ 5 ΦΑΟΥΛ"`.

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

6-16 Απόδειξε ότι η:

$$1 \le kmax \le k\text{–}1 \;\&\&\; tmax == t_{kmax} \;\&\&\; \forall j: 1..\, k\text{–}1 \bullet t_j \le tmax$$

είναι αναλλοίωτη της **while** στο:

```
      cin >> t;     // t == t1
      k = 1;  tmax = t;  kmax = k;
      k = k + 1;    // == 2
      while ( k <= n )
      {
         cin >> t;  // t == tk
         if ( t > tmax )  {  tmax = t;  kmax = k;  }
         k = k + 1;
      } // while
```

6-17 α) Γενίκευσε το πρόγραμμα ανάγνωσης ακεραίου της §4.5 ώστε να διαβάζει οποιαδήποτε τιμή τύπου `unsigned long`.

β) Βελτίωσε το πρόγραμμά σου ώστε να διαβάζει και προσημασμένες τιμές.

6-18 Βελτίωσε το πρόγραμμα που έγραψες στην άσκ. 6.17 ώστε να διαβάζει πραγματικούς της μορφής πρόσημο, ακέραιο μέρος, κλασματικό μέρος.

6-19 *Ο Χρόνος στη C++.* Η C++ μας δίνει τη συνάρτηση *time* που, με την κλήση `time(0)`, μας επιστρέφει τα δευτερόλεπτα που πέρασαν από τη στιγμή 00:00:00 *GMT, της* 1ης Ιανουαρίου 1970. Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <ctime>`".

Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο ακέραιους αριθμούς και θα σταματάει όταν διαβάσει την τιμή 0. Αρχίζοντας από τον δεύτερο αριθμό, μετά την ανάγνωση θα μας δίνει τον χρόνο που πέρασε από την προηγούμενη πληκτρολόγηση. Στο τέλος θα μας δίνει:

- το πλήθος των πληκτρολογήσεων,

- το μέγιστο χρονικό διάστημα μεταξύ δύο πληκτρολήσεων,
- τον μέσο χρόνο μεταξύ δύο πληκτρολογήσεων.