ΜΥΥ502

Προγραμματισμός Συστημάτων

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ ΠΑΝΕΠΙΣΤΗΜΙΟΥ ΙΩΑΝΝΙΝΩΝ

Β. Δημακόπουλος

dimako@cse.uoi.gr

http://www.cse.uoi.gr/~dimako

Εισαγωγή στη C

Πίνακες

Πίνακες

- Διαφορές με java:
  - Όταν τους ορίζεις, πρέπει να δηλώσεις και το μέγεθος
  - Οι αγκύλες πάνε μετά το όνομα της μεταβλητής

- Δηλ. δεν παίζει το:

```
int [] myarray;
myarray = new int[30];
```

- Σωστή (και μοναδική) δήλωση:

```
int myarray[30];
```

- Η αρίθμηση των στοιχείων ξεκινάει από τη θέση μηδέν (**0**)

Πίνακες

- ΔΕΝ μπορεί να αλλάξει το μέγεθος του πίνακα.
- ΔΕΝ θυμάται / ελέγχει η C τα όρια του πίνακα.
- ΔΕΝ μπορεί να γίνει αρχικοποίηση του πίνακα παρά μόνο τη στιγμή της δήλωσής του. Μετά καταχώρηση στοιχείο-στοιχείο. Παράδειγμα:

```
#include <stdio.h>
#define N 10

int main() {
  int myarray[N] = {10,20,30,40,50,60,70,80,90,100};

  myarray[3] = 400;
  myarray[10] = 900;      /* Εκτός ορίων – ΔΕΝ μας προειδοποιεί
  ...                        με κάποιο σφάλμα η C */
}
```

Πίνακες χωρίς μέγεθος

- Επιτρέπεται να μην δοθεί το μέγεθος του πίνακα κατά τη δήλωση *μόνο αν προκύπτει από την αρχικοποίησή του*:

```
int main() {
  int myarray[] = {10,20,30,40};
  ...
}
```

  - Προκύπτει (και δεν μπορεί να αλλάξει) ότι το μέγεθος είναι 4 στοιχεία.

- Το παρακάτω δεν επιτρέπεται!

```
int main() {
  int myarray[];
  ...
}
```

Πίνακες με «μεταβλητό» μέγεθος

- Το παρακάτω, όπου το n είναι μεταβλητή, **απαγορεύεται**:

```
int myarray[n];
```

- Αυτοί ονομάζονται πίνακες μεταβλητού μεγέθους (VLA – Variable Length Arrays)
  - Πρωτοεμφανίστηκαν στην C99
  - Έγινε προαιρετική η υποστήριξή τους στη C11 (ισχύει και στη C17)
  - Η C++ ποτέ δεν αποδέχτηκε τα VLAs
  - Έχουν σοβαρά προβλήματα συμβατότητας (δεν μεταφράζονται όλα τα προγράμματα με VLAs σωστά σε όλα τα συστήματα)
  - Έχουν σοβαρά θέματα ασφάλειας.
  - Κρίσιμα λογισμικά (π.χ. ο πυρήνας του Linux) δεν έχουν VLAs πουθενά

- Απαγορεύονται **δια ροπάλου** στο ΜΥΥ502…
  - Το μέγεθος των πινάκων θα είναι πάντα μία **σταθερά**.

Πίνακες - αρχικοποίηση

- Αν ο πίνακας δεν αρχικοποιηθεί κατά τη δήλωσή του
  - τα στοιχεία του θα έχουν τυχαίες τιμές («σκουπίδια»).
- Αν κατά τη δήλωση του πίνακα αρχικοποιηθούν λιγότερα στοιχεία από αυτά που έχει,
  - τα υπόλοιπα αρχικοποιούνται αυτόματα στο 0 (μηδέν)
- Π.χ. εύκολος τρόπος να αρχικοποιήσω όλα τα στοιχεία ενός πίνακα στο 0:

```
int myarray[100] = { 0 };
```

Συμβολοσειρές (strings): βασικά πίνακες χαρακτήρων

❖ Παράδειγμα:

```
int main() {
  char str1[5];
  char str2[6] = {'b', 'i', 'l', 'l', '\0' };

  str2[1] = 'a';  /* b a l l */
  ...
}
```

❖ Απλά πρέπει να υπάρχει στο τέλος και ο ειδικός χαρακτήρας '\0' (λεπτομέρειες αργότερα).

❖ Πολύ πιο εύκολη αρχικοποίηση, μόνο για συμβολοσειρές:

```
  char str3[6] = "maria";    /* Υπονοείται το \0 στο τέλος */
  char str4[] = "demi";      /* Μέγεθος 5 */
```

Πίνακες 2D

- Για διδιάστατους (τριδιάστατους κλπ), κατά τη δήλωση χρησιμοποιούνται πολλαπλές αγκύλες για να δηλώσουν το μέγεθος κάθε διάστασης.

```
int array1[10][20];          /* 10 γραμμών, 20 στηλών */

int main() {
  int array2[30][20];        /* 30 γραμμών, 20 στηλών */

  array1[0][0] = array2[10][10];
  array1[5] = array2[3];     /* Δεν επιτρέπεται */
  ...
}
```

- Οι πίνακες αυτοί είναι ΠΡΑΓΜΑΤΙΚΑ διδιάστατοι και ΣΤΑΘΕΡΟΙ
  - Όχι σαν τη java όπου βασικά πρόκειται για διάνυσμα όπου κάθε στοιχείο του είναι άλλο διάνυσμα (γραμμή).
  - Όλες οι γραμμές ίδιου (αμετάβλητου) μεγέθους (σε αντίθεση με τη java που κάθε γραμμή μπορούσε να έχει διαφορετικό μέγεθος).

Παράδειγμα: πολλαπλασιασμός πινάκων

```
#define N 10
#define M 10
#define L 10

float A[L][M], B[M][N], C[L][N];  /* 2D, global */

void matmul() {           /* main() should call this */
  int i, j, k;

  for(i=0; i<L; i++) {    /* Values for A elements */
    for(j=0; j<M; j++)
      A[i][j] = j;
  }
  for(i=0; i<M; i++) {    /* Values for B elements */
    for(j=0; j<N; j++)
      B[i][j] = j;
  }
  for(i=0; i<L; i++) {    /* C = AxB */
    for(j=0; j<N; j++) {
      C[i][j] = 0.0;
      for(k=0; k<M; k++)
        C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

Η γλώσσα C

Συναρτήσεις

MYY502

Συναρτήσεις

- Πρόκειται απλά για μια ακολουθία εντολών την οποία χρησιμοποιούμε σε αρκετά σημεία του προγράμματός μας και την οποία την ξεχωρίζουμε ως «υποπρόγραμμα», ώστε να μπορεί να κληθεί και να εκτελεστεί όσες φορές θέλουμε.

- «Μέθοδοι» στη Java.
- Στη C δεν αποτελούν μέρος καμίας κλάσης (δεν υπάρχουν κλάσεις), καλούνται από παντού.

Τα 3 βασικά σημεία των συναρτήσεων: **(α) πρωτότυπο**

- Πρωτότυπο (prototype) – **function declaration**
  - **δήλωση** (declaration) που περιγράφει τι χρειάζεται η συνάρτηση για να λειτουργήσει και τι είδους αποτέλεσμα θα επιστρέψει:

```
<τύπος_επιστροφής> όνομα_συνάρτησης (<τύπος_παραμέτρου> παράμετρος, ...);
```

Π.χ.

```
int power(int base, int exponent);
```

- Είναι απαραίτητο το πρωτότυπο;
  - Απάντηση: όχι πάντα, αλλά αν δεν το γράψετε μπορεί να δημιουργηθούν προβλήματα.
  - Θεωρείστε το ισοδύναμο με τη **δήλωση μίας μεταβλητής** (η οποία πρέπει να οριστεί / δηλωθεί πριν την χρησιμοποιήσετε)

Τα 3 βασικά σημεία των συναρτήσεων: **(β) ορισμός**

- Ορισμός / υλοποίηση της συνάρτησης – **function definition**

```
<πρωτότυπο>                    /* Προσοχή: χωρίς το ‘;’ */
{
    ...                        /* οι εντολές της συνάρτησης */
    return (<τιμή>);
}
```

- Η **`return`**: τερματίζει τη συνάρτηση και επιστρέφει το αποτέλεσμα.
  - Τι σημαίνει επιστρέφει το αποτέλεσμα?
  - Αν έχω στη `main()` : “εκτέλεσε τη συνάρτηση power για τα 2, 4” τότε ό,τι τιμή γίνεται return αποθηκεύεται σε μια προσωρινή θέση μνήμης tmp.
    - res = power(2, 4) → res = tmp όπου στο tmp έχει μπει το 2^4, δηλ. το 16
- Ειδικός τύπος συνάρτησης: **`void`**
  - Αν η συνάρτηση είναι τύπου void τότε ΔΕΝ επιστρέφει τίποτε
  - Έχουμε σκέτο:
    `return;`

Τα 3 βασικά σημεία των συναρτήσεων: **(γ) κλήση**

- Κλήση μίας συνάρτησης – **function call**
  - Μέσα στο πρόγραμμα
    - "εκτέλεσε τη συνάρτηση power για τα 2, 4 και υπολόγισε το αποτέλεσμα"
  - Τρόπος κλήσης:

```
<όνομα_συνάρτησης>(τιμές για πραγματικές παραμέτρους)
```

Παράδειγμα

```
#include <stdio.h>

int main() {
    int x, y, k, n, res;
    int power(int base, int n);     /* πρωτότυπο */

    x = 2; y=3; k=4; n=2;
    res = power(x, k);              /* κλήση */
    printf("%d \n", res);
    res = power(y, n);              /* κλήση */
    printf("%d \n", res);

    return 0;                  /* Η main() πρέπει να επιστρέφει 0 αν όλα ΟΚ */
}

/* Ορισμός (υπολογίζει το base υψωμένο στη δύναμη n, base^n) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p;                       /* τιμή επιστροφής */
}
```

Παράμετροι (parameters) ή τυπικές παράμετροι (formal parameters)

Ορίσματα (arguments) ή πραγματικές παράμετροι

Παράδειγμα – αλλού το πρωτότυπο

```
#include <stdio.h>
int power(int base, int n);        /* πρωτότυπο */

int main() {
    int x, y, k, n, res;

   x = 2; y=3; k=4; n=2;
    res = power(x, k);             /* κλήση */
    printf("%d \n", res);
    res = power(y, n);             /* κλήση */
    printf("%d \n", res);

    return 0;                  /* Η main() πρέπει να επιστρέφει 0 αν όλα ΟΚ */
}

/* Ορισμός (υπολογίζει το base^n) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
       p = p*base;
    return p;                      /* τιμή επιστροφής */
}
```

ΔΙΑΦΟΡΑ ΜΕ ΠΡΙΝ:

Όχι μόνο η main(), αλλά ΟΛΕΣ οι επόμενες συναρτήσεις θα «γνωρίζουν» την power().

Πριν, ΜΟΝΟ στη main() είχε γίνει γνωστή!

Παράδειγμα χωρίς πρωτότυπο. Τι θα γίνει;

```
#include <stdio.h>


int main() {
    int x, y, k, n, res;

   x = 2; y=3; k=4; n=2;
    res = power(x, k);              /* κλήση */
    printf("%d \n", res);
    res = power(y, n);              /* κλήση */
    printf("%d \n", res);

    return 0;                  /* Η main() πρέπει να επιστρέφει 0 αν όλα ΟΚ */
}

/* Ορισμός (υπολογίζει το base^n) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
       p = p*base;
    return p;                       /* τιμή επιστροφής */
}
```

ΑΠΟΤΕΛΕΣΜΑ:

Compiler **warnings**: δεν «γνωρίζει» την power() στα σημεία που γίνονται οι δύο κλήσεις. Πολλές φορές υπάρχουν και καταστροφικά αποτελέσματα.

Παράδειγμα χωρίς πρωτότυπο. Τι θα γίνει;

```
#include <stdio.h>


/* Ορισμός (υπολογίζει το base^n) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p;          /* τιμή επιστροφής */
}

int main() {
    int x, y, k, n, res;

    x = 2; y=3; k=4; n=2;
    res = power(x, k);              /* κλήση */
    printf("%d \n", res);
    res = power(y, n);              /* κλήση */
    printf("%d \n", res);

    return 0;                  /* Η main() πρέπει να επιστρέφει 0 αν όλα ΟΚ */
}
```

ΑΠΟΤΕΛΕΣΜΑ:

Μιας και η συνάρτηση ορίστηκε ΠΡΙΝ τα σημεία των δύο κλήσεων, ΘΕΩΡΕΙΤΑΙ ΓΝΩΣΤΗ.

Επομένως δεν δημιουργείται πρόβλημα και άρα δεν είναι απαραίτητο το πρωτότυπο.

ΜΗΝ ΤΟ ΚΑΝΕΤΕ!!

ΠΑΝΤΑ ΝΑ ΓΡΑΦΕΤΕ ΤΑ ΠΡΩΤΟΤΥΠΑ!!

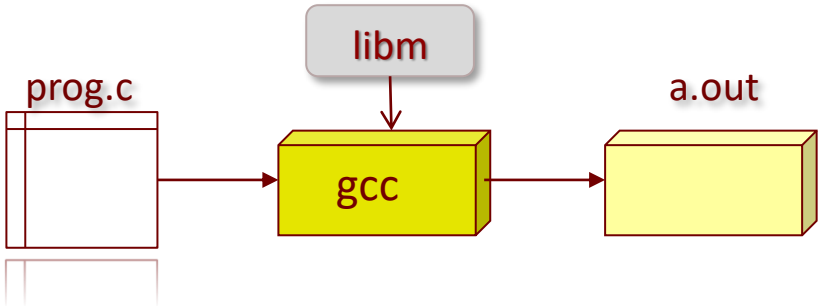
Έτοιμες μαθηματικές συναρτήσεις στη C

- Πρέπει:

```
#include <stdio.h>
#include <math.h>

int main()
{
  printf("cos(0) = %lf\n", cos(0.0));
  return 0;
}
```

- Επίσης, κατά τη μεταγλώττιση, πρέπει να συμπεριληφθεί και η βιβλιοθήκη των μαθηματικών συναρτήσεων με το **-lm:**

```
% gcc prog.c -lm
% ls
a.out prog.c
```



Συνήθεις μαθηματικές συναρτήσεις

```
double acos(double);
double asin(double);
double atan(double);
double cos(double);
double cosh(double);
double sin(double);
double sinh(double);
double tan(double);
double tanh(double);
```

```
double sqrt(double);
double exp(double);
double pow(double, double);
double log(double);
double log10(double);
double fabs(double);
```

Κλήση συναρτήσεων: παράμετροι

- Τι περιμένουμε να εκτυπωθεί στο παρακάτω; Τι τιμή θα έχει το x και τι το a στην main()?

```
void f(int a) {
  a++;
}

int main() {
  int x = 0, a = 0;

  f(x);
  f(a);
  printf("x = %d, a = %d\n", x, a);
  return 0;
}
```

- Μπορεί η f() να αλλάξει την τιμή μιας μεταβλητής της main()? Μπορούμε να επιλέξουμε εμείς πότε να το κάνει;

Πέρασμα παραμέτρων

- Οι μεταβλητές βασικού τύπου (int, char, float, …) περνιούνται ως παράμετροι σε μια συνάρτηση **με τιμή (by value)** .
  - δηλαδή αντιγράφονται οι τιμές τους σε τοπικές μεταβλητές της συνάρτησης.
  - όποιες αλλαγές γίνουν σε αυτές τις τιμές των μεταβλητών στο εσωτερικό της συνάρτησης δεν είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης.
- Αν θέλουμε να ξεπεράσουμε τον παραπάνω περιορισμό, δηλ. ότι αλλαγές γίνουν στις τιμές των μεταβλητών στο εσωτερικό της συνάρτησης να είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης, μπορούμε να περάσουμε τις διευθύνσεις των θέσεων μνήμης που έχουν δεσμευθεί για τις μεταβλητές, να κάνουμε όπως λέμε **πέρασμα με αναφορά** (**by reference**, με χρήση παραμέτρων τύπου **pointer**).
- Οι παράμετροι τύπου πίνακα πάντα περνιούνται με αναφορά.
  - όποιες αλλαγές γίνουν στα στοιχεία του πίνακα είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης.

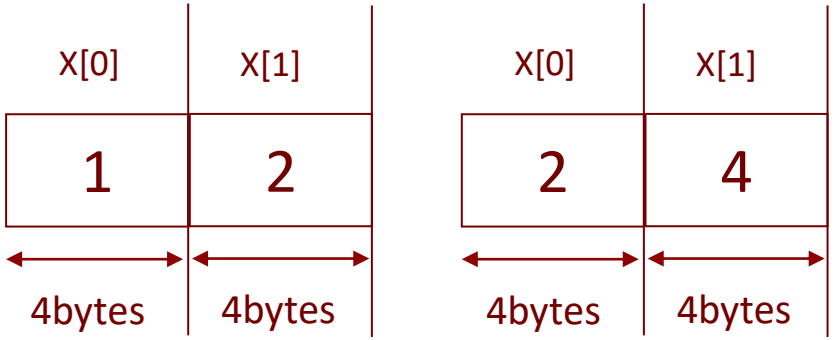
swap

```
#include <stdio.h>
void swap (int x, int y);

int main() {
   int a,b;
   a = 6;
   b = 7;
   swap(a, b);
   printf("%d %d\n", a, b);
   return 0;
}

void swap (int x, int y) {
   int temp;
   temp = x;
   x = y;
   y = temp;
   return;
}
```

```
#include <stdio.h>
void swap (int *x1, int *x2);

int main() {
   int a,b;
   a = 6;
   b = 7;
   swap(&a, &b);
   printf("%d %d\n", a, b);
   return 0;
}

void swap (int *x1, int *x2) {
   int temp;
   temp = *x1;
   *x1 = *x2;
   *x2 = temp;
   return;
}
```

Παράδειγμα πέρασμα by reference με πίνακα

```
#include <stdio.h>
void f(int a[2]);

int main() {
   int i;
   int x[] = {1, 2};
   f(x);
   for (i = 0; i < 2; i++)
       printf("%d\n", x[i]);
   return 0;
}

void f(int k[2]) {
   int i;
   for (i = 0; i < 2; i++)
       k[i] = k[i]*2;
}
```



Στοίβα (stack)

- Για κάθε κλήση συνάρτησης έχω ένα κομμάτι μνήμης αφιερωμένο σε αυτή
- Δεσμεύεται αυτόματα κατά την είσοδο στη συνάρτηση και αποδεσμεύεται επίσης αυτόματα κατά την έξοδο από αυτή
- Ο χώρος μνήμης που χρησιμοποιείται για το σκοπό αυτό καλείται **stack** (στοίβα)
  - Το συγκεκριμένο κομμάτι μνήμης για μια συνάρτηση καλείται **stack frame**
- Στη μνήμη αυτή (**stack frame**) τοποθετούνται οι παράμετροι και οι τοπικές μεταβλητές μιας συνάρτησης

Η γλώσσα C

Εμβέλεια και διάρκεια ζωής μεταβλητών

(scope & lifetime)

MYY502

Εμβέλεια μεταβλητών (scope / visibility)

- Τι είναι: Η **εμβέλεια** του ονόματος μιας μεταβλητής είναι το **τμήμα του προγράμματος στο οποίο η μεταβλητή μπορεί να χρησιμοποιηθεί**
  - Global (καθολική)
  - Local (τοπική, σε συνάρτηση)
  - Block (σε δομημένο μπλοκ εντολών { .. })

- Εμβέλεια τοπικών μεταβλητών → δηλώνονται στην αρχή κάθε συνάρτησης και μπορούν να χρησιμοποιηθούν μόνο μέσα στη συνάρτηση
- Εμβέλεια block → δηλώνεται στην αρχή ενός δομημένου block { … } και είναι ορατό *μόνο μέσα στο block αυτό*!
- Εμβέλεια καθολικών μεταβλητών → από εκεί που δηλώθηκαν μέχρι τέλος αρχείου

Εμβέλεια

```
int i=2, j=7;

int f(int i) {
  return i+3;
}

int g() {
   int k;
   if (j > 5) {
     int j;
     j = f(i);
     k = 2*j;
   }
   return (k);
}

main() {
  i = g();
}
```

Η εμβέλεια προκύπτει ΜΟΝΟ από το κείμενο του κώδικα – απλά παρατηρώντας που είναι τοποθετημένες οι δηλώσεις σε σχέση με τις συναρτήσεις και τα blocks

ΚΑΙ ΌΧΙ

από το πως εκτελείται ο κώδικας.

Τι θα τυπωθεί;

```
#include <stdio.h>
char color = 'B';

void paintItGreen() {
        char color = 'G';
        printf("color@pIG: %c\n", color);
}
void paintItRed() {
        char color = 'R';
        printf("\n\t-------start pIR----\n\t");
        printf("color@pIR: %c\n\t", color);
        paintItGreen();
        printf("\tcolor@pIR: %c\n\t", color);
        printf("-------end pIR----\n\n");
}

main() {
        printf("\t\tcolor@main: %c\n", color);
        paintItGreen();
        printf("\t\tcolor@main: %c\n", color);
        paintItRed();
        printf("\t\tcolor@main: %c\n", color);
}
```

Διάρκεια (lifetime)

- Το **χρονικό διάστημα για το οποίο δεσμεύεται μνήμη για τις μεταβλητές**.
- Τοπικές μεταβλητές: ως την ολοκλήρωση της συνάρτησης στην οποία είναι ορισμένες
- Καθολικές: ως την ολοκλήρωση της εκτέλεσης του προγράμματος

- **Τοπικές** μεταβλητές:
  - auto (το default, υπάρχουν όσο διαρκεί το block / συνάρτηση, μιας και αποθηκεύονται συνήθως στη στοίβα)
  - **static** (διατηρείται ο χώρος ακόμα και αν τελειώσει μία συνάρτηση)

Διάρκεια – μνήμη

```
int i=2, j=7;

int f(int i) {
  return i+3;
}

int g() {
   int k;
   if (j > 5) {
     int j;
     j = f(i);
     k = 2*j;
   }
   return (k);
}

main() {
  i = g();
}
```

Η διάρκεια προκύπτει από την εκτέλεση του κώδικα .

Μεταβλητές με διάρκεια static

```
#include <stdio.h>

void f(void) {
   static int y = 0;
   y++;
   printf("%d\n", y);
}

int main() {
   int i;
   for (i=0; i<5; i++)
       f();
   return 0;
}
```

1. Η αρχικοποίηση γίνεται κατά την εκκίνηση του προγράμματος – δεν γίνεται στην κάθε κλήση της συνάρτησης.
2. Η μεταβλητή συνεχίζει να υπάρχει στη μνήμη ακόμα και μετά τη λήξη της συνάρτησης

Αρχικοποίηση μεταβλητών: σύνοψη

- Με τη δήλωση:
  - Οι καθολικές (global) και οι τοπικές static
    - **Αρχικοποιούνται ΑΥΤΟΜΑΤΑ στο μηδέν (0)**
  - Οι τοπικές (εκτός των static)
    - **Τυχαία αρχική τιμή (συνήθως σκουπίδια)**
- Αρχικοποίηση πινάκων:
  - `int a[] = {1, 2, … };`
    - Από το πλήθος των στοιχείων προκύπτει το μέγεθος του πίνακα
  - `int a[5] = {1, 2, … };`
    - Αν παραπάνω από 5 στοιχεία στις αγκύλες → Error!
    - Αν λιγότερα από 5 → 0 στις υπόλοιπες θέσεις του a.
- Αρχικοποίηση πινάκων χαρακτήρων:
  - `char txt[6] = {'A', 'l', 'a', 'l', 'a', '\0'};`
  - ή πιο εύκολα: `char txt[6] = "Alala";`

Η γλώσσα C

Εμβέλεια extern &
πολλαπλά αρχεία κώδικα

MYY502

Πολλαπλά αρχεία κώδικα - scope.c

```
#include <stdio.h>
#define SCALE 2
int number;     /* Δήλωση/ορισμός καθολικής μεταβλητής */
int f(int param);  /* Πρωτότυπο */

int main() {
   int y;
   number = 5;
   y= f(4*SCALE);
   printf("%d, %d\n", number, y);
   return 0;
}

int f(int x) {
   int y;
   y = x*SCALE + number;
   return (y);
}
```

Σε δύο αρχεία

scope.c

```
#include <stdio.h>
#define SCALE 2

int number,   /* Δήλωση/ορισμός */
    f(int param);  /* Πρωτότυπο */

int main() {
   int y;
   number = 5;
   y = f(4*SCALE);
   printf("%d,%d\n",number,y);
   return 0;
}
```

func.c

```
#define SCALE 2

/* Μεταβλητή ορισμένη αλλού */
extern int number;

int f(int x) {
   int y;
   y = x*SCALE + number;
   return (y);
}
```

Με αρχείο επικεφαλίδας

scope.c

```
#include <stdio.h>
#include "myscope.h"

int number;

int main() {
    int y;
    number = 5;
    y = f(4*SCALE);
    printf("%d,%d\n",number,y);
    return 0;
}
```

func.c

```
#include  "myscope.h"

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

myscope.h

```
#define SCALE 2
int f(int param);
extern int number;
```

Πολλά αρχεία

- Χρήση συναρτήσεων από το ένα στο άλλο
  - Δήλωση prototype
  - Εναλλακτικά, δημιουργία include file με prototypes
  - Γενικός ρόλος που παίζουν τα include files?

- Πώς τα κάνω compile?
  - Είτε όλα μαζί:
    - gcc *.c
  - Είτε με χρήση **Makefile** γιατί πάντα ελέγχει τι έχει αλλάξει

Με Makefile

scope.c

```
#include <stdio.h>
#include "myscope.h"

int number;

int main() {
    int y;
    number = 5;
    y = f(4*SCALE);
    printf("%d,%d\n",number,y);
    return 0;
}
```

func.c

```
#include  "myscope.h"

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

myscope.h

```
#define SCALE 2
int f(int param);
extern int number;
```

Makefile

```
scope: scope.o func.o myscope.h
        gcc -o scope scope.o func.o

func.o: func.c myscope.h
        gcc -c func.c

scope.o: scope.c myscope.h
        gcc -c scope.c
```

scope.c

Makefile

```
scope: scope.o func.o myscope.h
    gcc –o scope scope.o func.o
func.o:  func.c myscope.h
    gcc –c func.c
scope.o: scope.c myscope.h
    gcc –c scope.c
```

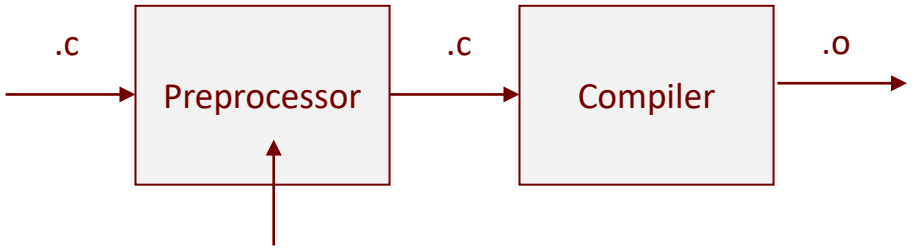
Κανόνας

στόχος (target)

αρχεία απ' τα οποία εξαρτάται (προαπαιτούμενα)

εντολή / ενέργεια για να γίνει (η γραμμή αρχίζει με **TAB**!!!)

Για κάθε αρχείο : `gcc -c x.c`



Αρχεία .o

άλλα αρχεία .o

```
gcc –o scope.exe scope.o f.o
```