ΗΥ-150
Προγραμματισμός

Συναρτήσεις

Αλγόριθμοι / Προγράμματα

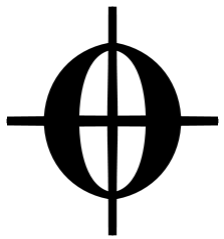
- Λύνουμε ένα πρόβλημα εκτελώντας μια ακολουθία από εντολές σε συγκεκριμένη σειρά
- Αλγόριθμος: καθορίζει τις εντολές και τη σειρά εκτέλεσης
- Έλεγχος προγράμματος (program control): καθορίζει αυστηρά τη σειρά εκτέλεσης των εντολών

Συναρτήσεις - Functions

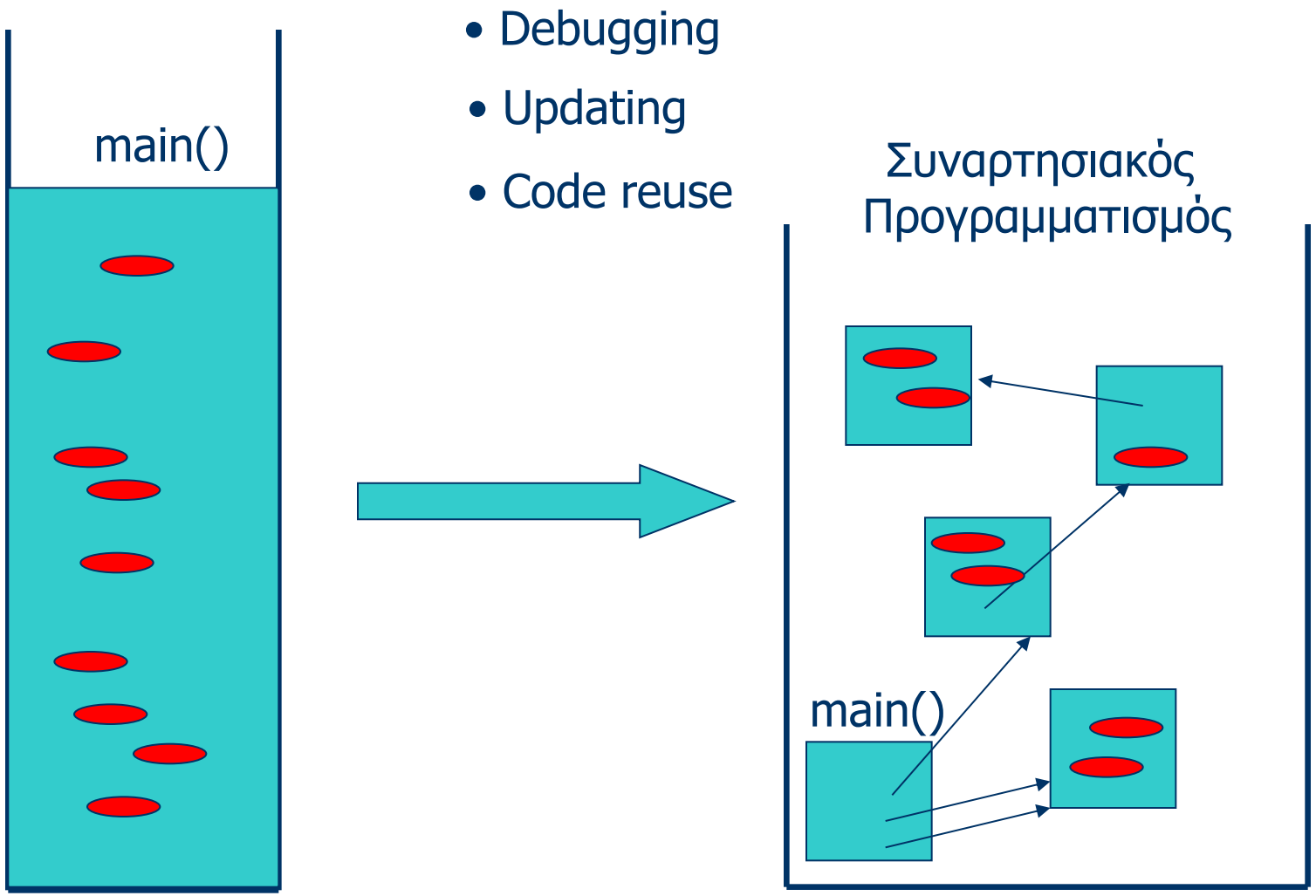
- Χωρίζουμε το πρόγραμμα σε μέρη
- Φιλοσοφία του «διαίρει και βασίλευε» (divide and conquer)
- Τα μέρη αντιστοιχούν σε επιμέρους και μικρότερα προβλήματα
- Αναλογία με μια επιχείρηση:
  - Ο προϊστάμενος αναθέτει μια δουλειά σε κάποιον υφιστάμενο
  - Παίρνει το αποτέλεσμα χωρίς να «ενδιαφέρεται» για τις λεπτομέρειες
  - Παρενέργειες;

Συναρτήσεις - Functions

- Τα υποπροβλήματα πιο εύκολα διαχειρίσιμα
- Κάθε υποπρόβλημα αντιστοιχεί σε μία **συνάρτηση**
- Πιο εύκολη **διαχείριση κώδικα**
  - Debugging
  - Updating
- Χρήση υπαρχόντων συναρτήσεων σε καινούρια προγράμματα για τη λύση νέων προβλημάτων: **code reuse**
  - Επαναχρησιμοποίηση κώδικα
- **Αποφυγή επανάληψης του ίδιου κώδικα**
- Αφαίρεση / **abstraction**: «κρύβονται» λεπτομέρειες που δεν είναι απαραίτητες συνεχώς στον προγραμματιστή



Συναρτήσεις - Functions



Δομημένος προγραμματισμός

- Οργάνωση του προγράμματός σε συναρτήσεις == απλοποίηση του κώδικα:

επικεντρωνόμαστε σε απλούστερες διαδικασίες:

  - αυτόνομη υλοποιηση, διορθωση και βελτιστοποιηση τμημάτων του κώδικα, ανεξάρτητα από το υπόλοιπο πρόγραμμα
  - να χρησιμοποιείται από εμάς ή άλλους σε διαφορετικά προγράμματα.

- Όταν υπάρξει «απομόνωση» του κώδικα σε αυτόνομη συνάρτηση, η χρήση του απλοποιείται σημαντικά καθώς μας απασχολεί μόνο το πώς τον καλούμε και τι ορίσματα πρέπει να "περάσουμε" στη συνάρτηση και όχι το ποιούς ακριβώς υπολογισμούς εκτελεί («μαύρο κουτί»).
- Η οργάνωση του κώδικα σε δεδομένα και σε διαδικασίες που επιδρούν σε αυτά περιγράφεται ως δομημένος (structured) ή διαδικαστικός (procedural) προγραμματισμός.

Ορισμός Συναρτήσεων

- *Όνομα-συνάρτησης:* οποιοδήποτε δεκτό όνομα
- *Επιστρεφόμενος-τύπος:* τύπος των δεδομένων που επιστρέφει η συνάρτηση
  - **void**: δεν επιστρέφει τίποτα
- *Λίστα-παραμέτρων:* δηλώσεις παραμέτρων, χωρίζονται με κόμμα (τύπος παράμετρος, τύπος παράμετρος, ...)

Επιστρεφόμενος-τύπος όνομα-συνάρτησης (λίστα-παραμέτρων)

```
{
    Δηλώσεις μεταβλητών;
    Εντολές;
}
```

```
int SquareIt(int a)
{
        int result;
        result = a*a;
        return result;
}
```

Κλήση Συναρτήσεων – Function Call

όνομα_συνάρτησης (όρισμα1, ..., όρισμαΝ)

έκφραση1 ← όνομα_συνάρτησης (έκφραση1 ..., έκφρασηΝ)

τιμή1 ← η τιμή που επιστρέφει ο κώδικας της συνάρτησης όταν εκτελεστεί με τα δοσμένα ορίσματα

τύπος1 ← ο τύπος που επιστρέφει η συνάρτηση

```
x = (SquareIt(5) > 7);
```

Η έκφραση `SquareIt(5)` αντικαθίσταται από την έκφραση `25.`

squareIt

```
int SquareIt(int x)
{
    int result;
    result = x*x;
    return result;
}
int
main()
{
...
}
```

max

```
/*Επιστρέφει το μέγιστο μεταξύ των x, y*/
int max(int x,int y)
{
   int z = x;   /*δηλώσεις μεταβλητών*/

   if (y > x)   /*λίστα εντολών*/
      z = y;
   return z;
}
```

Συναρτήσεις void

- Ο επιστρεφόμενος τύπος μπορεί να είναι **void**. Δηλώνεται έτσι ότι δεν επιστρέφεται τιμή.
- Η λίστα ορισμάτων μπορεί να είναι κενή ή, ισοδύναμα, να περιέχει τη λέξη **void**.

```
void f(int x)
{
  cout<<"Test"<<x<<"\n";
  return;
}
```

Λάθος:

```
  x = f() + 10; y = f(5) + 10;
```

Κλήση

- Το όνομά της, μέσα σε ζεύγος παρεθέσεων, από ποσότητες κατάλληλου τύπου ώστε να αντιστοιχούν στα ορίσματά της, ή να μπορούν να μετατραπούν σε αυτά.
- (Για τώρα) Οι ποσότητες αυτές πρεπει να είναι ακριβώς τόσες όσα και τα ορίσματα.

```
double func(double a, double b);
int read(double & a, char const fname[]);
void  print(char c);

int
main() {
     int i;
     double x,y,z,r,t;

     x = 3.2;
     y = 3.4;

     // Calls func with double, double.
     z = func(x,y);

     i = 3;

     // Calls func with double, int.
     // int is promoted to double.
     t = func(x,i);

     // calls a void function.
     cout << 'a';

     read(r, "input.dat");
     // ignores returned value.
}
```

Call by value

- Αντίγραφο της τιμής του ορίσματος περνάει στη συνάρτηση
- Αλλαγές στο όρισμα (αντίγραφο) δεν επηρεάζουν το πρωτότυπο
- Χρήση: όταν η συνάρτηση ΔΕΝ χρειάζεται να αλλάξει το όρισμα
  - Αποφεύγουμε αλλαγές κατά λάθος

```
/*Επιστρέφει το μέγιστο μεταξύ των x, y*/
int max(int x,int y)
{
     int z = x;   /*δηλώσεις μεταβλητών*/

    if (y > x)   /*λίστα εντολών*/
       z = y;
    return z;
}
```

```
int foo(int x)
{
    X++;
}

int main()
{
   int a = 5;
   cout << a;
   foo(a);
   cout << a;
```

Call by value

- Χρήσιμο στο δομημένο προγραμματισμό: καλούμε μια συνάρτηση χωρίς τον κίνδυνο να μας αλλάξει τις τιμές των μεταβλητών μας
  - Ειδάλλως θα πρέπει να το λαμβάνουμε κάθε φορά υπ’ όψιν

ΠΡΟΣΕΧΩΣ

Call by reference

- Θέλουμε να αλλάξουμε τιμή στα ορίσματα της συνάρτησης
- Περνάμε τη διεύθυνση της μεταβλητής
- Δουλεύουμε πάνω στα «ίδια» τα δεδομένα και όχι στα αντίγραφα τους
- Πρέπει να λαμβάνουμε υπ’ όψιν την επίδραση των σχετικών κλήσεων

Παράδειγμα - Call by reference

```
void swap( int *n1, int *n2)
{
    int temp;

    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

```
void swap2( int n1, int n2)
{
        int temp;

        temp = n1;
        n1 = n2;
        n2 = temp;
}
```

```
int A=1;
int B=2;
swap(&A,&B);
cout << A<< " " << B;

2 1
```

```
int A=1;
int B=2;
swap2(A,B);
cout << A<< " " << B;

1 2
```

Επιστροφή

- Η επιστροφή τιμής από τη συνάρτηση γίνεται με την εντολή **return** *value*;
- Η εντολή μπορεί να εμφανίζεται μία ή περισσότερες φορές στο σώμα της συνάρτησης.
- Μια συνάρτηση που δεν επιστρέφει τιμή (δηλαδή επιστρέφει **void**) μπορεί να περιλαμβάνει εντολές **return**; (χωρίς τιμή).
  - Επίσης, μια τέτοια συνάρτηση μπορεί να "επιστρέφει" την "τιμή" μιας συνάρτησης που "επιστρέφει" **void**.
- Αποτελεί καλή πρακτική το να υπάρχει μόνο ένα σημείο (επιτυχούς) εξόδου από τη συνάρτηση. Για το σκοπό αυτό, χρησιμοποιούμε μια μεταβλητή που αποθηκεύουμε το αποτέλεσμα της συνάρτησης, όπου αυτό παραχθεί. Κατόπιν, την "επιστρέφουμε" στο τέλος του σώματος της συνάρτησης.

Ροή του Προγράμματος

- Η ροή του προγράμματος αλλάζει όταν συναντήσει την κλήση μιας συνάρτησης
- `f(a, b, c)`

1. Πρώτα υπολογίζονται οι εκφράσεις που αντιστοιχούν στα ορίσματα
2. Οι τιμές των εκφράσεων a, b, c αντιγράφονται στα ορίσματα της f με την ίδια σειρά
3. Ο έλεγχος μεταπηδά στην πρώτη εντολή της f
4. Οι εντολές της f εκτελούνται μέχρι να συναντηθεί
   - return έκφραση;
   - Τέλος συνάρτησης (δηλαδή })
5. Η κλήση `f(a, b, c)` αντικαθίσταται με την επιστρεφόμενη τιμή

Συναρτήσεις βιβλιοθήκης

- Συναρτήσεις ορισμένες από το χρήστη (user defined functions)
- Συναρτήσεις βιβλιοθήκης (library functions)
  - η τυπική βιβλιοθήκη της C
  - βιβλιοθήκες συναρτήσεων στο διαδίκτυο

Η βιβλιοθήκη <math.h>

- **double sqrt(double x);** : Υπολογίζει την τετραγωνική ρίζα του x, πρέπει x > 0
- **double exp(double x);** : Υπολογίζει τον εκθετικό του x, e^x
- **double log(double x);** : Υπολογίζει τον φυσικό λογάριθμο του x, ln(x)
- **int abs(int val);** : Υπολογίζει την απόλυτη τιμή του val
- **double pow(double x, double y);** : Υπολογίζει την τιμή του x υψωμένη στη δύναμη y, $χ^y$
- **double sin(double x);** : Υπολογίζει το sine του x, σε rad
- **double cos(double x);** : Υπολογίζει το cosine του x, σε rad
- **double tan(double x);** : Υπολογίζει την εφαπτομένη του x, σε rad
- **double fmod(double x, double y);** : Υπολογίζει το δεκαδικό υπόλοιπο της διαίρεσης χ/y, y > 0.
- **double ceil(double x);** : Υπολογίζει την μικρότερη ακέραια τιμή όχι μικρότερη από x.
- **double floor(double x);** : Υπολογίζει την μεγαλύτερη ακέραια τιμή όχι μεγαλύτερη από χ.

Για να κάνουμε compile πρόγραμμα που χρησιμοποιεί την math.h: >**gcc –lm program.c**

```
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
```

Η συνάρτηση rand επιστρέφει ψευδοτυχαίους αριθμούς από 0 έως RAND_MAX
Η συνάρτηση srand χρησιμοποιείται για την αρχικοποίηση της rand() και τη γέννηση ψευδοτυχαίων αριθμών.
Καλούμε μία φορά την srand και μετά την rand διαδοχικά για την παραγωγή των τυχαίων αριθμών.
Αν ξανακαλέσουμε την srand και μετά την rand θα προκαλέσουμε την ίδια σειρά τυχαίων αριθμών.

Ορισμός Συναρτήσεων

- Δηλώσεις μεταβλητών μέσα σε blocks: ΤΟΠΙΚΕΣ μεταβλητές
- Συναρτήσεις ΔΕΝ μπορούν να οριστούν μέσα σε άλλες συναρτήσεις
- ΟΠΟΙΑΔΗΠΟΤΕ συνάρτηση μπορεί να καλέσει άλλες συναρτήσεις και τον εαυτό της (αναδρομή)

Πρωτότυπα Συναρτήσεων (Function Prototypes)

Επιστρεφόμενος-τύπος όνομα-συνάρτησης (λίστα τύπων-παραμέτρων);

(Προ-) Δηλώσεις των συναρτήσεων στην αρχή του αρχείου

- Χρειάζεται αν χρησιμοποιούμε τη συνάρτηση πριν τον ορισμό της
- Πλεονεκτήματα:
  - Ελέγχουμε την ορθή χρήση των συναρτήσεων (πριν ακόμη τις υλοποιήσουμε)
  - Μπορούμε να δούμε ποιες είναι οι συναρτήσεις που έχουν υλοποιηθεί στο πρόγραμμα
  - Μπορούμε να καλέσουμε όλες τις συναρτήσεις που έχουν υλοποιηθεί στο πρόγραμμα χωρίς να μας ενδιαφέρει η σειρά που έχουν γραφτεί

```
1  /* Fig. 5.4: fig05 04.c
2     Finding the maximum of three integers */
3  #include <iostream>
4
5  int maximum( int, int, int );   /* function prototype */
6  using namespace std;
7  int main()
8  {
9     int a, b, c;
10
11    cout << "Enter three integers: " ;
12    cin >> a; cin >> b; cin >> c ;
13    cout << "Maximum is: << maximum( a, b, c ) ;
14
15    return 0;
16 }
17
18 /* Function maximum definition */
19 int maximum( int x, int y, int z )
20 {
21    int max = x;
22
23    if ( y > max )
24       max = y;
25
26    if ( z > max )
27       max = z;
28
29    return max;
30 }
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

3. Function definition

Program Output

Συναρτήσεις

- Επικοινωνία με το υπόλοιπο πρόγραμμα
  - Δέχονται ορίσματα
  - Επιστρέφουν μια τιμή
  - Απομονώνουν την επίλυση του υποπροβλήματος
- Ιδανικά, δεν θέλουμε καμία άλλη επίδραση στο υπόλοιπο πρόγραμμα πέραν των παραπάνω
- Συναρτησιακός προγραμματισμός (functional programming)

- Ενδέχεται να επηρεάσουν κι αλλιώς το πρόγραμμά μας αν χρησιμοποιούμε καθολικές μεταβλητές, δείκτες, κτλ
- Παρενέργειες

Συναρτήσεις και Μεταβλητές

- Ιδιότητες μεταβλητών
  - Εμβέλεια (scope)
    - Ποια μέρη του προγράμματος αναγνωρίζουν την ίδια μεταβλητή (ίδια θέση μνήμης) με το ίδιο όνομα
  - Διάρκεια ύπαρξης στη μνήμη (storage duration)
    - Πότε μια συγκεκριμένη διεύθυνση μνήμης λαμβάνει ένα συγκεκριμένο όνομα και πότε αποδεσμεύεται αυτή η αντιστοίχιση
- Προς το παρόν θα αγνοήσουμε την περίπτωση να έχουμε χωρίσει το πρόγραμμα σε διαφορετικά αρχεία

Εμβέλεια

- Μια μεταβλητή μπορεί να έχει εμβέλεια:
  - Καθολική: σε όλο το πρόγραμμα ενός αρχείου
  - Τοπική: μέσα στον ορισμό μιας συνάρτησης
  - Τοπική: μέσα σε ένα block

    {

    …

    }

```
int foobar;

int
fun()
{
   int i, foo;
   {
     int bar;
   }
   for (i=0;i<N;i++)
   {
     int bar, foo;
   }
}
```

Καθολικές (global) μεταβλητές

- Δηλώνονται (συνήθως πριν και) έξω από κάθε συνάρτηση

```
int global1, global2;
int main(void)
{
  global1 = 5;
}
```

- Εμβέλεια: προσπελάσιμες από κάθε άλλη συνάρτηση μετά την δήλωσή τους
- Διάρκεια ύπαρξης: από την αρχή της εκτέλεσης μέχρι το τέλος όλου του προγράμματος

Καθολικές (global) μεταβλητές

- Χρήσιμες όταν πολλές συναρτήσεις μοιράζονται κοινά δεδομένα
  - δεν είναι πρακτικό να τα περνάμε ως ορίσματα κάθε φορά
- Η κλήση μιας συνάρτησης μπορεί να αλλάζει την κατάσταση και για τις υπόλοιπες.
  - Δύσκολο καμιά φορά να βεβαιώσουμε ότι οι αλλαγές θα γίνουν με σωστή σειρά
  - Πρέπει να λαμβάνουμε υπ' όψιν πολλές περιπτώσεις ταυτόχρονα
- Δύσκολη η επαναχρησιμοποίηση – τροποποίηση του προγράμματος
- *ΑΠΟΦΥΓΕΤΕ τις καθολικές μεταβλητές αν δεν είναι απαραίτητες!*

Τοπικές (local) Μεταβλητές

- **Δηλώση:** μέσα σε μία συνάρτηση/block

```
int main(void)
{
  int local1, local2;
  local1 = 5;
  {
       int local3;
       local3 = 7;
  }
}
```

- **Εμβέλεια:** προσπελάσιμες μόνο μέσα στη ίδια συνάρτηση/block
- **Διάρκεια:** από την αρχή μέχρι το τέλος της εκτέλεσης της συνάρτησης/block

Τοπικές (local) μεταβλητές

- Χρήσιμες για τις περισσότερες περιπτώσεις
  - Συνήθως θέλουμε μεταβλητές για αποθήκευση ενδιάμεσων αποτελεσμάτων. Οι «τελικοί» υπολογισμοί καταλήγουν συνήθως σε μία τιμή, αυτήν που επιστρέφει η συνάρτηση
- Προσπαθήστε να χρησιμοποιείτε **μόνο** τοπικές μεταβλητές και να σκέφτεστε λύσεις που δεν χρησιμοποιούν καθολικές μεταβλητές
  - Χρήσιμο στη δόμηση του προγράμματος με απλές συναρτήσεις

Μεταβλητές

- Καθολικές (global) έναντι τοπικών μεταβλητών (local)

```
double sum;         //καθολικές

int nextOfX(int x)
{
    int y = x;      //τοπική
    y = y + 1;
    sum = sum+1;
    return y;
}

main()
 {
    int x = 0;      //τοπική
    sum = 0;
    x = nextOfX(0);
    x = nextOfX(x);
}
```

«Υπάρχουν» όσο «τρέχει η συνάρτηση»

«Υπάρχουν» όσο «τρέχει το πρόγραμμα»

Στατικές (static) Μεταβλητές

- Δηλώνονται με το πρόθεμα static

```
static int x=8;
```

- Διάρκεια ύπαρξης: από την αρχή της εκτέλεσης μέχρι το τέλος όλου του προγράμματος **και για τις τοπικές**
  - Διατηρούν την τιμή τους μεταξύ των διαφόρων κλήσεων της συνάρτησης
- Εμβέλεια τοπικών στατικών: στην ίδια συνάρτηση
- Εμβέλεια καθολικών στατικών: **μόνο στο ίδιο αρχείο**

Στατικές (static) Μεταβλητές

- Γιατί στατικές μεταβλητές
  - **Τοπικές:** μια μόνο συνάρτηση χρησιμοποιεί την μεταβλητή που πρέπει να την θυμάται μεταξύ διαφορετικών κλήσεων
  - **Καθολικές:** ένα μικρό σύνολο από συναρτήσεις χρησιμοποιεί την μεταβλητή, οπότε η μεταβλητή δηλώνεται σαν στατική και οι όλες αυτές οι συναρτήσεις μπαίνουν στο ίδιο αρχείο
- Αρχικοποίηση στην αρχή της εκτέλεσης του **προγράμματος**
  - `static int x=8;`

Static variables

```
void a()
{
    static int staticVariable = 0; /* mono thn prwth fora*/
    staticVariable++;
    cout << "Number of calls :" << staticVariable << endl;
}
```

Output

1

2

3

….

Οργάνωση κώδικα

- Σε μεγαλύτερα προγράμματα, οι δηλώσεις των συναρτήσεων συγκεντρώνονται σε ένα ή περισσότερα header files (κατάληξη **.h**), τα οποία συμπεριλαμβάνονται κατά την προεπεξεργασία του κώδικα:

```
#include "name.h"
```

- όπου name.h το όνομα του header file και μπορεί να περιλαμβάνει και το path.
- Οι headers που ορίζει ο προγραμματιστής περικλείονται σε “”.
- Οι headers του συστήματος περικλείονται σε <>.
- Με την συμπερίληψη των κατάλληλων headers:
  - ο compiler γνωρίζει τον τρόπο κλήσης των συναρτήσεων ή εκτελέσιμων βιβλιοθηκών που χρειάζεται ένα τμήμα κώδικα.
  - ο προγραμματιστής μπορεί να επισκοπήσει εύκολα τη λειτουργικότητα του κώδικα

Ασκήσεις

- Γράψτε ένα πρόγραμμα που να διαβάζει αριθμούς από την κύρια είσοδο και να τυπώνει στην οθόνη πόσους θετικούς, αρνητικούς και ίσους με το 0 αριθμούς περιέχει.
- Γράψτε κώδικα που να τυπώνει τις ρίζες του πολυωνύμου $ax^2 + bx + c = 0$
- Γράψτε προγράμμα που να τυπώνει τους N πρώτους όρους της ακολουθίας *Fibonacci*

$$F_n = F_{n-1} + F_{n-2},$$

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

// Fibonacci

```
int main()
{
    int n = 16;
    int f0 = 0;
    int f1 = 1;
    if (n > 0)
        cout << f0;
    if (n > 1)
        cout << f0;
    for (int i = 2; i < n; ++i)
    {
        int f2 = f0 + f1;
        cout << f2;
        f0 = f1;
        f1 = f2;
    }
}
```

Ασκήσεις

- Γράψτε προγράμματα που να υπολογίζουν:

$$\cos(x)=\frac{(-1)^k x^{2k}}{(2k)!} \qquad \sin(x)=\frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!},$$

Ασκήσεις

- Γράψτε συνάρτηση που
  - να υπολογίζει το παραγοντικό ενός μικρού ακεραίου.
  - να ελέγχει αν το (ακέραιο) όρισμά της είναι πρώτος. Τι διαλέξατε για τον τύπο της επιστρεφόμενης ποσότητας;
  - να υπολογίζει το μέσο όρο των στοιχείων ενός συνόλου αριθμών.
  - να βρίσκει το μέγιστο στοιχείο μιας σειράς αριθμών και να αναφέρει τον αύξοντα αριθμό του αριθμού αυτού
  - να επιστρέφει ένα τυχαίο ακέραιο σε διάστημα [*a*, *b*] που θα προσδιορίζεται από τα ορίσματά της.

mylib

- Φτιάχνεις τα mylib.cpp, mylib.h
- Στο πρόγραμμα σου (έστω αρχείο myask1.cpp), δηλώνεις:
  - #include "mylib.h" (τα mylib.cpp, mylib.h, έστω πως είναι στο ίδιο directory)
- Μεταφράζεις ως εξής:
  - g++ myask1.cpp mylib.cpp

deg2rad, etc

```
#include <iostream.h>
#include <math.h>
double pi = acos(-1.0);
double degrees = 90;
double angle = degrees * 2 * pi / 360.0;
double x = exp (log (10.0));
```

stdout

```
#include <iostream.h>
void newLine ();
void threeLine ();

void main ()
{
        cout << "1st line";
        threeLine ();
        cout << "2nd line";
        newline();
}
```

```
void newLine ()
{
        cout << endl;
}
void threeLine ()
{
        newLine ();
        newLine ();
        newLine ();
}
```

area

```
double area (double radius)
{
        double pi = acos (-1.0);
        double area = pi * radius * radius;
        return area;
}
```

absoluteValue

```
double absoluteValue (double x)
{
        if (x < 0)
        {
                return -x;
        }
        else
        {
                return x;
        }
}
```

Multiple return values

If you put return statements inside a conditional, then you have to guarantee that every possible path through the program hits a return statement.

```
double absoluteValue (double x)
{
        if (x < 0) {
                return -x;
        } else if (x > 0) {
                return x;
        } // WRONG!!
}
```

Γράφοντας συναρτήσεις 1

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
double distance (double x1, double y1, double x2, double y2)
{
        return 0.0;
}
```

- double dist = distance (1.0, 2.0, 4.0, 6.0);
- cout << dist << endl;

Γράφοντας συναρτήσεις 2

```
double distance (double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        cout << "dx is " << dx << endl;
        cout << "dy is " << dy << endl;
        return 0.0;
}
```

Γράφοντας συναρτήσεις 3

```
double distance (double x1, double y1, double x2, double y2)
{
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx*dx + dy*dy;
        double result = sqrt (dsquared);
        return result;
}
double distance (double x1, double y1, double x2, double y2)
{
        return sqrt ((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
}
```

Παραλλαγές

- Two points: the center of the circle and a point on the perimeter. What is the area of the circle?

```
double fred (double xc, double yc, double xp, double yp)
{
   double radius = distance (xc, yc, xp, yp);
   double result = area (radius);
   return result;
}
double fred (double xc, double yc, double xp, double yp)
{
   return area (distance (xc, yc, xp, yp));
}
```

Overloading and polymorphism

- Having more than one function with the same name, which is called overloading, is legal in C++ as long as each version takes different parameters.

```
#include <iostream>
// volume of a cube
int volume(int s)
{
        return s*s*s;
}
// volume of a cylinder
double volume(double r, int h)
{
        return 3.14*r*r*((double) h);
}
// volume of a cuboid
long volume(long l, int b, int h)
{
        return l*b*h;
}

int main()
{
        std::cout << volume(10);
        std::cout << volume(2.5, 8);
        std::cout << volume(100, 75, 15);
}
```

In the above example, the volume of various components are calculated using the same function call "volume", with arguments differing in their data type or their number.

```
int Add(int nX, int nY)
{
  return nX + nY;
}

int AddI(int nX, int nY)
{
  return nX + nY;
}

double AddD(double dX, double dY)
{
  return dX + dY;
}
```

```
int Add(int nX, int nY); // integer version
double Add(double dX, double dY); // floating point version

int Add(int nX, int nY)
{
   return nX + nY;
}

double Add(double dX, double dY)
{
   return dX + dY;
}


int Add(int nX, int nY, int nZ)
{
   return nX + nY + nZ;
}
```

```
void Print(string szValue);
void Print(char *szValue);
```

```
void print(int i) {
        cout << " Here is int " << i << endl;
}
void print(double f) {
        cout << " Here is float " << f << endl;
}
void print(char c) {
        cout << " Here is char " << c << endl;
}
int main() {
        print(10); print(10.10); print('t');
}
```

Ambiguous Matches

```
void Print(unsigned int nValue);
void Print(float fValue);

Print('a');
Print(0);
Print(3.14159);
```

```
Print(static_cast<unsigned int>(0)); // will call Print(unsigned int)
```