

Limbaje de programare

## Iterația. Prelucrări de texte

24 octombrie 2011

## Orice program are o funcție main

```
int main(void)
{
    // --> AICI <-- scriem ce sa faca programul
    return 0;
}
```

Execuția programului începe cu funcția main

De aici apelăm toate celelalte funcții (scrise înainte).

## Codul se scrie doar în funcții

```
tip functie1 ( parametri )
{
    // codul functiei 1
}
// NU scriem cod între funcții
tip functie2 ( parametri )
{
    // codul functiei 2
}
// NU scriem cod aici

int main(void)
{
    // apelează funcție1, funcție2
    return 0;
}

// NU scriem cod aici
```

## Citiți atent enunțul

*citește*

*returnează*

*tipărește*

sunt verbe diferite!

O funcție *primește un parametru*

*când e apelată*: `f(5)`

NU citește *parametrii* de la intrare/tastatură!

O funcție care *returnează* o valoare: `return x + 2;`

NU o tipărește: `printf("%d", x + 2);`

NU o atribuie: `x = x + 2;`

## Funcțiile pot avea sau nu parametri și rezultat

```
int f(int x) { return (x + 2) % 26; }
```

Folosim rezultatul:

îl atribuim: `int y = f(3);`

îl tipărim: `printf("%d\n", f(25));`

într-o expresie: `putchar('a' + f('z' - 'a'));`  
(inclusiv în apelul la altă funcție)

Putem scrie funcții *fără rezultat*  $\Rightarrow$  tipul returnat e `void`

```
void print_hex(int n) {  
    putchar(n > 9 ? n - 10 + 'a' : n + '0');  
}
```

Folosim funcția într-o *instrucțiune*: `print_hex(n % 16);`

Funcții fără parametri: definim cu `void` în locul listei parametrilor

```
int citeste_mic(void) { return tolower(getchar()); }
```

Apelăm întotdeauna cu paranteze: `int c = citeste_mic();`

## Recursivitatea: șiruri recurente

În matematică, *un șir e o funcție definită pe numere naturale*:

$n$	0	1	2	3	4	5	...
$x_n$	3	5	7	9	11	13	...

Funcția ia ca parametru indicele  $n$  și returnează termenul  $x_n$   
⇒ la fel și funcția definită în program

Pentru un șir recurent (de ordinul 1):

$$x_n = \begin{cases} \text{valoarea inițială } (x_0) & \text{pentru } n = 0 \\ \text{expresie}(x_{n-1}) \text{ (în funcție de } x_{n-1}) & \text{altfel } (n > 0) \end{cases}$$

```
int sir(int n) {  
    return n == 0 ? termen_initial : expresie_folosind_sir(n-1) ;  
}  
  
int p_arit(int n) { return n == 0 ? 3 : p_arit(n-1) + 2; }
```

## Recursivitatea: calculul unei limite

Definim:  $t_0 = \text{valoare\_inițială}$  (pentru  $n = 0$ )

$t_n = \text{expresie / funcție de } t_{n-1}$  (pentru  $n > 0$ )

Dacă șirul are limită, putem calcula termeni până când diferența devine suficient de mică:

```
double limita(double term_crt)
{
    double term_nxt = expresie(term_crt);
    return fabs(term_nxt - term_crt) < precizie ?
        term_nxt : limita(term_nxt);
}
apelată cu limita(valoare_inicială)
```

## Iterația: citirea de caractere

Un ciclu tipic:

    citește un caracter

    verifică o condiție (dacă da, continuăm ciclul)

    prelucrăm caracterul (poate lipsi)

    citim următorul caracter, și revenim la test

```
int c = getchar();
while (!isspace(c) && c != EOF) {
    putchar(c);
    c = getchar();
}
```

Putem scrie mai scurt, citind caracterul în test (o singură dată!)

```
while (!isspace(c = getchar()) && c != EOF)
    putchar(c);
```

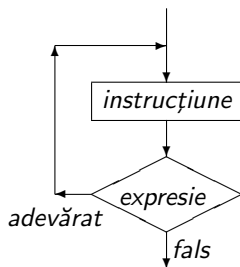
ATENȚIE! La atribuirii în condiții trebuie paranteze!

```
while ((c = getchar()) != EOF) ...
```



## Ciclul cu test final

```
do  
    instrucțiune  
while ( expresie );
```



Uneori știm sigur că un ciclu trebuie executat cel puțin o dată (citim cel puțin un caracter, un număr are măcar o cifră, etc.)

Ca și ciclul cu test inițial, execută *instrucțiune* atât timp cât execuția expresiei e nenulă (adevărată)

Expresia se evaluează însă *după* fiecare iterație

Echivalent cu:

```
instrucțiune  
while ( expresie )  
    instrucțiune
```

## Citirea caracter cu caracter: filtre

Frecvent: prelucrăm intrarea și extragem / calculăm ceva.

```
void skipspace(void)                void skipspace(void)
{                                     {
    int c;                            int c;
    while (isspace(c = getchar()));    do
    ungetc(c, stdin);                 c = getchar();
}                                     while (isspace(c));
                                     ungetc(c, stdin);
```

Ciclul are corpul ; (instrucțiunea *vidă*). }

**ATENȚIE!** Nu puneți ; din greșeală!

```
int wordlen(void) { // lungimea unui cuvânt citit
    int c, l = 0;
    while ((c = getchar()) != EOF && !isspace(c)) l++;
    return l;
}
```

**ATENȚIE**, testați întotdeauna sfârșitul intrării, poate apare oricând!  
Fără acest test, ciclul *s-ar bloca* când c e EOF (care nu e spațiu)

## Operatori de atribuire

**ATENȚIE:** Nu greșiți folosind atribuirea în loc de test de egalitate!!

if (x = y) testează: valoarea lui y (atribuită lui x) e nenulă ?

**Operatori compuși de atribuire:** += -= \*= /= %=

x += expr e o formă mai scurtă de a scrie x = x + expr

la fel și pentru operatorii pe biți >> << & ^ |

**Operatori de incrementare/decrementare** prefix/postfix: ++ --

++i crește i cu 1, valoarea expresiei este cea de *după* atribuire

i++ crește i cu 1, valoarea expresiei este cea *dinainte* de atribuire

expresiile au același *efect lateral* (atribuirea) dar *valoare* diferită

int x=2, y, z; y = x++; /\* y=2,x=3 \*/; z = ++x; // x=4,z=4

**ATENȚIE** Evitați expresii compuse cu mai multe efecte laterale!  
(nu e precizat care se execută întâi).

INCORECT: i = i++ (două atribuiri în aceeași expresie: = și ++)

INUTIL: c = toupper(c); return c; Bine: return toupper(c);

## Instrucțiunea break

Iese din corpul ciclului *imediat înconjurător*

Folosită dacă nu dorim să continuăm restul prelucrărilor din ciclu

De regulă: `if (condiție ) break;`

```
#include <ctype.h>
#include <stdio.h>
int main(void) {           // numără cuvintele din intrare
    int c;
    unsigned nrw = 0;
    while (1) {           // ciclu infinit, iese doar cu break;
        while (isspace(c = getchar())); // consumă spațiile
        if (c == EOF) break;           // gata, nu mai urmează nimic
        nrw = nrw + 1;                 // altfel e început de cuvânt
        while (!isspace(c = getchar()) && c != EOF); // cuvântul
    }
    printf("%u\n", nrw);
    return 0;
}
```

## Instrucțiunea for

<code>for (expr-init ; expr-test ; expr-actualiz)</code>	<code>expr-init;</code>
<code>instrucțiune</code>	<code>while (expr-test) {</code>
	<code>instrucțiune</code>
	<code>expr-actualiz;</code>
	<code>}</code>

e echivalentă\* cu:

\* excepție: instrucțiunea continue, vezi ulterior

Oricare din cele 3 expresii poate lipsi (dar cele două ; rămân)

Dacă *expr-test* lipsește, e tot timpul adevărată (ciclu infinit)

C99 permite în loc de *expr-init* o *declarație* de variabile (inițializate) cu domeniu de vizibilitate întreaga instrucțiune (dar nu și după)

Cel mai des folosit: *numără* (repetă de un număr fix de ori)

```
for (int i = 0; i < 10; ++i) { /*fă de 10 ori*/ } // i dispare  
int i; for (i = 1; i <= 10; ++i) { /*fă de 10 ori*/ } // i e 11
```

**ATENȚIE** Instrucțiunea ; e *instrucțiunea vidă*: nu face nimic!

Scriem ; după ) la while sau for doar pentru ciclu cu corp vid!

```
while (isspace(c = getchar())); (consumă oricâte spații)
```

## Exemplu: rescrie fiecare cuvânt cu majusculă

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
    int c;
    for (;;) {                // repetă continuu, iese doar cu break;
        while (isspace(c = getchar())) // cât timp sunt spații
            putchar(c);        // scrie și spațiile
        if (c == EOF) break;    // nu mai urmează nimic
        putchar(toupper(c));    // prima literă
        while ((c = getchar()) != EOF) {
            putchar(c);        // scrie caracter din cuvânt
            if (isspace(c)) break; // la primul spațiu iese
        }                    // a ajuns aici: reia ciclul
    }
    return 0;
}
```

## Scrierea ciclurilor

Ne gândim:

ce variabilă se modifică în fiecare iterație ?

care e condiția de continuare a ciclului ?

Nu uităm instrucțiunea care modifică acea variabilă  
(altfel ciclul continuă la infinit)

Ce știm la ieșirea din ciclu ? Condiția e *falsă*.

Ținem cont de asta când gândim mai departe programul.

Verificăm programul:

mental, rulându-l “cu creionul pe hârtie” (întâi pe cazuri simple)

apoi la rulare, cu teste tot mai complexe, și pentru situații limită