Η γλώσσα C

Περί C

- Είναι μια γλώσσα «προστακτικού» (imperative) προγραμματισμού με σχετικά λίγες εντολές.
- Έχει «αδύναμο» σύστημα ελέγχου τύπων.
- Η ελάχιστη μονάδα υποπρογράμματος (δόμησης κώδικα) είναι η συνάρτηση (function).
- Ένα πρόγραμμα C μεταφράζεται και μετά εκτελείται.
- Υποστηρίζεται ξεχωριστή μετάφραση διαφορετικών υποπρογραμμάτων με στατική ή δυναμική σύνδεση.
- Είναι ιδιαίτερα διαδεδομένη, σε επίπεδο υλοποίησης λειτουργικών συστημάτων αλλά και εφαρμογών.
- Οι γλώσσες C++ και Java είναι «απόγονοι» της C.

Μορφοποίηση κειμένου

- Σχεδόν κάθε εντολή τερματίζεται με '`;`'.
- Μπορούμε να γράφουμε πολλές διαδοχικές εντολές στην ίδια γραμμή του κειμένου.
- Χρησιμοποιούμε τον όρο «σώμα» ή «μπλοκ» για να αναφερθούμε σε μια ή περισσότερες εντολές που δίνονται ανάμεσα σε '`{`' και '`}`'.
- Σχόλια (κείμενο που δεν λαμβάνει υπ' όψη του ο μεταφραστής) δίνονται ανάμεσα σε ' `/*`' και ' `*/`' .
- Κενοί χαρακτήρες και γραμμές που βρίσκονται ανάμεσα σε εντολές δεν λαμβάνονται υπ' όψη από τον μεταφραστή – χρησιμεύουν, όπως και τα σχόλια, αποκλειστικά για την αναγνωσιμότητα του κώδικα.

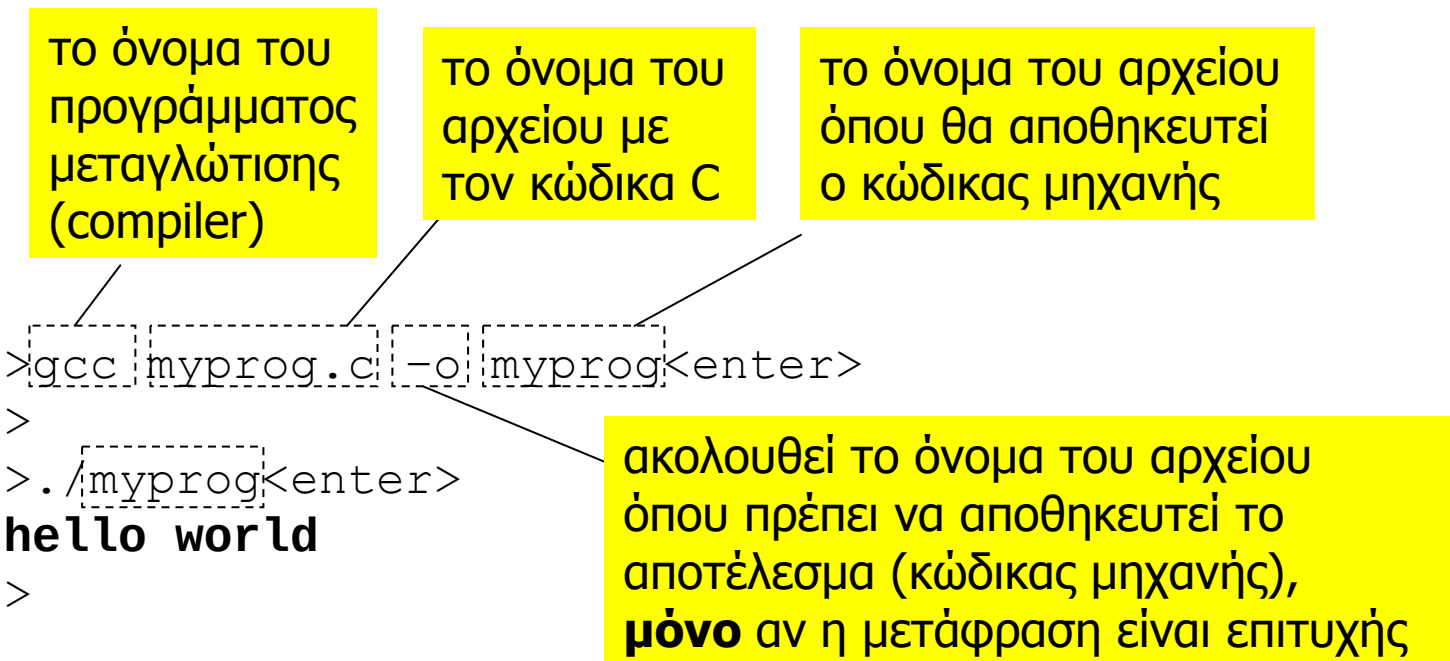
Η συνάρτηση ως υποπρόγραμμα

- Η κύρια μονάδα υποπρογράμματος (δόμησης κώδικα σε ξεχωριστή ομάδα εντολών) είναι η συνάρτηση.
- Μια συνάρτηση μπορεί να βασίζεται σε άλλες συναρτήσεις, που με την σειρά τους μπορεί να βασίζονται σε άλλες συναρτήσεις κλπ.
- Η συνάρτηση `main` (κυρίως συνάρτηση) καλείται από το περιβάλλον εκτέλεσης (λειτουργικό) για να αρχίσει η εκτέλεση του προγράμματος – ανεξάρτητα από το αν το πρόγραμμα έχει και άλλες συναρτήσεις.
- Σημείωση: αρχικά, για ευκολία, θα εστιάσουμε σε προγράμματα που αποτελούνται μόνο από την `main`.

Η μορφή της συνάρτησης `main`

```
int main(int argc, char *argv[]) {

  <δηλώσεις μεταβλητών>

  <εντολές εισόδου, εξόδου, ελέγχου και
   εντολές επεξεργασίας μεταβλητών>

}
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  printf("hello world\n");

}
```



```
#include <stdio.h>

int main(int argc, char* argv[]) {

  putchar('h');
  putchar('e');
  putchar('l');
  putchar('l');
  putchar('o');
  putchar(' ');
  putchar('w');
  putchar('o');
  putchar('r');
  putchar('l');
  putchar('d');
  putchar('\n');

}
```

```
>./myprog<enter>
hello world
>
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  putchar('5');
  putchar(' ');
  putchar('+');
  putchar(' ');
  putchar('2');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putchar('9');
  putchar('\n');

}
```

```
>./myprog<enter>
5 + 2 = 9
>
```

Είσοδος / Έξοδος

Εντολές εισόδου / εξόδου

- Οι εντολές εισόδου/εξόδου είναι ειδικές συναρτήσεις που βρίσκονται στην βιβλιοθήκη `stdio` –που πρέπει να χρησιμοποιεί το πρόγραμμα (εντολή `include`).
- `getchar`: διαβάζει ένα (τον επόμενο) χαρακτήρα από τήν είσοδο του προγράμματος (πληκτρολόγιο).
- `putchar`: γράφει ένα χαρακτήρα χαρακτήρα στην έξοδο του προγράμματος (οθόνη).
- `scanf`: διαβάζει από την είσοδο χαρακτήρες και αναθέτει τιμές σε συγκεκριμένες μεταβλητές (διαβάζονται όσοι χαρακτήρες είναι απαραίτητοι).
- `printf`: εκτυπώνει στην έξοδο κείμενο καθώς και τιμές από συγκεκριμένες μεταβλητές.

Οι συναρτήσεις `putchar` και `getchar`

- Η `putchar` εκτυπώνει ένα χαρακτήρα στην έξοδο και η `getchar` διαβάζει ένα (τον επόμενο) χαρακτήρα από την είσοδο του προγράμματος.
- Χρησιμοποιούναι συνήθως όταν επιθυμούμε μια επεξεργασία της εισόδου / εξόδου χαρακτήρα προς χαρακτήρα (αντί της πιο προχωρημένης επεξεργασίας που παρέχουν οι συναρτήσεις `printf` και `scanf`).
- Με τις `putchar` και `getchar`, ο προγραμματιστής μπορεί να υλοποιήσει «δικές» του συναρτήσεις εισόδου/εξόδου, σύμφωνα με την λειτουργικότητα που επιθυμεί.

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  char c1,c2,c3;

  putchar('3'); putchar(' '); putchar('c');
  putchar('h'); putchar('a'); putchar('r');
  putchar('s'); putchar(':'); putchar('\n');

  c1=getchar(); c2=getchar(); c3=getchar();

  putchar(c1); putchar(c2); putchar(c3);

  putchar('\n');
}
```

```
>./myprog<enter>
3 chars:
a2$<enter>
a2$
>
```

```
>./myprog<enter>
3 chars:
a bcdefg<enter>
a b
>
```

το πρόγραμμα δεν διαβάζει τα «επιπλέον» δεδομένα αφού δεν υπάρχουν αντίστοιχες εντολές στον κώδικα

Η συνάρτηση `printf`

- Δέχεται μια παράμετρο σε μορφή συμβολοσειράς από εκτυπώσιμους χαρακτήρες ASCII, και προαιρετικά έναν **απεριόριστο** αριθμό εκφράσεων αποτίμησης.
- Η πρώτη παράμετρος περιέχει (α) το κυριολεκτικό κείμενο προς εκτύπωση, και (β) τους προσδιορισμούς εκτύπωσης για τις τιμές κάθε μιας έκφρασης που δίνεται ως επιπλέον παράμετρος.
- Για τις συμβάσεις των προσδιορισμών εκτύπωσης βλέπε (οπωσδήποτε) το εγχειρίδιο της γλώσσας!
- Σημείωση**: δεν** ελέγχεται η αντιστοιχία ούτε και η συμβατότητα ανάμεσα στους προσδιορισμούς εκτύπωσης και στους τύπους των εκφράσεων αποτίμησης που δίνονται σαν παράμετροι.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int a=1,b=2;

  printf("a is %d, b is %d and a+b is %d\n",a,b,a+b);
}
```

```
>./myprog<enter>
a is 1, b is 2 and a+b is 3
>
```

Η συνάρτηση `scanf`

- Δέχεται μια παράμετρο σε μορφή συμβολοσειράς από εκτυπώσιμους χαρακτήρες ASCII, και έναν **απεριόριστο** αριθμό **διευθύνσεων** μεταβλητών.
- Η πρώτη παράμετρος περιέχει τους προσδιορισμούς ανάγνωσης για τις τιμές κάθε μιας μεταβλητής η **διεύθυνση** της οποίας δίνεται ως παράμετρος.
- Για τις συμβάσεις των προσδιορισμών εκτύπωσης βλέπε (οπωσδήποτε) το εγχειρίδιο της γλώσσας!
- Σημείωση**: δεν** ελέγχεται η αντιστοιχία ούτε και η συμβατότητα ανάμεσα στους προσδιορισμούς ανάγνωσης και στους τύπους των μεταβλητών αποτίμησης που δίνονται σαν παράμετροι.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int a,b;

  printf("enter 2 int values: ");

  scanf("%d %d",&a,&b);

  printf("a is %d and b is %d\n",a,b);
}
```

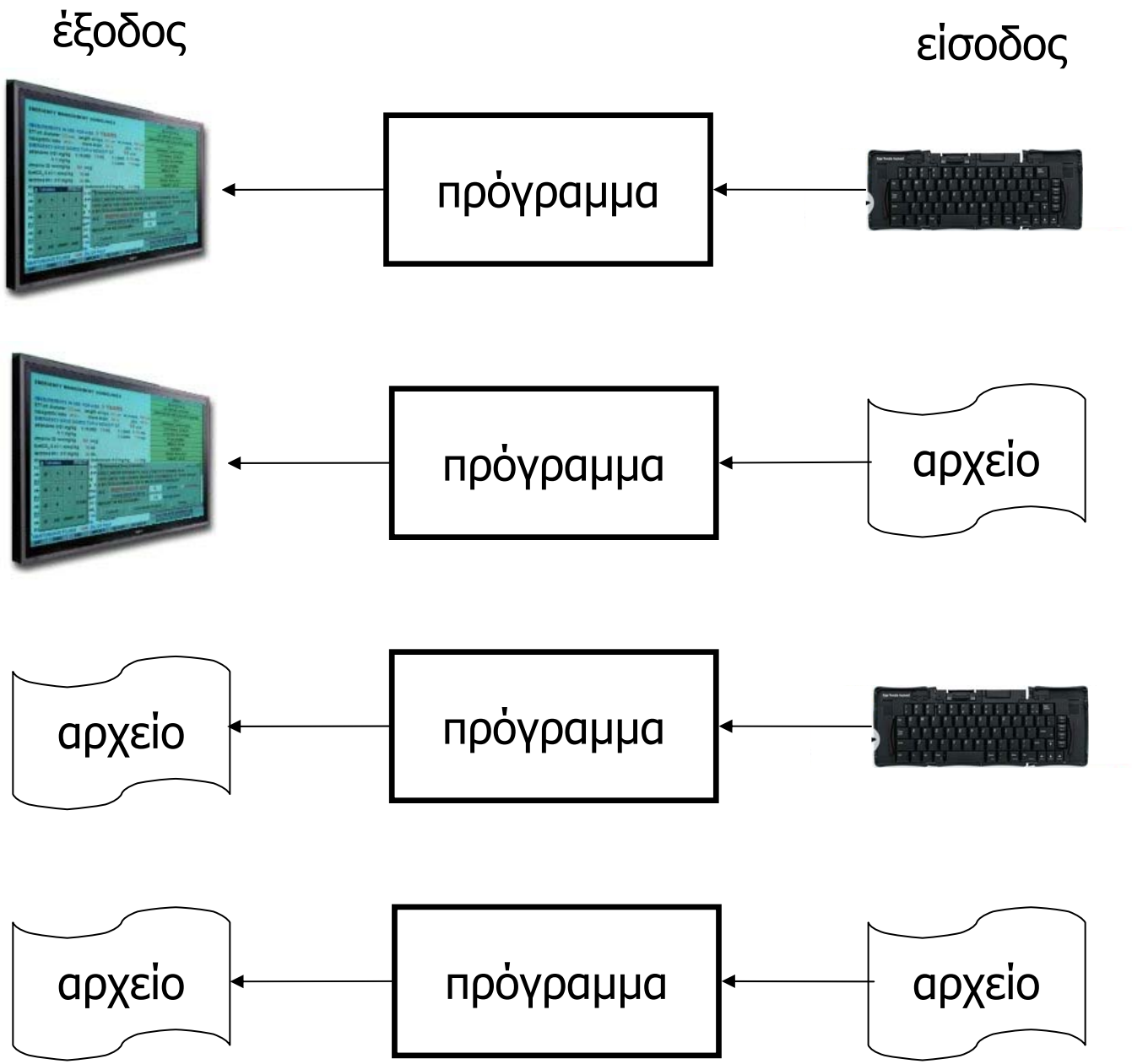
προσοχή: πάντα να υπάρχει το & πριν το όνομα της μεταβλητής (επεξήγηση σε λίγο)

το πρόγραμμα δεν διαβάζει τα «επιπλέον» δεδομένα αφού δεν υπάρχουν αντίστοιχες εντολές στον κώδικα

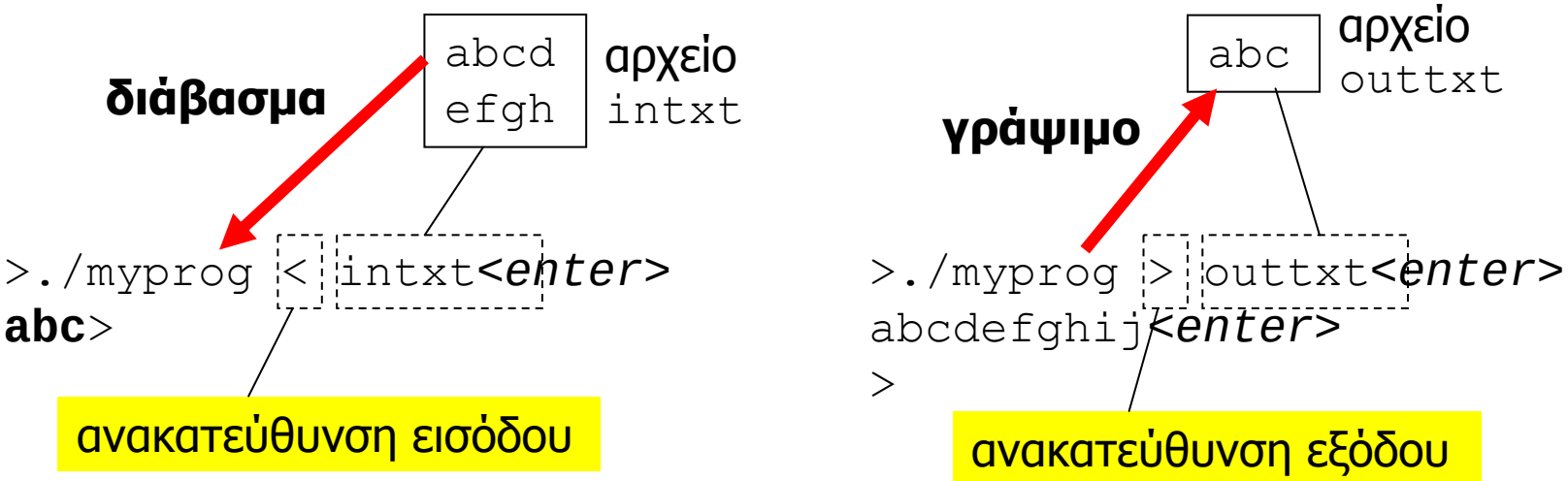
```
>./myprog<enter>
enter 2 int values: 5 10 15<enter>
a is 5 and b is 10
>
```

Ανακατεύθυνση εισόδου / εξόδου

- Οι πράξεις εισόδου / εξόδου διαβάζουν / γράφουν από την είσοδο / έξοδο του προγράμματος.
- Συνήθως η είσοδος αντιστοιχεί στο πληκτρολόγιο (με εκτύπωση των χαρακτήρων που εισάγονται στην οθόνη) και η έξοδος στην οθόνη.
- Τόσο η είσοδος όσο και η έξοδος ενός προγράμματος μπορούν να ανακατευθυνθούν, π.χ. σε αρχεία έτσι ώστε οι πράξεις εισόδου να διαβάζουν δεδομένα από ένα αρχείο και οι πράξεις εξόδου να γράφουν δεδομένα σε ένα αρχείο.
- Τα αρχεία που δίνονται για είσοδο πρέπει να είναι ASCII και τα αρχεία που δημιουργούνται είναι ASCII.


έξοδος
είσοδος
πρόγραμμα
πρόγραμμα
αρχείο
αρχείο
πρόγραμμα
πρόγραμμα
αρχείο

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  char c1,c2,c3;

  c1=getchar(); c2=getchar(); c3=getchar();

  putchar(c1); putchar(c2); putchar(c3);

}
```



Σχόλιο

- Όταν το πρόγραμμα φτάσει σε μια εντολή εισόδου (π.χ. `getchar` ή `scanf`) τότε η εκτέλεση σταματά μέχρι να υπάρξουν δεδομένα έτοιμα προς ανάγνωση.
- Αν τα δεδομένα εισάγονται από το πληκτρολόγιο τότε οι χαρακτήρες «στέλνονται» στο πρόγραμμα **αφού** πατηθεί το πλήκτρο `<enter>` (μέσω του οποίου δημιουργείται αυτόματα και ο χαρακτήρας `'\n'`).
- Αν τα δεδομένα εισόδου δίνονται μέσω ενός αρχείου (με ανακατεύθυνση), και το πρόγραμμα επιχειρήσει να διαβάσει χωρίς να υπάρχουν άλλα δεδομένα τότε επιστρέφεται η τιμή (σταθερά) EOF (end of file).
- Το πρόγραμμα πρέπει να κάνει κατάλληλο έλεγχο.

Βιβλιοθήκη `stdio`

- Οι συναρτήσεις εισόδου/εξόδου υλοποιούνται στην βιβλιοθήκη `stdio` η χρήση της οποίας πρέπει να δηλώνεται στην αρχή του προγράμματος με την εντολή `#include<stdio.h>`
- Εκτός των `getchar`, `putchar`, `scanf`, `printf`, υπάρχουν πολλές άλλες συναρτήσεις (που όμως δεν θα εξετάσουμε στο μάθημα – φυσικά μπορείτε να διαβάσετε μόνοι σας τα εγχειρίδια της γλώσσας).
- Περισσότερα για το τι είναι και πως φτιάχνεται μια βιβλιοθήκη προς το τέλος του μαθήματος ...

Τύποι, Κυριολεκτικά, Μεταβλητές, Τελεστές, Μετατροπές Τύπων

Βασικοί τύποι

όνομα	τύπος / κωδικοποίηση	μέγεθος
`char`	χαρακτήρας ASCII	1
`int`	ακέραιος (2's complement)	2(*)
`float`	πραγματικός απλής ακριβείας	4
`double`	πραγματικός διπλής ακριβείας	8
`void`	μη ύπαρξη τιμής (...)	

(*) Το μέγεθος του τύπου `int` εξαρτάται από την αρχιτεκτονική του επεξεργαστή (2 ή 4 bytes).

Προσδιορισμοί μεγέθους / πεδίου τιμών

όνομα	τύποι / ερμηνεία	μέγεθος
`short`	μικρό `int`	1(*)
`long`	μεγάλο `int` / `double`	4(*) / 16

όνομα	τύποι	πεδίο
`signed`	`int` / `char` με πρόσημο	$[-2^{N-1}...2^{N-1}-1]$
`unsigned`	`int` / `char` χωρίς πρόσημο	$[0...2^{N}-1]$

(*) Το μέγεθος του `short/long int` εξαρτάται από την αρχιτεκτονική του επεξεργαστή (1/4 ή 2/8 bytes).

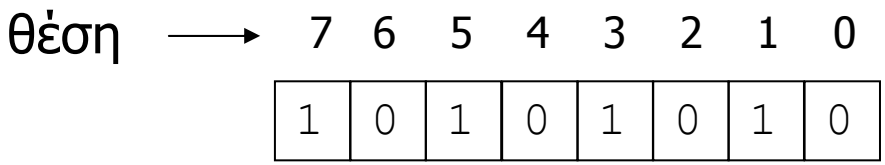
- Όταν οι προσδιορισμοί χρησιμοποιούνται μόνοι τους τότε εννοείται ο τύπος `int`, και όταν δεν αναφέρεται προσδιορισμός για το πρόσημο εννοείται `signed`.

Πίνακας τύπων, μεγεθών και πεδίων τιμών

`unsigned char`	1 byte	0 ... 255
`char`	1 byte	-128 ... 127
`unsigned int`	2 bytes	0 ... 65,535
`short int`	2 bytes	-32,768 ... 32,767
`int`	2 bytes	-32,768 ... 32,767
`unsigned long`	4 bytes	0 ... 4,294,967,295
`long`	4 bytes	-2,147,483,648 ... 2,147,483,647
`float`	4 bytes	1.17549435 * (10^-38) ... 3.40282347 * (10^+38)
`double`	8 bytes	2.2250738585072014 * (10^-308) ... 1.7976931348623157 * (10^+308)
`long double`	16 bytes	3.4 * (10^-4932) ... 1.1 * (10^4932)

```
/* εκτύπωση μεγέθους βασικών τύπων */

#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("sizeof(char)=%d bytes\n",sizeof(char));
  printf("sizeof(short)=%d bytes\n",sizeof(short);
  printf("sizeof(int)=%d bytes\n",sizeof(int));
  printf("sizeof(long)=%d bytes\n", sizeof(long));
  printf("sizeof(float)=%d bytes\n",sizeof(float));
  printf("sizeof(double)=%d bytes\n",sizeof(double));
  printf("sizeof(long double) = %d bytes\n",
                                sizeof (long double));
  printf("\n");
}
```

Κωδικοποίηση ακεραίων



unsigned: $1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 =$

$128 + 0 + 32 + 0 + 8 + 0 + 2 + 0 = 170$

signed: $-1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 =$

$-128 + 0 + 32 + 0 + 8 + 0 + 2 + 0 = -86$

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value
00	00000000	0
01	00000001	1
02	00000010	2
03	00000011	3
04	00000100	4
05	00000101	5
06	00000110	6
07	00000111	7

hex	binary	value
08	00001000	8
09	00001001	9
0A	00001010	10
0B	00001011	11
0C	00001100	12
0D	00001101	13
0E	00001110	14
0F	00001111	15

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value
10	00010000	16
11	00010001	17
12	00010010	18
13	00010011	19
14	00010100	20
15	00010101	21
16	00010110	22
17	00010111	23

hex	binary	value
18	00011000	24
19	00011001	25
1A	00011010	26
1B	00011011	27
1C	00011100	28
1D	00011101	29
1E	00011110	30
1F	00011111	31

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value
20	00100000	32
21	00100001	33
22	00100010	34
…	……	…

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value
70	01110000	112
71	01110001	113
72	01110010	114
73	01110011	115
74	01110100	116
75	01110101	117
76	01110110	118
77	01110111	119

hex	binary	value
78	01111000	120
79	01111001	121
7A	01111010	122
7B	01111011	123
7C	01111100	124
7D	01111101	125
7E	01111110	126
7F	01111111	127

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value*
80	10000000	-128/128
81	10000001	-127/129
82	10000010	-126/130
83	10000011	-125/131
84	10000100	-124/132
85	10000101	-123/133
86	10000110	-122/134
87	10000111	-121/135

hex	binary	value*
88	10001000	-120/136
89	10001001	-119/137
8A	10001010	-118/138
8B	10001011	-117/139
8C	10001100	-116/140
8D	10001101	-115/141
8E	10001110	-114/142
8F	10001111	-113/141

(*) εξαρτάται από τον τύπο/κωδικοποίηση `signed`/`unsigned`

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value*
90	10010000	-112/144
91	10010001	-111/145
92	10010010	-110/146
…	……	…/…

(*) εξαρτάται από τον τύπο/κωδικοποίηση `signed`/`unsigned`

Δεκαεξαδική κωδικοποίηση ακεραίων (8 bits)

hex	binary	value*
F0	11110000	-16/240
F1	11110001	-15/241
F2	11110010	-14/242
F3	11110011	-13/243
F4	11110100	-12/244
F5	11110101	-11/245
F6	11110110	-10/246
F7	11110111	- 9/247

hex	binary	value*
F8	11111000	-8/248
F9	11111001	-7/249
FA	11111010	-6/250
FB	11111011	-5/251
FC	11111100	-4/252
FD	11111101	-3/253
FE	11111110	-2/254
FF	11111111	-1/255

(*) εξαρτάται από τον τύπο/κωδικοποίηση `signed`/`unsigned`

Ερμηνεία δυαδικών δεδομένων

- Το υλικό του Η/Υ **δεν** γνωρίζει την **σημασία** των δεδομένων που αποθηκεύονται στην μνήμη.
- Τα bits ερμηνεύονται με βάση την κωδικοποίηση που αντιστοιχεί στον **τύπο** που έχει η μεταβλητή μέσω της οποίας προσπελαύνεται η μνήμη.
- Π.χ.:

  `01100001`: `'a'` (`char`) ή `193` (`short`)

  `11111111`: `-1` (`short`) ή `255` (`unsigned short`)
- Η ερμηνεία των περιεχομένων της μνήμης πρέπει να γίνεται με πλήρη επίγνωση και ιδιαίτερη προσοχή, έτσι ώστε να μην γίνονται **σημασιολογικά** λάθη.

Δηλώσεις μεταβλητών

- Οι δηλώσεις μεταβλητών δίνονται πριν από (σχεδόν) όλες τις υπόλοιπες εντολές ενός προγράμματος.
- Η μορφή των εκφράσεων δήλωσης είναι:

```
<τύπος> <όνομα> ;
<τύπος> <όνομα>,..., <όνομα> ;
<τύπος> <όνομα> = <τιμή> ;
```

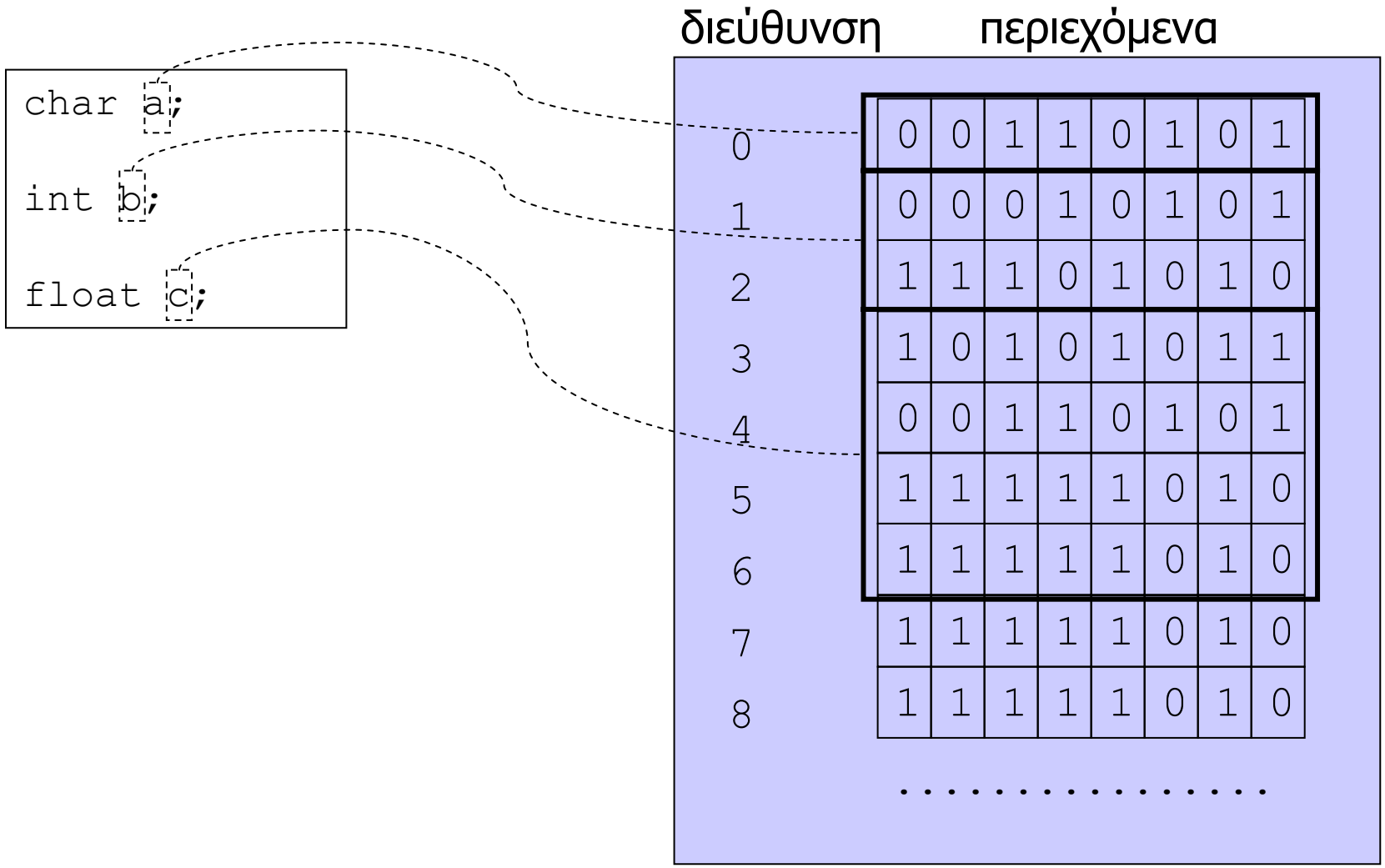
- Κάθε μεταβλητή (και συνάρτηση) πρέπει να έχει διαφορετικό όνομα.
- Κατά την δήλωση της, μια μεταβλητή μπορεί προαιρετικά να λάβει και μια αρχική τιμή.
- Το πρόθεμα `const` σε συνδυασμό με την ανάθεση αρχικής τιμής υποδηλώνει ότι η τιμή της μεταβλητής δεν θα αλλάξει κατά την διάρκεια της εκτέλεσης.

```
char c;          /* μεταβλητή χαρακτήρα με όνομα c */

short si=1;   /* μεταβλητή μικρού ακεραίου με
                 όνομα si και αρχική τιμή 1 */

const float pi=3.1415;     /* μεταβλητή πραγματικού
                              αριθμού απλής ακριβείας
                              με όνομα pi και σταθερή
                              τιμή 3.1415          */

int c;           /* μεταβλητή ακεραίου με όνομα c */
```

ο μεταφραστής θα διαμαρτυρηθεί, καθώς το όνομα c χρησιμοποιείται

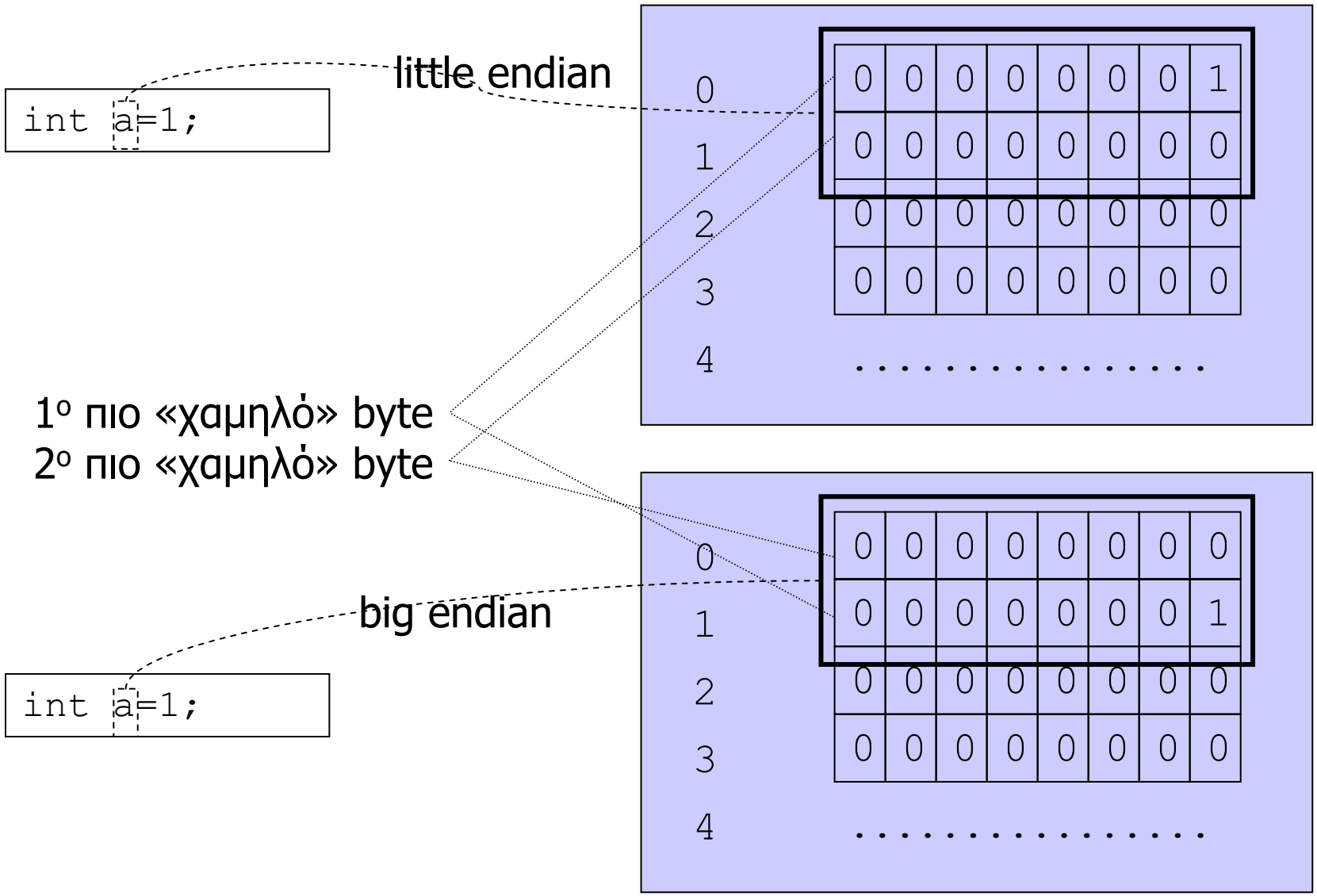
Μεταβλητές και μνήμη

- Κατά την εκτέλεση του προγράμματος, η δήλωση μιας μεταβλητής οδηγεί, στην δέσμευση αντίστοιχου χώρου μνήμης για την αποθήκευση των τιμών της.
- Η δέσμευση μνήμης γίνεται όταν «ενεργοποιείται» η δήλωση της μεταβλητής –περισσότερα σε λίγο ...
- Οι μεταβλητές ενός προγράμματος καταλαμβάνουν (συνήθως) **συνεχόμενες** περιοχές μνήμης: εκεί που τελειώνει η περιοχή της μεταβλητής που δηλώθηκε πρώτη, αρχίζει η περιοχή της μεταβλητής που δηλώθηκε δέυτερη, κλπ.
- Σημείωση: αυτό δεν ισχύει πάντα, καθώς εξαρτάται από την υλοποίηση του μεταφραστή ή/και του περιβάλλοντος εκτέλεσης της γλώσσας.


διεύθυνση
περιεχόμενα
char a;
int b;
float c;
0 0 1 1 0 1 0 1
0 0 0 1 0 1 0 1
1 1 1 0 1 0 1 0
1 0 1 0 1 0 1 1
0 0 1 1 0 1 0 1
1 1 1 1 1 0 1 0
1 1 1 1 1 0 1 0
1 1 1 1 1 0 1 0
1 1 1 1 1 0 1 0
0
1
2
3
4
5
6
7
8
. . . . . .

Αποθήκευση ακεραίων στη μνήμη

- Οι τύποι `short int`, `int` και `long int` έχουν (συνήθως) μέγεθος αποθήκευσης πάνω από 1 byte.
- Με ποιά σειρά αποθηκεύονται τα bytes στην μνήμη;
- **Σύστημα big endian**: τα πιο σημαντικά bytes αποθηκεύονται στις μικρότερες θέσεις μνήμης.
- **Σύστημα little endian**: τα λιγότερο σημαντικά bytes αποθηκεύονται στις μικρότερες θέσεις μνήμης.
- Ο προγραμματιστής δεν καταλαβαίνει τη διαφορά όσο χρησιμοποιεί κανονικές πράξεις πρόσβασης.
- Η διαφορά γίνεται αισθητή όταν κανείς χρησιμοποιεί γλώσσα μηχανής ή/και προσπελαύνει τα δεδομένα που βρίσκονται στη μνήμη με άμεσο τρόπο –βλέπε επόμενα μαθήματα ...


little endian
int a=1;
0
1
2
3
4
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
. . . . . .
1ο πιο «χαμηλό» byte
2ο πιο «χαμηλό» byte
big endian
int a=1;
0
1
2
3
4
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
. . . . . .

Τελεστής ανάθεσης

- Το σύμβολο της ανάθεσης είναι το = (**προσοχή:** το σύμβολο ισότητας είναι το ==).
- Η μορφή των προτάσεων ανάθεσης είναι:

  `<όνομα> = <έκφραση>`

1. Αποτιμάται η έκφραση στο δεξί μέρος.
2. Η τιμή που παράγεται αποθηκεύεται στην μεταβλητή το όνομα της οποίας δίνεται στο αριστερό μέρος.

- Σημείωση: η έκφραση μπορεί να συμπεριλαμβάνει μεταβλητές, στην οποία περίπτωση επιστρέφεται η τιμή που έχει αποθηκευτεί στην μνήμη τους.
- Σημείωση 2: στο δεξί μέρος μπορεί να εμφανίζεται η ίδια μεταβλητή που εμφανίζεται και το αριστερό.

Τελεστής ανάθεσης (2)

- Η ανάθεση αποτελεί μια **έκφραση αποτίμησης** που **επιστρέφει την τιμή που ανατίθεται**!
- Μια έκφραση ανάθεσης μπορεί να χρησιμοποιηθεί ως τμήμα άλλων, πιο πολύπλοκων εκφράσεων.
- Π.χ. επιτρέπονται εκφράσεις «αλυσιδωτής» ανάθεσης τιμών (από δεξιά προς τα αριστερά), όπου όλες οι μεταβλητές παίρνουν την τιμή που εμφανίζεται στο δεξί μέρος.

```
int i,j,k;

i = 1;                  /* i γίνεται 1 */

j = 1+1;                /* j γίνεται 2 */

k = i+j;                /* k γίνεται 3 */

i = k = j;              /* i,k γίνονται 2 */

j = (i=3) + k;          /* i γίνεται 3, j γίνεται 5 */

i = i+1;                /* i γίνεται 4 */
```

Τελεστής ανάθεσης (3)

- Πολλές φορές χρησιμοποιούμε την ανάθεση για να αλλάξουμε την τιμή μιας μεταβλητής **σε σχέση** με την **παλιά** τιμή της (ίδιας μεταβλητής).
- Για το πετύχουμε αυτό, γράφουμε:

  `<όνομα> = <όνομα><op>(<έκφραση>)`

  όπου `<op>` ένας δυαδικός τελεστής.
- Το ίδιο αποτέλεσμα μπορεί να επιτευχθεί με χρήση της «σχετικής» ανάθεσης `<op>=`, ως εξής:

  `<όνομα> <op> = <έκφραση>`
- Προσοχή στις προτεραιότητες, καθώς η έκφραση που εμφανίζεται στα δεξιά αποτιμάται **πριν** εφαρμοστεί ο τελεστής `<op>`

```
int i,j,k;

i = 1;

j = 2;

i += 2;                 /* i γίνεται 3 */

j *= i+1;               /* j γίνεται 8 */

k = (i+=j) + 1;         /* i γίνεται 11, k γίνεται 12 */
```

Τελεστές αυξομείωσης

- Μια ειδική περίπτωση ανάθεσης είναι η αύξηση ή μείωση της τιμής μιας μεταβλητής κατά 1:

  `<όνομα> = <όνομα><op>1`

  `<όνομα><op> = 1`

  όπου `<op>` ο τελεστής `+` ή `-`
- Παρόμοιο αποτέλεσμα μπορεί να επιτευχθεί με χρήση των τελεστών `++` ή `--` :

  `<όνομα> ++` ή `<όνομα> --`

  `++ <όνομα>` ή `-- <όνομα>`
- Όταν ο τελεστής εμφανίζεται **πριν** / **μετά** το όνομα της μεταβλητής, τότε ως αποτέλεσμα της έκφρασης επιστρέφεται η **νέα** / **παλιά** τιμή της μεταβλητής.

```
int i,j,k;

i = 0;

i++;                    /* i γίνεται 1 */

j = i++;                /* j γίνεται 1, i γίνεται 2 */

k = --j;                /* k γίνεται 0, j γίνεται 0 */

i = (k++) + 1;          /* i γίνεται 1, k γίνεται 1 */

i = (++k) + 1;          /* i γίνεται 3, k γίνεται 2 */
```

Σχόλιο

- Μπορεί να κατασκευαστούν συντακτικά ορθές εκφράσεις που δίνουν μη προφανή αποτελέσματα.
- Η σημασιολογία κάποιων από αυτές τις εκφράσεις μπορεί να μην ορίζεται (ξεκάθαρα) από τη γλώσσα.
- Καλό είναι να μην χρησιμοποιούνται εκφράσεις όταν δεν **γνωρίζουμε** την **ακριβή** σημασιολογία τους (π.χ. δεν έχουμε διαβάσει το εγχειρίδιο γλώσσας).
- Σημείωση: είναι **προγραμματιστικό λάθος** να χρησιμοποιούνται εκφράσεις των οποίων η σημασιολογία **δεν ορίζεται** από την γλώσσα (το αποτέλεσμα εξαρτάται από το μεταφραστή, με άλλα λόγια προγραμματίζουμε «στην τύχη»).

```
int i;

i = 0;
i = (i++);                    /* i γίνεται 0 */

i = 0;
i = (++i);                    /* i γίνεται 1 */

i = 0;
i = (i=i+1);                  /* i γίνεται 1 */

i = 0;
i = (i++) + (i++);            /* i γίνεται 2 */

i = 0;
i = (++i) + (++i);            /* i γίνεται 4 */

i = 0;
i = (i=i+1) + (i=i+1);        /* i γίνεται 3 */
```

Αριθμητικοί τελεστές για `int`/`float`/`double`

- '`+`' πρόσθεση δύο τιμών
- '`-`' αφαίρεση δύο τιμών
- '`*`' πολλαπλασιασμός δύο τιμών
- '`/`' διαίρεση δύο τιμών
- '`%`' υπόλοιπο διαίρεσης δύο τιμών
- Τα '`/`' και '`%`' έχουν μεγαλύτερη προτεραιότητα αποτίμησης σε σχέση με τα '`+`' και '`-`'.
- Η αποτίμηση γίνεται από αριστερά προς τα δεξιά.
- **Σημείωση**: ισχύει ότι `x = x/y + x%y`, όμως δεν ισχύει πάντα η «μαθηματική» ιδιότητα του υπολοίπου (>=0), π.χ. όταν ο αριθμητής είναι αρνητικός.
- Π.χ. `-5%4` ή `-5%-4`.

Τελεστές σε επίπεδο bits

- `&` δυαδικό «and»
- `|` δυαδικό «or»
- `^` δυαδικό «xor»
- `~` δυαδικό «not»
- `>>` δεξιά ολίσθηση (msb -> lsb)
- `<<` αριστερή ολίσθηση (lsb -> msb)
- **Προσοχή:** στην αριστερή ολίσθηση τα «λιγότερο σημαντικά» bits παίρνουν **πάντα** την τιμή `0`, ενώ στην δεξιά ολίσθηση τα «περισσότερο σημαντικά» bits παίρνουν την τιμή του περισσότερο σημαντικού bit (arithmetic shift) ή `0` (logical shift), **ανάλογα** με αν το όρισμα ερμηνεύεται ως `signed` ή `unsigned`.

```
char a,b,c;
unsigned char d;

a = 'a';              /* a γίνεται   01100001 */

b = 'b';              /* b γίνεται   01100010 */

c = a|b;              /* c γίνεται   01100011 */

c = a&b;              /* c γίνεται   01100000 */

c = a^b;              /* c γίνεται   00000011 */

d = c = ~a;           /* d,c γίνεται 10011110 */

d = d>>3;             /* d γίνεται   000 10011 */

c = c>>3;             /* c γίνεται   111 10011 */
```

Γρήγορος πολλαπλασιασμός / διαίρεση

- Με τον τελεστή ολίσθησης bits μπορούμε να υλοποιήσουμε γρήγορες πράξεις πολλαπλασιασμού και διαίρεσης με τιμές που είναι δυνάμεις του 2:

```
v = v<<i; /* v = v*2^i */
v = v>>i; /* v = v/2^i */
```

- Και φυσικά μπορούμε να υπολογίσουμε εύκολα τις δυνάμεις του 2:

```
v = 1<<i; /* v = 2^i */
```

```
short int i;

i = 5;          /* i γίνεται   00000000 00000101 */

i = i<<4;       /* i γίνεται   00000000 01010000 */

i = i>>2;       /* i γίνεται   00000000 00010100 */

i = 1<<3;       /* i γίνεται   00000000 00001000 */

i = 1<<8;       /* i γίνεται   00000001 00000000 */
```

Σχεσιακοί και λογικοί τελεστές

- `==, !=` ισότητα, ανισότητα
- `>, >=` μεγαλύτερο, μεγαλύτερο ίσο
- `<, <=` μικρότερο, μικρότερο ίσο
- `!` λογική άρνηση
- Οι σχεσιακοί και λογικοί τελεστές χρησιμοποιούνται για την κατασκευή λογικών εκφράσεων (συνθηκών).
- **Δεν υπάρχει λογικός τύπος** (boolean).
- Το αποτέλεσμα μιας λογικής έκφρασης είναι `0` ή `1` (για ψευδές ή αληθές), και μπορεί να χρησιμοποιηθεί **και ως ακέραιος** – κλασική πηγή λαθών στην C.
- Επίσης, η τιμή `0` ερμηνεύεται ως «ψευδές» (false) και οποιαδήποτε τιμή διάφορη του `0` ως «αληθές» (true).

```
int a=1, b=2, c;

c = (a==b);                    /* c γίνεται 0 */

c = (a!=b);                    /* c γίνεται 1 */

c = (a<=b);                    /* c γίνεται 1 */

c = ((c+a)!=b);                /* c γίνεται 0 */

c = (!a==!b);                  /* c γίνεται 1 */

c = (a!=b) + !(a==b);          /* c γίνεται 2 */

c = !( (a!=b) + !(a==b) ); /* c γίνεται 0 */
```

Λογικοί σύνδεσμοι

- `||` λογικό «ή»
- `&&` λογικό «και»
- Οι λογικοί σύνδεσμοι χρησιμοποιούνται για την κατασκευή σύνθετων λογικών εκφράσεων.
- Η αποτίμηση των λογικών εκφράσεων γίνεται από «τα αριστερά προς τα δεξιά» και **μόνο** όσο χρειάζεται για να διαπιστωθεί το τελικό αποτέλεσμα της έκφρασης (conditional evaluation).
- Για το `||` η αποτίμηση σταματά μόλις η ενδιάμεση τιμή γίνει διάφορη του `0`, ενώ για το `&&` η αποτίμηση σταματά μόλις η ενδιάμεση τιμή γίνει `0`.
- Σημείωση: προσοχή στις παρενέργειες!

```
int a=1, b=0, c;

c = a&&b;                       /* c γίνεται 0 */

c = (a==1)||(b==1);             /* c γίνεται 1 */

c = (a++>1);                    /* c γίνεται 0,
                                   a γίνεται 2 */

c = b && (a++);                 /* c γίνεται 0,
                                   a παραμένει 2 */

c = (b++) && (a++);             /* c γίνεται 0,
                                   b γίνεται 1,
                                   a παραμένει 2 */

c = (--b) || (a++);             /* c γίνεται 1,
                                   b γίνεται 0,
                                   a γίνεται 3 */
```

Παρενέργειες

- Ο όρος **παρανέργεια** χρησιμοποιείται για να αναφερθούμε σε αλλαγή τιμής μιας μεταβλητής χωρίς αυτό να είναι εύκολα ορατό από τον κώδικα.
- Η τεχνική αποτίμησης λογικών εκφράσεων μπορεί να οδηγήσει σε κώδικα με παρενέργειες, π.χ.

  ```
  <lexpr> && (a++)
  ```

  οδηγεί σε αλλαγή της τιμής της μεταβλητής `a` **μόνο** όταν η `<lexpr>` αποτιμάται ως αληθής, πράγμα που (στη γενικότερη περίπτωση) δεν είναι εύκολο να εκτιμηθεί διαβάζοντας τον (υπόλοιπο) κώδικα.
- Ο προγραμματιστμός με παρενέργειες θεωρείται **κακό** στύλ, καθώς οδηγεί σε κώδικα που είναι αρκετά δύσκολο να κατανοηθεί.

Υπόλοιποι τελεστές

- Ο τελεστής `<lexpr>?<expr1>:<expr2>` αποτιμά την έκφραση `lexpr` και εφόσον η τιμή της είναι διάφορη του `0` αποτιμά και επιστρέφει το αποτέλεσμα της έκφρασης `expr1`, διαφορετικά αποτιμά και επιστρέφει το αποτέλεσμα της έκφρασης `expr2`.
- Ο τελεστής `expr1,expr2,…,exprn` αποτιμά τις εκφράσεις `expr1`, `expr2` μέχρι και `exprn`, «από αριστερά προς τα δεξιά», και επιστρέφει το αποτέλεσμα της τελευταίας.
- Π.χ.:

```
int a=1,b=2,c,d;
c = (a<b)?a:b;              /* c == 1 */
d = (c=a,a=b,b=c,c=0); /* a,b,c,d == 2,1,0,0 */
```

Προτεραιότητες τελεστών

μέγιστη

```
() [] . ->
! ~ ++ -- + - * & (cast)
* / %
+ -
<< >>
> >= < <=
==, !=
&
^
|
&&
||
? :
= += -= *= /= %= &= |= <<= >>=
,
```

(ατομικοί)

ελάχιστη

Κυριολεκτικά

- Κάποιες μεταβλητές του προγράμματος πρέπει συνήθως να αρχικοποιηθούν με συγκεκριμένη τιμή.
- Υπάρχει επίσης η περίπτωση να επιθυμούμε να χρησιμοποιήσουμε μια ειδική συγκεκριμένη τιμή σε εκφράσεις αποτίμησης, π.χ. γνωστές σταθερές.
- Μια έκφραση που ορίζει μια τιμή χωρίς αναφορά σε κάποια μεταβλητή ονομάζεται «κυριολεκτικό».
- Για κάθε τύπο δεδομένων ορίζονται διαφορετικές μορφές προσδιορισμού κυριολεκτικών.
- Η τιμή των κυριολεκτικών, καθώς και εκφράσεων που χρησιμοποιούν αποκλειστικά και μόνο κυριολεκτικά, είναι ήδη γνωστή πριν την εκτέλεση του κώδικα.

Κυριολεκτικά `char`

- Η έκφραση της μορφής `'<c>'` συμβολίζει τον αντίστοιχο εκτυπώσιμο χαρακτήρα ASCII `<c>`.
- Η έκφραση `'\<c>'` συμβολίζει τον αντίστοιχο ειδικό **μη εκτυπώσιμο** χαρακτήρα ASCII (βλέπε manual).
- Η έκφραση `'\x<`d_1d_2`>'` συμβολίζει τον χαρακτήρα ASCII με τον αντίστοιχο δεκαεξαδικό κωδικό d_1d_2.
- Η έκφραση `'\<`$d_1d_2d_3$`>'` συμβολίζει τον χαρακτήρα ASCII με τον αντίστοιχο οκταδικό κωδικό $d_1d_2d_3$.
- Π.χ.:

`'a','\x61','\141'`	χαρακτήρας `a`
`'\n','\x0A','\012'`	newline / linefeed

Δεκαεξαδική κωδικοποίηση ASCII

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- Σημείωση: οι αλφαβητικοί χαρακτήρες έχουν **διαδοχικές** τιμές, αν ερμηνευτούν ως ακέραιοι.
- Σημείωση 2: οι χαρακτήρες των δεκαδικών ψηφίων έχουν **διαδοχικές** τιμές, αν ερμηνευτούν ως ακέραιοι.

Κυριολεκτικά `int`

- Αν η έκφραση αρχίζει με δεκαδικό ψηφίο διαφορετικό του `0`, τότε ερμηνεύεται με το δεκαδικό σύστημα.
- Αν έκφραση αρχίζει με `0` τότε ερμηνεύεται σύμφωνα με το οκταδικό σύστημα, και αν αρχίζει με `0x` τότε ερμηνεύεται σύμφωνα με το δεκαεξαδικό σύστημα.
- Εκφράσεις με κατάληξη `l` ή `L` ερμηνεύονται ως `long int`, με κατάληξη `u` ή `U` ως `unsigned int`, με κατάληξη `ul` ή `UL` ως `unsigned long int`, διαφορετικά ως `int`.
- Π.χ.:

  `0xFF` τιμή `255` (`-1` ως `char`)

  `32768u` τιμή `32768` (`-32768` ως `short`)

  `0123` τιμή `83`

Κυριολεκτικά `float` και `double`

- Η έκφραση πρέπει να δίνεται σε συμβατική μορφή, δηλαδή σε δεκαδικό σύστημα.
- Προαιρετικά, μπορεί να δίνεται με δεκαδικά ψηφία ή/και με δεκαδικό εκθέτη που μπορεί να λαμβάνει και αρνητικό πρόσημο.
- Εκφράσεις με κατάληξη `f` ή `F` ερμηνεύονται ως `float`, ενώ με κατάληξη `l` ή `L` ως `long double`, διαφορετικά ως `double`.

Μετατροπές τιμών διαφορετικών τύπων

- Γίνονται **αυτόματα** σε αποτιμήσεις και αναθέσεις:
  - `char` -> `int`, `short` -> `int`, `float` -> `double`
  - αν ένα από τα ορίσματα μιας πράξης είναι `long double`, `double`, `float`, `long`, `unsigned`, `int` τότε και το άλλο όρισμα «προάγεται» αντίστοιχα
- Προσοχή στην μετατροπή `char` -> `int` καθώς γίνεται sign extension αν το `char` είναι `signed`.
- Όταν μια «μεγάλη» τιμή ανατίθεται σε «μικρότερη» μεταβλητή τότε **χάνεται** μέρος των δεδομένων.
- Ο προγραμματιστής μπορεί να **εκβιάσει** μια «παράνομη» μετατροπή με το λεγόμενο type casting.

```
char c1='\xff'; unsigned char c2='\xff';
short i1; unsigned short i2;

i1 = i2 = c1; /* i1,i2 γίνεται -1,65535(=2^16-1) */
i1 = i2 = c2; /* i1,i2 γίνεται 255,255 */
```

```
double d1=10.0,d2; int i1=3,i2=10;

d2 = d1/i1;            /* d2 γίνεται 3.33… */
d2 = (double)i2/i1;    /* d2 είναι 3.33… */
d2 = (double)(i2/i1); /* d2 είναι 3.0 */
```

```
int i; char c;
i = 256;              /* i γίνεται 256 */
c = i;                /* c γίνεται 0 */
i = c;                /* i γίνεται 0 */
```

Αριθμητική με χαρακτήρες

- Ένα (από τα πολλά) «φαινόμενα» της αυτόματης μετατροπής τύπων είναι η αριθμητική με χαρακτήρες.
- Μπορούμε να συνδυάσουμε ένα χαρακτήρα με ένα ακέραιο ή ένα χαρακτήρα με ένα χαρακτήρα:

```
'a' + 1              /* 98,0x62,'b' */
'b' - 'a'            /* 1 */
'5' - '3' + '0'      /* 50,0x32,'2' */
```

- Μια συνάρτηση που δέχεται παράμετρο ένα ακέραιο, μπορεί να δεχτεί σαν παράμετρο έναν χαρακτήρα

```
putchar(97);      /* 'a' */
putchar('a'+2);   /* 'c' */
```

```
/* ανάγνωση δύο δεκαδικών ψηφίων, υπολογισμός της
   διαφοράς τους και εκτύπωση της ως ακέραιος */

#include <stdio.h>

int main(int argc, char *argv[]) {

  char c1,c2;
  int i;

  printf("enter two chars: ");
  c1 = getchar(); c2 = getchar();

  i = c1-c2;

  printf("%d\n",i);

}
```

```
/* ανάγνωση δύο δεκαδικών ψηφίων, υπολογισμός της
   διαφοράς / αθροίσματος τους και εκτύπωση του
   αποτελέσματος ως δεκαδικό ψηφίο */

#include <stdio.h>

int main(int argc, char *argv[]) {

  char c1,c2,c3;

  printf("enter two chars (0-9): ");
  c1 = getchar(); c2 = getchar();

  c3 = c1-c2+'0';
  putchar(c3); putchar('\n');

  c3 = c1+c2-'0';
  putchar(c3); putchar('\n');

}
```

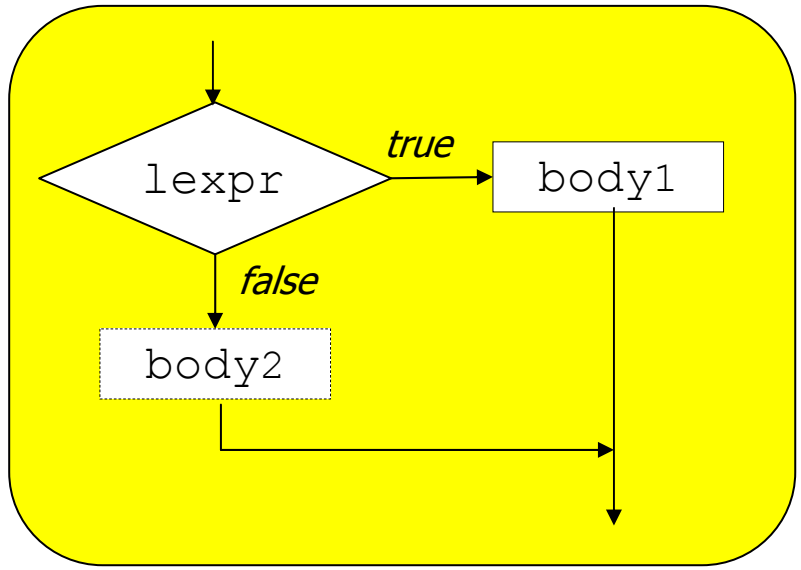
Δομές Ελέγχου

Δομές ελέγχου

- Με τις εντολές εισόδου, εξόδου και επεξεργασίας των τιμών των μεταβλητών μπορεί να γραφτούν κάποια απλά προγράμματα.
- Οι δυνατότητες είναι περιορισμένες, καθώς το πρόγραμμα ακολουθεί **μια μοναδική** και **εκ των προτέρων** προδιαγεγραμμένη εκτέλεση.
- Με τις δομές ελέγχου, ο προγραμματιστής μπορεί να αφήσει το **ίδιο**_το πρόγραμμα να πάρει αποφάσεις σχετικά με την εκτέλεση (ή μη) κάποιων εντολών.
- Αυτές οι αποφάσεις λαμβάνονται **την ώρα της εκτέλεσης**, με βάση τις (τρέχουσες) τιμές των μεταβλητών του προγράμματος.
- Δομές ελέγχου: εκτέλεση υπό συνθήκη & επανάληψη.

Εκτέλεση υπό συνθήκη: `if-else`

```
if <lexpr>
   <body1>
else
   <body2>
```



- Αν η συνθήκη `lexpr` αποτιμηθεί σε τιμή διάφορη του `0`, τότε η εκτέλεση συνεχίζεται με το `body1`, διαφορετικά με το εναλλακτικό `body2`.
- Το εναλλακτικό σκέλος `else` είναι προαιρετικό: αν δεν υπάρχει και η συνθήκη `lexpr` αποτιμηθεί σε `0` τότε απλά παρακάμπεται το `body1`.

```
/* b λαμβάνει την απόλυτη τιμή του a */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int a,b;

  scanf("%d", &a);

  if (a >= 0) {  b = a; }
  else /* a < 0 */ { b = -a; }

  printf("%d\n", b);

}
```

```
/* d λαμβάνει την μέγιστη τιμή των a,b,c */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int a,b,c,d;

  scanf("%d %d %d", &a, &b, &c);

  if (a > b) {
    if (a > c) { d = a; }
    else /* a <= c */ { d = c; }
  }
  else /* a <= b */ {
    if (b > c) { d = b; }
    else /* b <= c */ { d = c; }
  }

  printf("%d\n", d);

}
```

```
/* d λαμβάνει την μέγιστη τιμή των a,b,c */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int a,b,c,d;

  scanf("%d %d %d", &a, &b, &c);

  d = a;
  if (d < b) { d = b; }
  if (d < c) { d = c; }

  printf("%d\n", d);

}
```

Σώμα εντολών σε δομές ελέγχου

- Το σώμα μιας εντολής ελέγχου (π.χ. του `if-else`) δίνεται **υποχρεωτικά** ανάμεσα σε `{}` όταν αυτό αποτελείται από πολλές εντολές.
- Οι `{}` είναι προαιρετικές **μόνο** όταν το σώμα αποτελείται από **μια** εντολή (σημείωση: μια δομή ελέγχου θεωρείται συντακτικά ως μια εντολή).
- Η χρήση `{}` και η μορφοποίηση του κειμένου του προγράμματος με κενούς χαρακτήρες και κατανομή σε ξεχωριστές γραμμές είναι θέμα «γούστου», αλλά επηρεάζει **σαφώς** την αναγνωσιμότητα.
- Ένας άπειρος προγραμματιστής καλό είναι να χρησιμοποιεί **πάντα** `{}`, ακόμα και για σώματα εντολών που περιέχουν μια μοναδική εντολή.

```
/* εκτύπωση ονόματος ημέρας με τον κωδικό day */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int day;

  scanf("%d", &day);

  if      (day == 1) { printf("Mon\n"); }
  else if (day == 2) { printf("Tue\n"); }
  else if (day == 3) { printf("Wed\n"); }
  else if (day == 4) { printf("Thu\n"); }
  else if (day == 5) { printf("Fri\n"); }
  else if (day == 6) { printf("Sat\n"); }
  else if (day == 7) { printf("Sun\n"); }
  else { printf("wrong code %d\n",day); }

}
```

```
if      (day == 1) { printf("Mon\n"); }
else if (day == 2) { printf("Tue\n"); }
else if (day == 3) { printf("Wed\n"); }
else if (day == 4) { printf("Thu\n"); }
else if (day == 5) { printf("Fri\n"); }
else if (day == 6) { printf("Sat\n"); }
else if (day == 7) { printf("Sun\n"); }
else { printf("wrong code %d\n",day); }
```

```
if      (day == 1) printf("Mon\n");
else if (day == 2) printf("Tue\n");
else if (day == 3) printf("Wed\n");
else if (day == 4) printf("Thu\n");
else if (day == 5) printf("Fri\n");
else if (day == 6) printf("Sat\n");
else if (day == 7) printf("Sun\n");
else printf("wrong code %d\n",day);
```

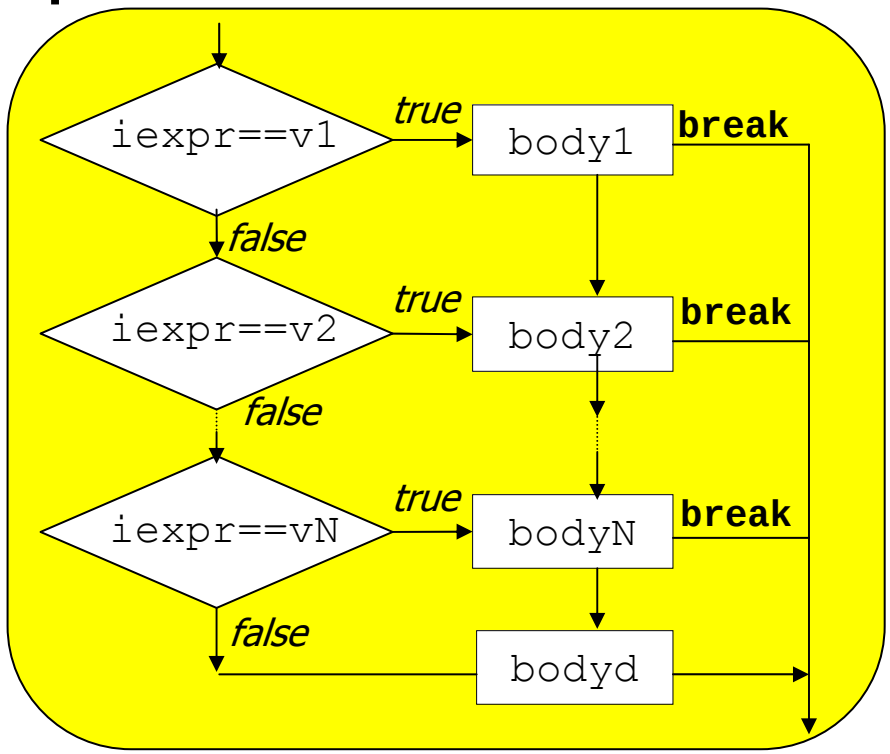
```
if (day == 1)
  printf("Mon\n");
else if (day == 2)
  printf("Tue\n");
else if (day == 3)
  printf("Wed\n");
else if (day == 4)
  printf("Thu\n");
else if (day == 5)
  printf("Fri\n");
else if (day == 6)
  printf("Sat\n");
else if (day == 7)
  printf("Sun\n");
else
  printf("wrong code %d\n",day);
```

```
if (day == 1) printf("Mon\n");
else
  if (day == 2) printf("Tue\n");
  else
    if (day == 3) printf("Wed\n");
    else
      if (day == 4) printf("Thu\n");
      else
        if (day == 5) printf("Fri\n");
        else
          if (day == 6) printf("Sat\n");
          else
            if (day == 7) printf("Sun\n");
            else
              printf("wrong code %d\n",day);
```

```
if (day == 1) {
  printf("Mon\n");
}
else {
  if (day == 2) {
    printf("Tue\n");
  }
  else {
    if (day == 3) {
      printf("Wed\n");
    }
    else {
      if (day == 4) {
        printf("Thu\n");
      }
      else {
        if (day == 5) {
          printf("Fri\n");
        }
        else {
          if (day == 6) {
            printf("Sat\n");
          }
          else {
            if (day == 7) {
              printf("Sun\n");
            }
            else {
              printf("wrong code %d\n",day);
            }
          }
        }
      }
    }
  }
}
```

Εκτέλεση υπό συνθήκη: `switch-case`

```
switch <iexpr> {
  case <v1>: <body1>
  case <v2>: <body2>
  …
  case <vN>: <bodyN>
  default:   <bodyd>
}
```



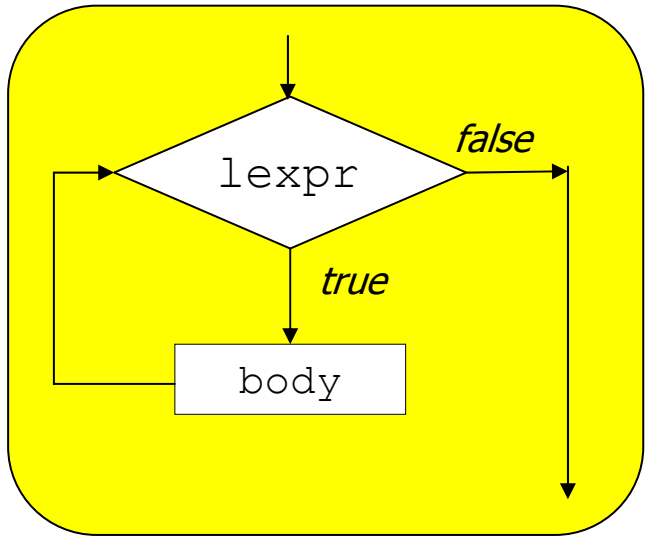
- Αποτιμάται (μια φορά) η αριθμητική έκφραση `iexpr` και επιλέγεται και εκτελείται το `bodyX` της πρώτης περίπτωσης που έχει την ίδια τιμή `valX`.
- Αν το σώμα `bodyX` δεν περιέχει την εντολή `break` στη συνέχεια εκτελείται το επόμενο σώμα `bodyX+1`.
- Το σκέλος `default` είναι προαιρετικό, και επιλέγεται αν δεν επιλεγεί άλλη περίπτωση.

```c
/* εκτύπωση ονόματος ημέρας με τον κωδικό day */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int day;

  scanf("%d", &day);

  switch (day) {
    case 1: { printf("Monday\n"); break; }
    case 2: { printf("Thuesday\n"); break; }
    case 3: { printf("Wednesday\n"); break; }
    case 4: { printf("Thursday\n"); break; }
    case 5: { printf("Friday\n"); break; }
    case 6: { printf("Saturday\n"); break; }
    case 7: { printf("Sunday\n"); break; }
    default:{ printf("wrong day code %d\n",day); }
  }

}
```

χωρίς την εντολή `break` το πρόγραμμα θα **συνέχιζε** την εκτέλεση με το **σώμα** της επόμενης περίπτωσης `case`

Επανάληψη εκτέλεσης: `while`

```
while <lexpr>
   <body>
```



- Όσο η συνθήκη επανάληψης `lexpr` αποτιμάται σε μια τιμή διάφορη του `0` τότε εκτελείται το `body`, διαφορετικά η εκτέλεση συνεχίζεται μετά το `while`.
- Το `body` μπορεί να μην εκτελεσθεί καθόλου αν την πρώτη φορά η `lexpr` αποτιμηθεί σε `0`, ή επ' άπειρο, αν η `lexpr` δεν αποτιμηθεί ποτέ σε `0` (κάτι που μπορεί να συμβεί λόγω προγραμματιστικού λάθους).

Παράδειγμα: υπολογισμός x!

- Θέλουμε το `y = 1*2*3*…*(x-1)*x;`
- **Πρόβλημα:** αν το `x` λαμβάνει τιμή από το χρήστη, τότε **δεν** γνωρίζουμε την τιμή του την ώρα που γράφουμε τον κώδικα μας, συνεπώς δεν γνωρίζουμε μέχρι ποιά τιμή να συνεχίσουμε τον πολλαπλασιασμό.
- Λύση: μετασχηματίζουμε την παραπάνω (στατική) έκφραση ως (δυναμική) επανάληψη των εντολών:

```
y = y * i; i = i + 1;
```

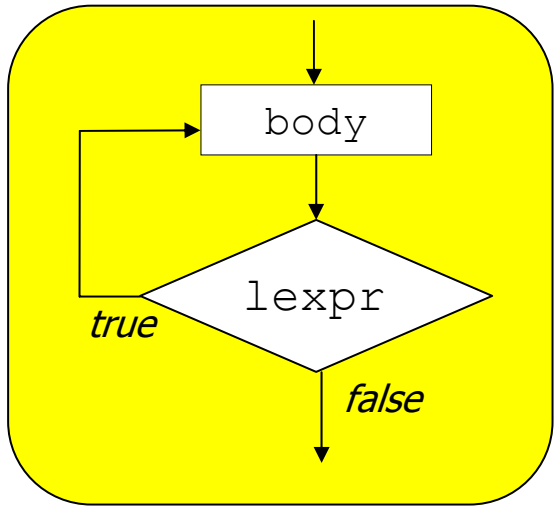
- Η επανάληψη πρέπει να γίνει τόσες φορές όσες η τιμή της μεταβλητής `x` δηλαδή μέχρι η μεταβλητή `i` να περάσει την τιμή της `x` : `i <= x`.
- Οι αρχικές τιμές είναι: `y = 1; i = 2;`

```c
/* y=x! */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int x,y,i;

  scanf("%d", &x);

  y = 1; i = 2;

  while (i <= x) {
    y = y*i;
    i = i+1;
  }

  printf("%d\n", y);

}
```

```
/* μέγιστος κοινός διαιρέτης x,y */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int x,x1,y,y1;

  scanf("%d %d", &x, &y);

  x1 = x; y1 = y;

  while (x1 != y1) {
    if (x1 > y1) { x1 = x1-y1; }
    else /* x1 <= y1 */ { y1 = y1-x1; }
  }

  printf("%d\n", x1);

}
```

Επανάληψη εκτέλεσης: `do-while`

```
do
   <body>
while <lexpr>
```

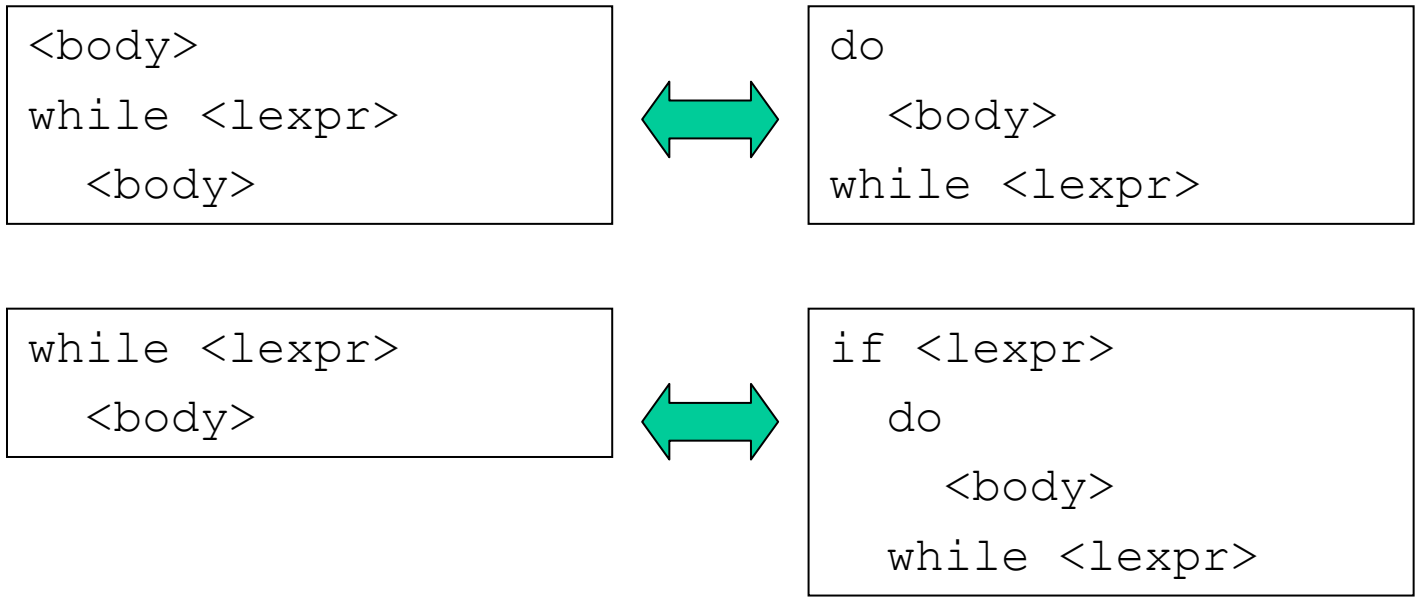


- Η συνθήκη επανάληψης `lexpr` αποτιμάται αφού εκτελεσθεί πρώτα το `body`, και αν η τιμή της είναι διάφορη του `0` τότε το `body` εκτελείται ξανά.
- Το `body` θα εκτελεσθεί τουλάχιστον μια φορά, και επ' άπειρο αν η `lexpr` δεν αποτιμηθεί ποτέ σε `0` (συνήθως από προγραμματιστικό λάθος).

```
/* αντιγραφή χαρακτήρων από είσοδο σε έξοδο
   μέχρι να διαβαστεί ο χαρακτήρας '~' */

#include <stdio.h>

int main(int argc, char *argv[]) {

  char c;

  do {
    c = getchar();
    putchar(c);
  } while (c != '~');

}
```

```
/* ανάγνωση αριθμητικής τιμής σύμφωνα με την
   έκφραση number={space}digit{space}blank */

#include <stdio.h>

int main(int argc, char *argv[]) {

  char c;
  int v;

  do {
    c = getchar();
  } while (c == ' ');

  v = 0;
  do {
    v = v + c-'0';
    c = getchar();
  } while (c != ' ');

  printf("%d\n", v);
}
```

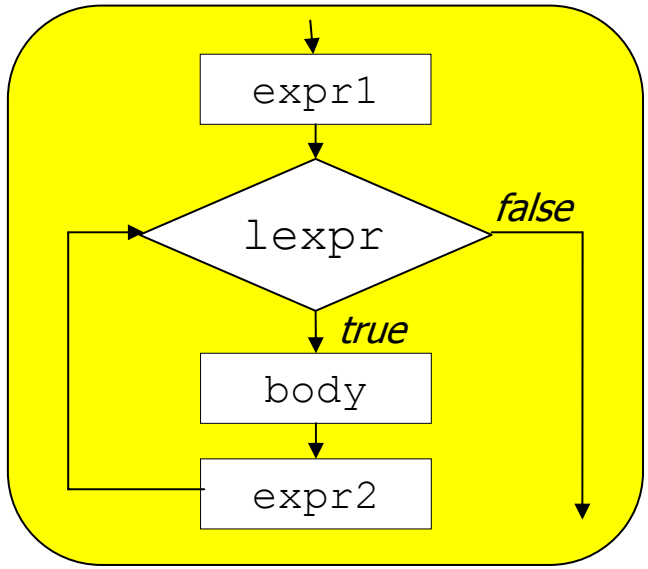
Ισοδυναμία `while` <-> `do-while`



- Για κάθε πρόγραμμα που χρησιμοποιεί `while` μπορεί πάντα να φτιαχτεί ένα ισοδύναμο πρόγραμμα που χρησιμοποιεί `do-while`, και το αντίστροφο.

Επανάληψη εκτέλεσης: `for`

```
for (<expr1>;<lexpr>;<expr2>)
  <body>
```



- Η έκφραση `expr1` αποτιμάται μια μοναδική φορά, και όσο η συνθήκη επανάληψης `lexpr` αποτιμάται σε τιμή διάφορη του `0` τότε εκτελείται το `body` και μετά η έκφραση `expr2`.
- Οι εκφράσεις `expr1` και `expr2` χρησιμοποιούνται συνήθως για την «αρχικοποίηση» και «πρόοδο» των μεταβλητών της συνθήκης επανάληψης `lexpr`.

```
/* s=1+2+…+n */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,s,i;

  scanf("%d", &n);

  s = 0;
  for (i=1; i<=n; i++) {
    s = s+i;
  }

  printf("%d\n", s);

}
```

```c
/* s=1+2+…+n */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,s,i;

  scanf("%d", &n);

  for (s=0,i=1; i<=n; i++) {
    s = s+i;
  }

  printf("%d\n", s);

}
```

```
/* s=1+2+…+n */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,s,i;

  scanf("%d", &n);

  for (s=0,i=1; i<=n; s=s+i,i++) { }

  printf("%d\n", s);

}
```

```
/* s=1+2+…+n */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,s,i;

  scanf("%d", &n);

  for (s=0,i=1; i<=n; s=s+i++) { }

  printf("%d\n", s);

}
```

```
/* πολλαπλασιασμός με πρόσθεση z=x*y, y>=0 */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int x,y,z,i;

  scanf("%d %d", &x, &y);

  z =0;
  for (i=0; i<y; i++) {
    z = z+x;
  }

  printf("%d\n", z);

}
```

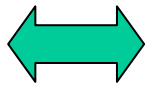
```
/* συνδυασμοί <i,j> με i:[0,n) και j:[0,m) */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,m,i,j;

  scanf("%d %d", &n, &m);

  for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
      printf("%d,%d\n", i, j);
    }
  }

}
```

```
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    printf("%d,%d\n", i, j);
  }
}
```

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    printf("%d,%d\n", i, j);
```

Ισοδυναμία `while` <-> `for`

```
<expr1>
while <lexpr> {
  <body>
  <expr2>
}
```



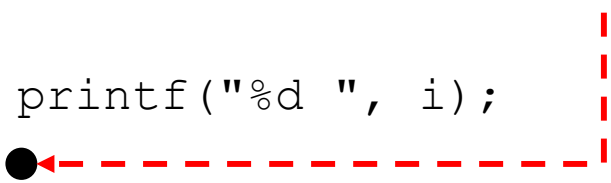
```
for (expr1;lexpr;expr2)
  <body>
```

- Για κάθε πρόγραμμα που χρησιμοποιεί `while` μπορεί πάντα να φτιαχτεί ένα ισοδύναμο πρόγραμμα που χρησιμοποιεί `for`, και το αντίστροφο.
- Η δομή `for` επιτυγχάνει καλύτερη αναγνωσιμότητα (και υπό προυποθέσεις την παραγωγή πιο αποδοτικού κώδικα από τον μεταφραστή), καθώς ξεχωρίζει τις εκφράσεις «αρχικοποίησης» και «πρόδου» από τον υπόλοιπο κώδικα της επανάληψης.

Η εντολές `break` και `continue`

- Η κανονική έξοδος μέσα από μια δομή επανάληψης πραγματοποιείται όταν η αντίστοιχη συνθήκη ελέγχου αποτιμάται (κάποια στιγμή) σε `0`, **πριν** ή **μετά** την εκτέλεση του αντίστοιχου «σώματος» / «μπλοκ».
- Σε κάποιες περιπτώσεις αυτό μπορεί να είναι αρκετά περιοριστικό και να κάνει τον κώδικα πολύπλοκο.
- Με την εντολή `break` επιτυγχάνεται «εξόδος» από οποιοδήποτε σημείο του κώδικα της επανάληψης.
- Με την εντολή `continue` παρακάμπτονται οι (υπόλοιπες) εντολές του κώδικα της επανάληψης, χωρίς έξοδο από την επανάληψη.
- **Σημείωση:** στη δομή `for` το `continue` **δεν** παρακάμπτει την έκφραση «προόδου».

```
/* εκτύπωση ζυγών αριθμών στο διάστημα [1..n) */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int n,i;

  scanf("%d", &n);

  for (i=1; i<n; i++) {

    if (i%2 != 0) { continue; }

    printf("%d ", i);

  }

  printf("\n");

}
```



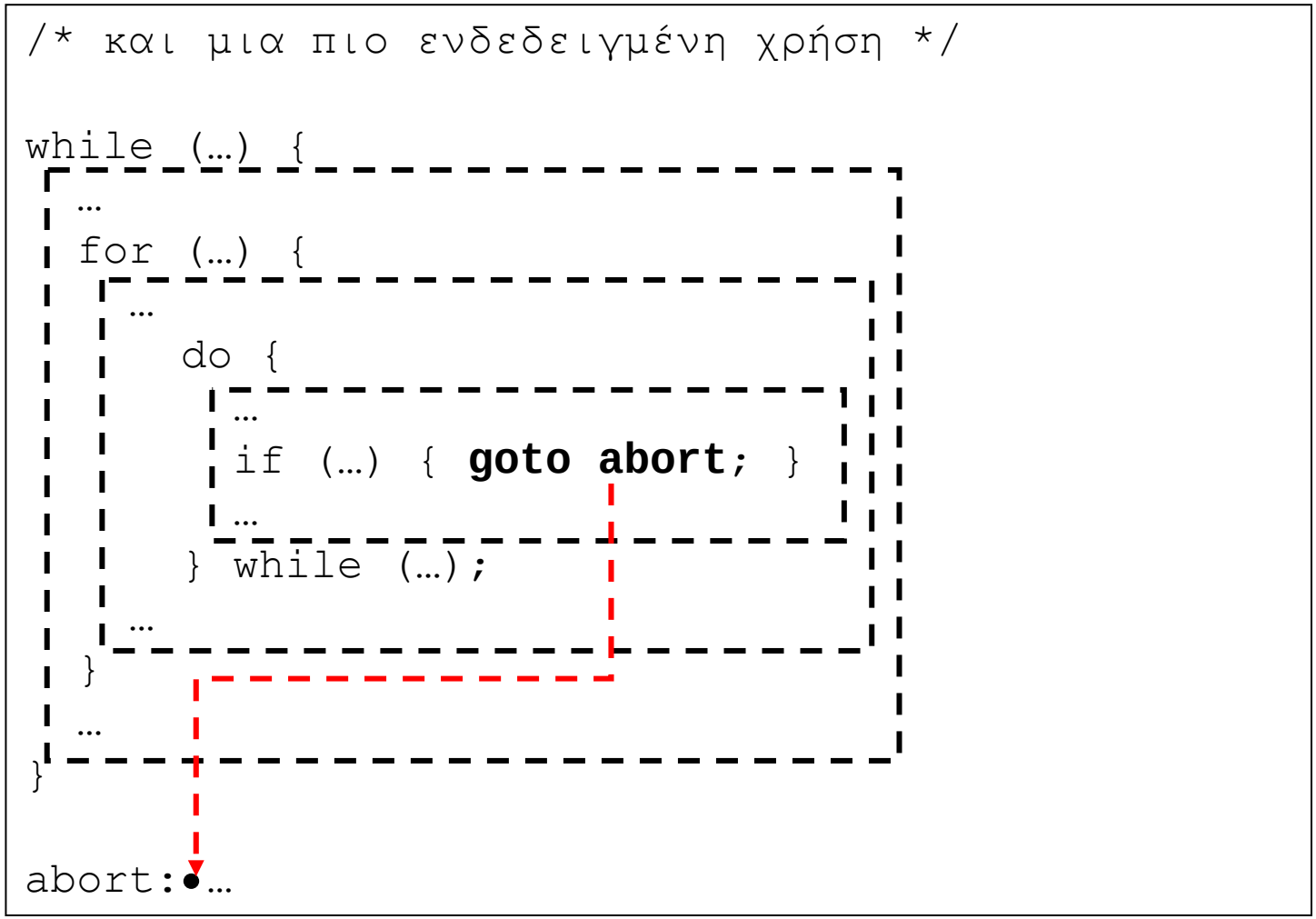
```
/* ανάγνωση και πρόσθεση δύο θετικών αριθμών,
   μέχρι να δοθεί μια τιμή μικρότερη-ίση 0  */

#include <stdio.h>

int main(int argc, char *argv[]) {

  int a,b;

  while (1) {

    printf("enter 2 ints > 0 or 0 to stop: ");
    scanf("%d",&a); scanf("%d",&b);

    if ((a <= 0) || (b <= 0)) { break; }

    printf("%d plus %d is %d\n", a, b, a+b);

  }

}
```

Η εντολή `goto`

- Με την εντολή `goto <label>` η εκτέλεση συνεχίζεται από το σημείο με την ετικέτα `<label>`.
- Η `goto` δίνει μεγάλη ευελιξία καθώς επιτρέπει την μεταφορά του ελέγχου σε οποιοδήποτε σημείο του προγράμματος, με άμεση έξοδο «μέσα από» πολλά επίπεδα επανάληψης.
- Χρειάζεται **προσοχή** ώστε να μην δημιουργούνται δυσνόητα προγράμματα, κάτι που πολύ εύκολα μπορεί να συμβεί με πρόχειρη χρήση της `goto`.
- Η `goto` χρησιμοποιείται ως τελευταία λύση, όταν όλοι οι υπόλοιποι συνδυασμοί δομών και εντολών ελέγχου κάνουν τον κώδικα λιγότερο ευανάγνωστο.

```
/* αντιπαράδειγμα */
…
get:    c = getchar();
        goto check1;
cont1:  goto check2;
cont2:  putchar(c);
        goto get;
…
check1: if (c == '\n') { goto theend; } else { goto cont1; }
…
check2: if (( c>= 'a') && (c <= 'z')) { c = c-32; }
        goto cont2;

theend: putchar('\n');
```

```
…
do {
  c = getchar();
  if ((c >= 'a') && (c <= 'z')) { c = c-32; }
  putchar(c);
} while (c != '\n');
```


/* και μια πιο ενδεδειγμένη χρήση */
while (…) {
…
for (…) {
…
do {
…
if (…) { goto abort; }
…
} while (…);
…
}
…
}
abort:…

Δομές επανάληψης χωρίς σώμα

- Στις δομές ελέγχου `while`, `do-while` και `for` μπορεί να μην χρειάζεται να βάλουμε κάποιο σώμα εντολών (πως μπορεί κάτι τέτοιο να έχει λογική;).
- Η C **δεν** υποστηρίζει την **απουσία** σώματος.
- Σε αυτή την περίπτωση έχουμε την επιλογή ανάμεσα στην χρήση
  - της «κενής» εντολής `;` (που δεν κάνει τίποτα)
  - του «άδειου» σώματος εντολών `{}` (που δεν περιέχει καμία εντολή)
- Το αποτέλεσμα είναι το ίδιο (δεν εκτελείται τίποτα).

```
/* υπολογισμός s=1+2+…N */

int i,s;

for (i=1,s=0; i<=N; s=s+i++) {}
```

το «άδειο» σώμα χωρίς εντολές

```
/* υπολογισμός s=1+2+…N */

int i,s;

for (i=1,s=0; i<=N; s=s+i++) ;
```

η «κενή» εντολή που δεν κάνει τίποτα

Γιατί τόσες δομές ελέγχου;

- Κάθε δομή ελέγχου έχει τα πλεονεκτήματα της, κυρίως όσον αφορά την αναγνωσιμότητα του κώδικα.
- Μερικές δομές διευκολύνουν τον μεταφραστή στην παραγωγή καλύτερου κώδικα μηχανής.
- Πολλές φορές η επιλογή γίνεται με βάση το προσωπικό στύλ του καθενός –φυσικά, διαφορετικοί άνθρωποι έχουν και διαφορετικά γούστα ...
- Πρωταρχικός στόχος για εσάς: **αναγνωσιμότητα**!
- Σημείωση: η όποια «βελτιστοποίηση» του κώδικα γίνεται **αφού** σιγουρευτούμε ότι το πρόγραμμα είναι σωστό (και **αφού** γίνουν κατάλληλες μετρήσεις που θα δείξουν το σημείο όπου χρειάζεται κάτι τέτοιο).

Σχόλιο

- Μην προσπαθείτε να φανείτε υπερβολικά (και προκαταβολικά) «έξυπνοι» όταν γράφετε κώδικα.
- Να φροντίζετε να γράφετε ένα πρόγραμμα με πρώτο κριτήριο την **αναγνωσιμότητα** του – δυστυχώς είναι εύκολο να γράφει κανείς ακατανόητο κώδικα.
- Βάζετε **σχόλια** στον κώδικα, και **όταν** χρειάζεται.
- Τυπικές περιπτώσεις όπου ένα σχόλιο βοηθάει: (α) περιγραφή λειτουργικότητας σε ψηλό επίπεδο, (β) τεκμηρίωση «περίεργου» κώδικα με παρενέργειες, (γ) χρησιμότητα μεταβλητών του προγράμματος.
- Συχνά, η (σωστή) μορφοποίηση του κειμένου είναι από μόνη της η πιο χρήσιμη περιγραφή του κώδικα.

```
/* αντιπαράδειγμα */

#include <stdio.h>

int main(int argc, char *argv[]) {

int i;   /* μεταβλητή ακεραίος i */
int s,n; /*άλλες δύο τέτοιες μεταβλητές*/

/* αρχικοποίηση μεταβλητών */
i=1;   /* i γίνεται 1 */
s=0;   /* s γίνεται 0 */

scanf("%d", &n); /* διάβασε τιμή */

/* αρχίζουμε τον υπολογισμό μας */
while (i<=n)  /* δεν έχουμε τελειώσει */  {
  s=s+i; /* αύξησε s κατά i */
  i++;   /* αύξησε i κατά 1 */
}

printf("%d\n", s); /* τύπωσε αποτέλεσμα */
```

```
/* αντιπα
    ρά
δειγμα */ #include <stdio.h>
int main(int argc, char *argv[]) {

int          i;/* μεταβλητή ακεραίος i */
int s,n;/*άλλες δύο τέτοιες μεταβλητές*/
/* αρχικοποίηση μεταβλητών */
i=1; /* i γίνεται 1 */
s=0;          /* s γίνεται 0 */
scanf("%d", &n); /* διάβασε τιμή */
/* και τώρα αρχίζουμε τον υπολογισμό μας */
while (i<=n)  /* δεν έχουμε τελειώσει */  {
  s=s+i;/* αύξησε s κατά i */ i++;/* αύξησε i
κατά 1 */}
printf("%d\n", s);
/* τέλος */
```

```c
/* καλύτερα */

#include <stdio.h>

int main(int argc, char *argv[]) {
  int n;      /* από είσοδο */
  int i;      /* μετρητής από 1 μέχρι (και) n */
  int s;      /* s == 0+1+…+i-1 */

  scanf("%d", &n);

  i = 1; s = 0;

  while (i <= n) {
    s = s+i;
    i++;
  }

  printf("%d\n", s);
}
```