

# Προγραμματισμός σε C

---

*Πράξεις με bits  
(bitwise operators)*



# Όλοι οι τελεστές για πράξεις με bits

Τελεστής	Περιγραφή
$x \& y$	AND bit-προς-bit
$x   y$	OR bit-προς-bit
$x \wedge y$	XOR bit-προς-bit
$\sim x$	Αντιστροφή των bits ενός αριθμού
$x \ll n$	Binary Left Shift Operator. Αριστερή μετατόπιση των bits ενός κατά $n$ θέσεις.
$x \gg n$	Binary Right Shift Operator. Δεξιά μετατόπιση των bits ενός κατά $n$ θέσεις.

# Παράδειγμα

```
int main() {
    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13; /* 13 = 0000 1101 */

    int c = 0; c = a & b; /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c );

    c = a | b; /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c );

    c = a ^ b; /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c );

    c = ~a; /* -61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c );

    c = a << 2; /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c );

    c = a >> 2; /* 15 = 0000 1111 */
    printf("Line 6 - Value of c is %d\n", c );
    return 0;
}
```

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

# Άλλα παραδείγματα

- ❖ Ο αριθμός **0...00100...0** (k μηδενικά στα δεξιά του 1) – ποιος είναι;
  - Είναι ο  $2^k$ . Πώς τον φτιάχνω;  
 $x = (1 \ll k);$
- ❖ Ο αριθμός **1...11011...1** (k μονάδες στα δεξιά του 0) – πώς τον φτιάχνω;  
 $x = \sim(1 \ll k);$  /\* Αντιστρέφω τα bits του 0...00100...0 \*/
- ❖ Ο αριθμός **0...00111...1** (k μονάδες) – πώς τον φτιάχνω;
  - Πρόκειται για τον αριθμό 0...00100...0 (k μηδενικά στα δεξιά του 1) **ΜΕΙΟΝ 1**  
 $x = (1 \ll k) - 1;$  /\* 2-to-k minus 1 \*/
- ❖ Διαίρεση δια / πολλαπλασιασμός επί  $2^n$ :  
 $x = x \gg n; y = y \ll n;$  /\* <<1 είναι πολ/μός επί 2 \*/
- ❖ Κάνε το 1<sup>ο</sup> bit (από δεξιά) ίσο με 1:  
 $x = x | 1;$  /\* OR με το 0...001 \*/
- ❖ Κάνε το 1<sup>ο</sup> bit (από δεξιά) ίσο με 0:  
 $x = x \& (\sim 1);$  /\* AND με το 1...110 \*/
- ❖ Έλεγχος αν ο αριθμός είναι περιττός:  
 $\text{if } (x \& 1)$  /\* TRUE αν το 1<sup>ο</sup> bit είναι 1 (περιττός) \*/

## Κι άλλα παραδείγματα («μάσκες»)

- ❖ Οι μάσκες είναι αριθμοί που χρησιμοποιούνται για να εξαγάγουμε ή να θέσουμε σε κάποια τιμή τα bits ενός άλλου αριθμού.
- ❖ Ποιο είναι το λιγότερο σημαντικό bit ενός  $x$ ?

```
y = x & 1;           /* AND με το 000...01 (μάσκα) */
```

- ❖ Το 1<sup>ο</sup> byte ενός ακεραίου

```
y = x & 255;           /* AND με το 000...011111111 */
```

```
y = x & 0xFF;           /* ισοδύναμο */
```

- ❖ Το 2<sup>ο</sup> byte ενός ακεραίου

```
y = (x >> 8) & 0xFF;    /* ολίσθηση δεξιά 8 θέσεις */
```

- ❖ **Θέσε** το 6<sup>ο</sup> bit από δεξιά ίσο με 1:

```
x = x | 32; /* OR με το 32 = 25 = 000...0100000 */
```

- ❖ Μηδένισε το 6<sup>ο</sup> bit:

```
x = x & (~32);    /* AND με το 111...1011111 */
```

- ❖ Αντέστρεψε το 6<sup>ο</sup> bit:

```
x = x ^ 32;      /* XOR με το 000...0100000 */
```

# Προγραμματισμός σε C

---

*Απαριθμήσεις  
(enum)*



# Απαριθμήσεις (enums)

- ❖ Φανταστείτε ότι θέλουμε να ορίσουμε πολλές ακέραιες σταθερές με συνεχόμενες τιμές. Πώς το επιτυγχάνουμε αυτό; Με πολλά #define.
- ❖ Παράδειγμα: μία μεταβλητή φυλάει την ημέρα της εβδομάδας:

```
#define MON 1
#define TUE 2
#define WED 3
#define THU 4
#define FRI 5
#define SAT 6
#define SUN 7
```

```
int main() {
    int day = SUN;
    ...
}
```

# Απαριθμήσεις (enum)

- ❖ Η C παρέχει τις απαριθμήσεις (enum) για ευκολότερο ορισμό. Πρόκειται για σύνολο συγκεκριμένων ακέραιων σταθερών.
- ❖ Βελτιώνει την αναγνωσιμότητα και δομή του προγράμματος μιας και φαίνεται σαν να είναι νέος τύπος δεδομένων (δεν είναι, είναι ακέραιοι)
- ❖ Για το προηγούμενο παράδειγμα:

```
/* Ορισμός του enum */  
enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };  
  
int main() {  
    enum weekday day = WED;    /* Δήλωση μεταβλητής */  
    ...  
    if (day != SUN) day++;  
    ...  
}
```



- ❖ Αν δεν οριστεί τιμή για το πρώτο στοιχείο του συνόλου τότε θεωρείται ότι είναι το 0
- ❖ Μπορούμε να δώσουμε ότι τιμές θέλουμε στις σταθερές. Αν δεν δώσουμε, τότε συνεχίζουν από την τελευταία +1 κάθε φορά.
  - Π.χ. στο παρακάτω, τα `third` και `fourth` είναι το 5 και 6 αντίστοιχα:  

```
enum ival { first=1, second = 4, third, fourth };
```
- ❖ Μπορεί να χρησιμοποιηθεί και με `typedef`:

```
enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };  
typedef enum weekday weekday_t;
```

```
int main() {  
    weekday_t day = WED;  
    ...  
}
```

## ❖ Ψευτο-boolean:

```
enum bool { FALSE, TRUE };  
typedef enum bool boolean;  
int main() {  
    boolean x;  
    x = TRUE;  
}
```

## ❖ Έχουν διαφορά τα δύο παρακάτω?

- A) enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };
- B) enum { MON = 1, TUE, WED, THU, FRI, SAT, SUN } weekday;

## ❖ ΝΑΙ, το A) ορίζει μία νέα απαρίθμηση και της δίνει το όνομα «weekday» ενώ το B) ορίζει μία **μεταβλητή** «weekday» τύπου απαρίθμησης (όπως και τα struct, έτσι και τα enum επιτρέπεται να μην έχουν όνομα).

# Προγραμματισμός σε C

---

*Συναρτήσεις με μεταβλητό πλήθος ορισμάτων  
(Variadic functions)*



- ❖ “Variadic”
- ❖ Π.χ. η `printf()`. Πόσα ορίσματα παίρνει;
- ❖ Στη C μπορούμε να ορίσουμε συναρτήσεις με άγνωστο πλήθος παραμέτρων/ορισμάτων.
  - Όμως πρέπει να υπάρχει τουλάχιστον 1 παράμετρος (δεν γίνεται να μην έχει καμία).
- ❖ Απαιτείται η χρήση του `#include <stdarg.h>`
  - Παλιά χρησιμοποιούσαμε το `<varargs.h>` αλλά όχι πλέον

❖ Ορίζονται όπως όλες οι συναρτήσεις, όμως έχουμε δύο είδη παραμέτρων:

- Πρώτα είναι οι **ΥΠΟΧΡΕΩΤΙΚΕΣ** παράμετροι (τουλάχιστον 1)
- Στη συνέχεια τοποθετούνται οι **ΠΡΟΑΙΡΕΤΙΚΕΣ** (άγνωστο πλήθος), **απλά βάζοντας τρεις τελείες:**

```
int sum(int n, ...) { /* το n υποχρεωτικό */  
    <κώδικας>  
}
```

- ❖ Ακριβώς όπως οι κανονικές συναρτήσεις με όσες παραμέτρους θέλουμε (τουλάχιστον όσες και οι υποχρεωτικές):

```
int sum(int n, ...) { /* το n υποχρεωτικό */  
    <κώδικας>  
}
```

```
int main() {  
    x = sum(2, 3, 5);      /* Άθροισε 2 αριθμούς */  
    y = sum(4, 1, 5, 7, 9); /* Άθροισε τέσσερις */  
    return 0;  
}
```

# Μέσα στη συνάρτηση;

- ❖ Για να μπορέσουμε να βρούμε τις μη-υποχρεωτικές παραμέτρους, πρέπει να ορίσουμε μία μεταβλητή τύπου “va\_list” και να την αρχικοποιήσουμε με την κλήση va\_start():

```
int sum(int n, ...) {    /* το n υποχρεωτικό */
    va_list args;
    int sum = 0;

    /* Αρχικοποίηση βάζοντας το όνομα της τελευταίας
       υποχρεωτικής παραμέτρου */
    va_start(args, n);

    <κώδικας>
}
```

# Μη υποχρεωτικές παράμετροι

- ❖ Η προσπέλαση των παραμέτρων αυτών γίνεται η μία μετά την άλλη, συνήθως μέσα σε ένα loop. Για να πάρω την τιμή της επόμενης τέτοιας παραμέτρου, χρησιμοποιώ την `va_arg()` όπου είναι υποχρεωτικό να γνωρίζω και να αναγράψω τον ΤΥΠΟ της!
- ❖ Πριν την επιστροφή, πρέπει `va_end()`.

```
int sum(int n, ...) {    /* το n υποχρεωτικό */
    va_list args;
    int i, sum = 0, t;

    va_start(args, n);    /* Αρχικοποίηση */
    for (i = 0; i < n; i++) {
        t = va_arg(args, int);
        sum += t;
    }
    va_end(args);          /* Τέλος */
    return (sum);
}
```



- ❖ Πρέπει πάντα με κάποιο τρόπο να γνωρίζω το πλήθος των παραμέτρων αλλιώς μπορεί να «κρασάρει» το πρόγραμμα αν χρησιμοποιήσω την `va_arg()` και:
  - Οι παράμετροι έχουν τελειώσει ή
  - Η παρέμετρος δεν είναι του τύπου που βάλαμε στην `va_arg()`
- ❖ Για αυτό συνήθως
  - η πρώτη παράμετρος φροντίζει να καθορίζει πόσα είναι τα ορίσματα ή
  - βάζω στο τέλος μία παράμετρο-«σημάδι» ώστε αν φτάσω εκεί να τελειώσω.

# Προγραμματισμός σε C

---

*Δείκτες σε συναρτήσεις  
(function pointers)*



# Δείκτες σε συνάρτηση

- ❖ Η C επιτρέπει να έχουμε δείκτες που δείχνουν σε *συναρτήσεις*!
- ❖ Δηλώνονται ως:  
    <τύπος επιστροφής> (\*όνομα)(<παράμετροι>);
- ❖ Πού χρησιμεύουν; Παράδειγμα: θέλω να κάνω μία συνάρτηση `update()` η οποία αλλάζει τα στοιχεία ενός πίνακα.
  - Κάποιες φορές θέλω να τριπλασιάζονται οι τιμές των στοιχείων.
  - Κάποιες άλλες φορές θέλω να αντιστρέφονται οι τιμές των στοιχείων.
  - Κάποιες άλλες φορές θέλω να αλλάζουν πρόσημο
  - Κλπ κλπ.
  
  - Πρέπει να κάνω διαφορετικές εκδόσεις της `update()` οι οποίες κάνουν ακριβώς τα ίδια πράγματα – διαφέρουν μόνο στην πράξη που κάνουν σε κάθε στοιχείο.

# Πολλά αντίγραφα της update()

```
void update_triple(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = 3*x[i];  
}
```

```
void update_inverse(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = 1/x[i];  
}
```

```
void update_revsign(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = -x[i];  
}
```

- ❖ Όλα τα αντίγραφα είναι ολόιδια (εκτός από την πράξη στο κάθε στοιχείο του πίνακα).

# Ευκολία: δείκτης σε συνάρτηση

- ❖ Κάνω μόνο 1 έκδοση της update() και βάζω ως επιπλέον όρισμα τη λειτουργία που πρέπει να γίνει στα στοιχεία. Οι λειτουργίες υλοποιούνται με ξεχωριστές συναρτήσεις:

```
float triple(float f) { return (3*f); }  
float inverse(float f) { return (1/f); }  
float revsign(float f) { return (-f); }
```

```
void update(int n, float x[], float (*operation)(float)) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = (*operation)(x[i]); /* Ή ισοδύναμα: operation(x[i]) */  
}  
  
int main()  
{  
    float array[5] = { 1.5, 2.5, 3.5, 4.5, 5.5 };  
    float (*func)(float); /* Μεταβλητή func - δείκτης σε συνάρτηση */  
  
    func = triple; /* Δείχνει στη συνάρτηση triple */  
    update(5, array, func);  
    func = inverse; /* Δείχνει στη συνάρτηση inverse */  
    update(5, array, func);  
    update(5, array, revsign);  
    return 0;  
}
```

# Έτοιμη συνάρτηση στο stdlib.h

## ❖ Quicksort:

```
void qsort(void *start, int numelems, int size,  
           int (*compare)(void *, void *));
```

## ❖ Παράδειγμα κώδικα για ταξινόμηση ακεραίων κατά αύξουσα σειρά :

```
/* Πρέπει να επιστρέφει (όπως η strcmp()):  
   0 αν ίσα, < 0 αν a < b και > 0 αν a > b */  
int cmp(void *a, void *b) {  
    int x = *((int *) a),  
        y = *((int *) b);  
    return ( x-y );  
}  
  
int main(int argc, char *argv[]) {  
    int array[100];  
    ...  
    qsort(array, 100, sizeof(int), cmp);  
    ...  
}
```

# Προγραμματισμός σε C

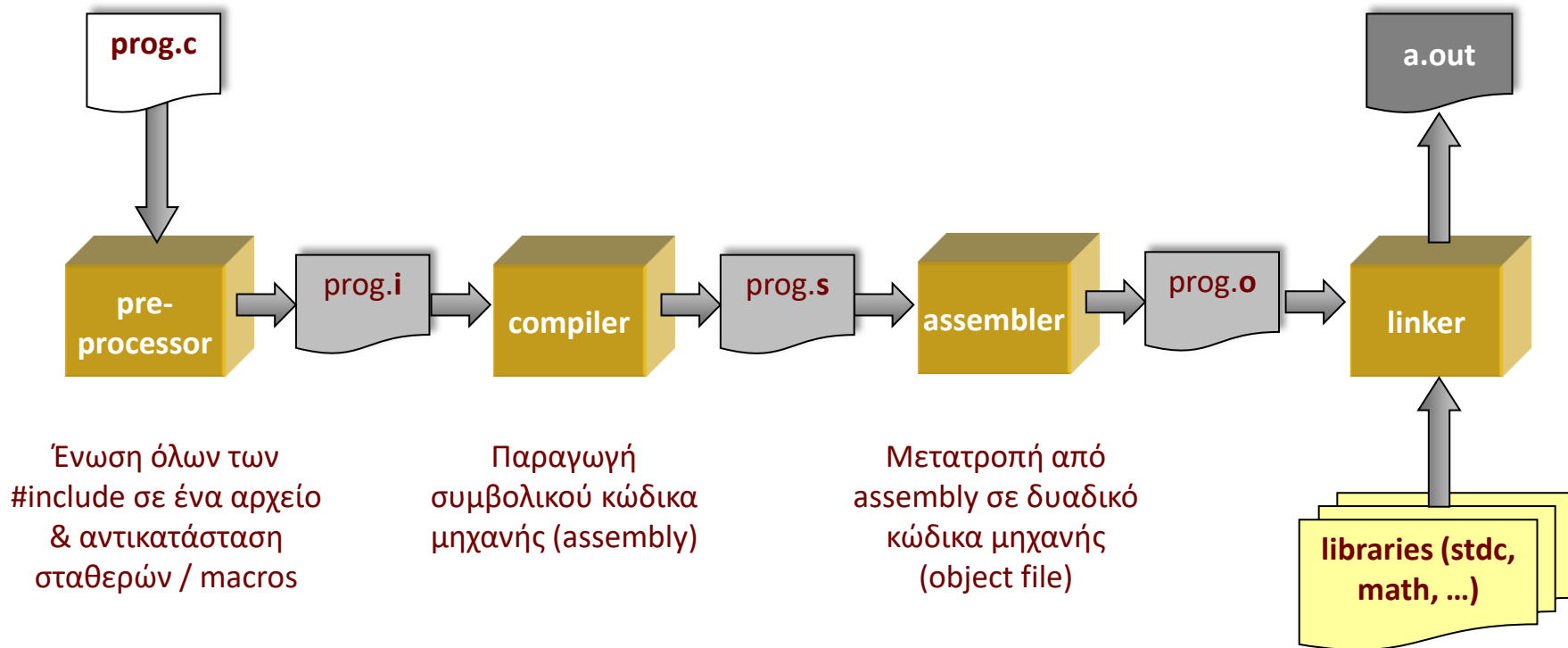
---

*Προχωρημένος προεπεξεργαστής  
(C preprocessor)*



# Διαδικασία/Αλυσίδα μετάφρασης (π.χ. gcc)

❖ Τι γίνεται ακριβώς όταν μεταφράζεται ένα prog.c??



- Όλα τα ενδιάμεσα αρχεία μπορείτε να τα δείτε με: **gcc --save-temps prog.c**
- Για να τρέξει μέχρι τον preprocessor: **gcc -E prog.c**
- Για να τρέξει μέχρι τον compiler: **gcc -S prog.c**
- Για να τρέξει μέχρι τον assembler: **gcc -c prog.c**



1. «Πετάει» όλα τα σχόλια.
2. Ενώνει το πρόγραμμά μας μαζί με όλα τα αρχεία που κάνουμε `#include` σε ΕΝΑ αρχείο
3. Κάνει αντικατάσταση των σταθερών / macros (`#define`) που έχουμε στο πρόγραμμά μας
4. Το τελικό αποτέλεσμα τοποθετείται στο αρχείο `prog.i`
5. Το `prog.i` είναι αυτό που μεταφράζεται τελικά!

## ❖ Παρατηρήσεις:

- Οι σταθερές αντικαθίστανται ΠΡΙΝ το `compile` – άρα ο μεταφραστής δεν τις γνωρίζει!
- Ο preprocessor έχει μία μίνι-γλώσσα δική του, η οποία μπορεί να χρησιμοποιηθεί για διάφορα προχωρημένα τεχνάσματα.

# Απλός preprocessor

## ❖ Ορισμός σταθερών – και όχι μόνο!!

```
#define N          0
#define MINUS1    4-5      /* ΔΕΝ κάνει υπολογισμούς */
#define KEYBOARD  stdin
#define then      /* τίποτε */
#define START    {
#define END      }
#define msg      "Please enter a positive number\n"
#define str       "hi!\n"
#define longloop for (i = 0; i < n; i++)\
                    printf("%d ", i)
```

```
int main()
START
    int n = MINUS1, i;

    fscanf(KEYBOARD, "%d", &n);
    if (n < N)
        then {
            printf(str);
            printf(msg);
        END
    else START
        printf(str);
        longloop;
    }
    return 0;
END
```

### Αποτέλεσμα preprocessor (.i)

```
int main()
{
    int n = 4-5, i;

    fscanf(stdin, "%d", &n);
    if (n < 0) {
        {
            printf("hi!\n");
            printf("Please enter a positive number\n");
        }
    }
    else {
        printf("hi!\n");
        for (i = 0; i < n; i++) printf("%d ", i) ;
    }
    return 0;
}
```

## ❖ Macro: παραμετροποιημένη έκφραση / αντικατάσταση

```
#define inch2cm(i) i*2.54
#define Square(x) x*x
int main() {
    int    a, b;
    float l;
    a = Square(4);
    b = Square(a);
    l = inch2cm(2.0);
    ...
}
```

### Αποτέλεσμα preprocessor (.i)

```
int main() {
    int    a, b;
    float l;
    a = 4*4;
    b = a*a;
    l = 2.0*2.54;
    ...
}
```

- ❖ ΔΕΝ ΕΙΝΑΙ ΣΥΝΑΡΤΗΣΕΙΣ! Η αντικατάσταση γίνεται άμεσα (πριν τη μετάφραση).
- ❖ Το «όρισμα» της μακροεντολής αντικαθίσταται ως έχει
  - Αυτό δημιουργεί προβλήματα και θέλει προσοχή!

❖ Τι θα γίνει παρακάτω;

```
#define Square(x) x*x
```

```
int main() {  
    int a, b;  
    a = Square(2+2);  
    ...  
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {  
    int a, b;  
    a = 2+2*2+2;    /* 8 (όχι 16!) */  
    ...  
}
```

❖ **ΠΑΝΤΑ ΠΡΕΠΕΙ ΝΑ ΒΑΖΟΥΜΕ ΠΑΡΕΝΘΕΣΕΙΣ ΣΤΙΣ ΠΑΡΑΜΕΤΡΟΥΣ, όπου αυτές εμφανίζονται στον ορισμό του macro.**

# Παράμετροι και παρενθέσεις

❖ Τι θα γίνει παρακάτω;

```
#define Square(x) (x)*(x)
#define Double(x) (x)+(x)
```

```
int main() {
    int a, b;
    a = Square(2+2);
    b = Square(-1);
    a = Double(a+b);
    b = 5*Double(a);
    ...
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {
    int a, b;
    a = (2+2)*(2+2);    /* Σωστό */
    b = (-1)*(-1);      /* Σωστό */
    a = (a+b)+(a+b)     /* Σωστό */
    b = 5*(a)+(a)       /* !!! */
    ...
}
```

❖ Όταν πρέπει, **ΒΑΖΟΥΜΕ ΠΑΡΕΝΘΕΣΕΙΣ ΚΑΙ ΣΕ ΟΛΗ ΤΗΝ ΕΚΦΡΑΣΗ ΤΟΥ MACRO**

```
#define Double(x) ((x)+(x))
```

## ❖ Με περισσότερα ορίσματα:

```
#define max(a,b) ( ((a) > (b)) ? (a) : (b) )
```

```
int min(int a, int b) {  
    return ( (a < b) ? a : b );  
}
```

```
int main() {  
    int x = 4, y = 5;  
    x = max(x,y);  
    y = min(x,y);  
    return 0;  
}
```

## Αποτέλεσμα preprocessor (.i)

```
int min(int a, int b) {  
    return ( (a < b) ? a : b );  
}  
  
int main() {  
    int x = 4, y = 5;  
    x = ( ((x) > (y)) ? (x) : (y) );  
    y = min(x,y);  
    return 0;  
}
```

## ❖ Macro ή συνάρτηση?

- Εδώ η συνάρτηση είναι «χρονοβόρα» καθώς μεταφέρονται ορίσματα και γίνεται η κλήση της κατά τη διάρκεια εκτέλεσης του προγράμματος.
- Το συγκεκριμένο macro μεταφράζεται άμεσα και ο υπολογισμός του γίνεται χωρίς κλήση σε συνάρτηση.
- Το macro δεν γνωρίζει τύπους (και καλό και κακό...)

# #undef

- ❖ Κατάργηση μίας σταθεράς / μακροεντολής από ένα σημείο και κάτω.

```
#define PI 3.14
```

```
int main() {  
    double x, y;  
    x = PI;  
#undef PI  
    y = PI;  
    ...  
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {  
    double x, y;  
    x = 3.14;  
  
    y = PI;    /* Error: unknown  
                identifier */  
    ...  
}
```

- ❖ Πολλές φορές δεν γνωρίζουμε αν κάποιες σταθερές / συναρτήσεις υπάρχουν στο σύστημά μας. Π.χ. το NULL είναι ορισμένο κάπου? Το TRUE, το FALSE? Πρέπει να κάνουμε `#include` κάποιο header για να οριστεί?
- ❖ Αν πάμε και κάνουμε μόνοι μας π.χ. `#define NULL 0`, ενώ ήδη υπάρχει κάπου ορισμένο, τότε στην καλύτερη περίπτωση θα «γκρινιάξει» ο compiler για re-definition.
- ❖ Μπορούμε να κάνουμε κάτι για αυτό;
- ❖ Απάντηση: conditional compilation
  - Μπορούμε να ελέγχουμε αν κάτι έχει ήδη γίνει `#define` και να μην το ξανακάνουμε εμείς.
  - `#ifdef` / `#ifndef` / `#else` / `#endif`



# Παραδείγματα conditional compilation

```
#ifdef FALSE
    /* Τίποτε */
```

```
#else
    #define FALSE 0
#endif
```

```
#ifndef TRUE
    #define TRUE 1
#endif
```

```
#ifdef __unix__
    #include <unistd.h>
#else
    #ifdef _WIN32
        #include <windows.h>
    #endif
#endif
```

```
/* Ανάλογα με το σύστημα */
```

## ❖ Γενική σύνταξη

```
#if <condition>
...
#elif <condition>
...
#elif <condition>
...
...
#else
...
#endif
```

## ❖ Η συνθήκη είναι απλή έκφραση που εμπλέκει σταθερές συν το defined(). Παράδειγμα:

```
#if !defined(TRUE)      /* Ισοδύναμο με το #ifndef */
    #define TRUE 1
#endif
```

# Γενικότερη σύνταξη

```
#if defined(NOTHING)
    #define DEBUG_LEVEL 0
#elif defined(SIMPLE) && !defined(ADVANCED)
    #define DEBUG_LEVEL 1
#elif defined(ADVANCED) || defined(Advanced)
    #define DEBUG_LEVEL 2
#else
    #define DEBUG_LEVEL 0
#endif

main() {
    ...
    #if DEBUG_LEVEL==2
        printf("...
    #endif
    ...
}
```

# Τεχνική διατήρησης τμήματος κώδικα

- ❖ Θέλουμε το παρακάτω κομμάτι κώδικα να το «βάλουμε» σε σχόλια – δηλ. να μην το σβήσουμε (γιατί θα χρησιμοποιήσουμε στο μέλλον):

```
if (x > 0) { /* test for positive */
    ...
}
else {      /* negative */
    ...
}
```

- ❖ Πώς; Αν απλά το βάλουμε σε σχόλια `/* ... */`, θα προκύψει λάθος από τη μετάφραση μιας και ήδη έχουμε σχόλια μέσα στο τμήμα αυτό του κώδικα.
- ❖ «Κλασική» λύση:

```
#if 0      /* Πάντα false => δεν μεταφράζεται! */
if (x > 0) { /* test for positive */
    ...
}
else {      /* negative */
    ...
}
#endif
```