

Limbaje de programare

Decizia. Atribuirea. Iterația

20 octombrie 2009

Secvențierea

Pentru a face mai multe prelucrări (într-o funcție), scriem instrucțiunile una după alta (*secvențial*)

⇒ împreună cu *decizia* și *recursivitatea* putem scrie orice program

Instrucțiunea compusă: mai multe instrucțiuni grupate cu *acolade* { }

Un exemplu e de instrucțiune compusă (*bloc*) e chiar corpul unei funcții.

```
{                                     {
    instrucțiune                       int c = getchar();
    ...                                  printf("tiparim caracterul: ");
    instrucțiune                       putchar(c);
}                                     }
```

Instrucțiunea compusă e considerată *o singură instrucțiune*

Poate conține și declarații: oriunde (C99)/doar la început (ANSI C)

Orice instrucțiune care *nu e* compusă se termină cu *punct-virgulă* ;

Operatorul de secvențiere pentru expresii e *virgula*: *expr1* , *expr2*

Se evaluează *expr1*, se ignoră, valoarea întregii expresii e cea a lui *expr2*

Instrucțiunea condițională (if)

Operatorul condițional ? : selectează din două *expresii* de evaluat

Instrucțiunea condițională selectează din două *instrucțiuni* de executat

Sintaxa:

```
if ( expresie )                sau    if ( expresie )
    instrucțiune1                instrucțiune1
else
    instrucțiune2
```

Efectul:

Dacă expresia e *adevărată* se execută *instrucțiune1*,
altfel se execută *instrucțiune2* (sau nimic, dacă nu există)

Fiecare ramură are *o singură* instrucțiune. Dacă e nevoie de mai multe
instrucțiuni, trebuie grupate într-o *instrucțiune compusă* { }

Parantezele () din jurul condiției sunt obligatorii.

Expresii cu valoare logică în limbajul C

Obișnuit, *condiția* din instrucțiunea `if` sau operatorul `?:` e o expresie relațională, cu valoare logică: `x != 0`, `n < 5`, etc. Limbajul C a fost însă conceput fără un tip boolean dedicat.

O valoare se consideră *adevărată* dacă e *nenulă* și *falsă* dacă e *nulă* (atunci când e folosită ca și condiție: în `?:`, `if`, `while` etc.) \Rightarrow Condiția în `if` trebuie să aibă tip *scalar* (întreg, real, enumerare)

Corespunzător: *Operatorii de comparație* (`==` `!=` `<` etc.) întorc în C valorile *întregi* 1 (pentru *adevărat*) sau 0 (pentru *fals*)

C99 adaugă tipul `_Bool`, și are în fișierul `stdbool.h` definițiile macro: `bool` (pentru `_Bool`), `true` (pentru 1) și `false` (pentru 0)

O ramură `else` aparține întotdeauna de *cel mai apropiat* `if`
`if (x > 0) if (y > 0) printf("x+, y+"); else printf("x+, y-");`

Exemple cu instrucțiunea if

```
#include <stdio.h>
void printnat(unsigned n) { // tipareste recursiv un nr. natural
    if (n > 9) // daca are mai multe cifre
        printnat(n/10); // atunci tipareste si prima parte
    putchar('0' + n % 10); // oricum, tipareste ultima cifra
}
int main(void) { printnat(312); return 0; }
```

Tipărirea soluțiilor ecuației de gradul II:

```
void printsol(double a, double b, double delta) {
    if (delta >= 0) {
        printf("Sol. 1%f\n", (-b-sqrt(delta))/2/a);
        printf("Sol. 2%f\n", (-b+sqrt(delta))/2/a);
    } else printf("nu are solutie\n");
}
```

Operatorul condițional ? : se rescrie (mai puțin concis) cu `if`

```
int abs(int x) { if (x > 0) return x; else return -x; }
```

Operatori logici

Cu operatorii logici, putem scrie *decizii cu condiții complexe*:

Un an e bisect dacă: se divide cu 4 **și**
 nu se divide cu 100 **sau** se divide cu 400

```
int e_bisect(unsigned an) { // raspuns 1: e bisect, 0: nu e
    return an % 4 == 0 && (!(an % 100 == 0) || an % 400 == 0);
} // se putea scrie și (an % 100 != 0)
```

Reamintim: operatorii logici produc **1** pt. *adevărat*, **0** pt. *fals*

Un întreg e interpretat ca *adevărat* dacă e *nenul*, și ca *fals* dacă e **0**

<i>expr</i>	! <i>expr</i>
0	1
$\neq 0$	0

negație **!** NU

		<i>e</i> ₂	
		0	$\neq 0$
<i>e</i> ₁	<i>e</i> ₁ && <i>e</i> ₂	0	0
	<i>e</i> ₁ && <i>e</i> ₂	0	1

conjuncție **&&** ȘI

		<i>e</i> ₂	
		0	$\neq 0$
<i>e</i> ₁	<i>e</i> ₁ <i>e</i> ₂	0	1
	<i>e</i> ₁ <i>e</i> ₂	1	1

disjuncție **||** SAU

Precedența operatorilor logici

Operatorul logic unar ! (negație logică): precedență cea mai ridicată

if (!gasit) e echivalent cu if (gasit == 0) (nul e fals)

if (gasit) e echivalent cu if (gasit != 0) (nenul e adevărat)

Operatorii relaționali: precedența mai mică decât cei aritmetici

⇒ putem scrie natural $x < y + 1$ pentru $x < (y + 1)$

Precedența: întâi $>$ $>=$ $<$ $<=$, apoi $==$ $!=$ (egal, diferit)

Operatorii logici binari: $\&\&$ (ȘI) e prioritar lui $\|\|$ (SAU)

Au precedență mai mică decât cei relaționali

⇒ putem scrie natural $x < y + z \&\& y < z + x$

Evaluarea în scurt-circuit

Evaluarea expresiilor logice se face de la stânga la dreapta.

Evaluarea se oprește (scurt-circuit) când rezultatul e cunoscut:

primul argument e fals a `&&`

primul argument e adevărat la `||`

```
if (p != 0 && n % p == 0)    if (p != 0)           // doar atunci fa
    printf("p e divizor");    if (n % p == 0)       // si testul 2
                              printf("p e divizor");
```

⇒ Atenție la modul cum scriem testele compuse !

Atribuirea

În *recursivitate* fiecare apel creează *noi copii* de parametri cu *alte valori*
Uneori ajunge să *atribuim* (dăm) *o valoare nouă* unei variabile

Sintaxa: *variabilă = expresie* Totul e o *expresie (de atribuire)*.

Efect: Se evaluează expresia; valoarea se *atribuie* variabilei (și devine valoarea întregii expresii). `c = getchar()` `n = n-1` `r = r * n`

Poate fi folosită în alte expresii: `if ((c = getchar()) != EOF) ...`

inclusiv atribuire în lanț `a = b = x + 3` (a și b primesc aceeași valoare)

Orice *expresie* (ex. apel de funcție, atribuire) cu `;` devine *instrucțiune*
`printf("salut");` `printnat(n);` `c = getchar();` `x = x + 1;`

O variabilă *se poate modifica doar prin atribuire*, nu prin transmiterea ca parametru la funcții, sau prin alte expresii!

`n + 1` `sqr(x)` `toupper(c)` calculează ceva, NU modifică nimic!

ATENȚIE! `=` operatorul de atribuire `==` operatorul de comparare.

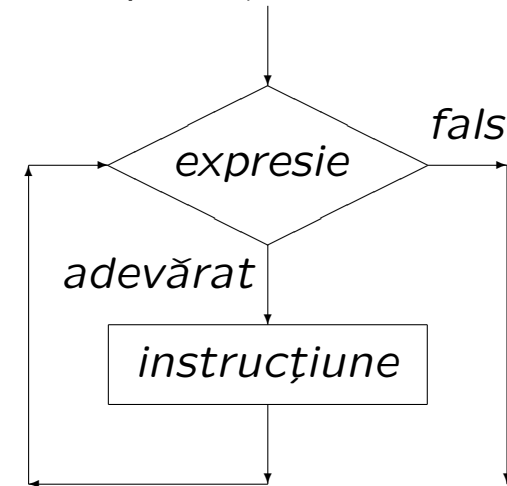
Iterația. Ciclul cu test inițial

Am scris funcții recursive ca să *repetăm* prelucrări – ceva esențial. Adesea, putem controla direct repetiția unei instrucțiuni, cu o condiție:

Sintaxa:

```
while ( expresie )  
    instrucțiune
```

ATENȚIE! Parantezele ()
sunt obligatorii la expresie!



Semantica (efectul): evaluează expresia. Dacă e adevărată (nenulă):

(1) se execută instrucțiunea (*corpul* ciclului)

(2) se revine la începutul lui `while` (evaluarea expresiei)

Altfel (dacă condiția e falsă/nulă) nu se execută nimic.

⇒ corpul se execută repetat *atât timp* cât condiția e adevărată

Putem defini iterația recursiv. Pct. (2) = “execută instrucțiunea `while`”

Rescrierea recursivității ca iterație

```
unsigned fact_r(unsigned n,
                unsigned r) {
    return n > 0
        ? fact_r(n - 1, r * n)
        : r;
} // apelat cu fact_r(n, 1)

int pow_r(int x, unsigned n,
          int r) {
    return n > 0
        ? pow_r(x, n-1, x*r)
        : r;
} // apelat cu pow_r(x, n, 1)

unsigned fact_it(unsigned n) {
    unsigned r = 1;
    while (n > 0) {
        r = r * n;
        n = n - 1;
    }
    return r;
}

int pow_it(int x, unsigned n) {
    int r = 1;
    while (n > 0) {
        r = x * r;
        n = n - 1;
    }
    return r;
}
```

Rescrierea recursivității ca iterație

- se face mai direct dacă funcția e *recursivă la dreapta*: e scrisă cu acumularea rezultatului parțial, transmis mai departe ca parametru (r)
- testul de oprire și valoarea inițială pentru rezultat rămân aceleași
- în varianta recursivă, fiecare apel creează *copii noi* de parametri, cu valori proprii (în funcție de cele vechi): ex. $n * r, n - 1, x * r$, etc.
- varianta iterativă, *actualizează (atribuie)* la fiecare iterație valorile variabilelor, după aceleași relații. Ex. $r = n * r, n = n - 1, r = x * r$
- ambele variante returnează valoarea acumulată a rezultatului

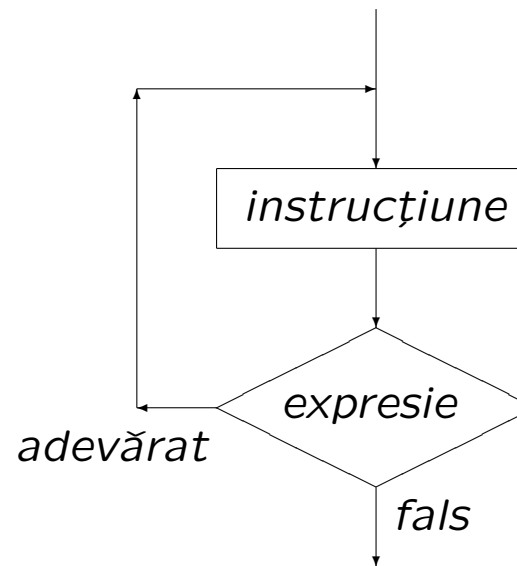
ATENȚIE: recursivitatea și iterația produc ambele prelucrări *repetate*.
⇒ în probleme simple folosim una sau cealaltă, rareori amândouă!

Citirea iterativă a unui număr, cifră cu cifră

```
#include <ctype.h> // pentru isdigit()
#include <stdio.h> // pt. getchar(), ungetc(), stdin
unsigned readnat(void)
{
    int c; unsigned r = 0; // caracterul si rezultatul
    while (isdigit(c = getchar())) // cat timp e cifra
        r = 10*r + c - '0'; // compune numarul
    ungetc(c, stdin); // pune inapoi ce nu-i cifra
    return r;
}
int main(void) {
    printf("numarul citit: %u\n", readnat());
}
```

`ungetc(c, stdin)` pune înapoi caracterul `c` în intrarea standard
Caracterul va fi preluat de următorul apel de citire, de ex. `getchar()`

```
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* atâț timp când 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 skipspaces(void) {
    int c;
    while (isspace(c = getchar()));
    ungetc(c, stdin);
}

void skipspaces(void) {
    int c;
    do
        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 apărea 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 dacă valoarea lui `y` (atribuită și lui `x`) e nenulă.

Operatori compuși de atribuire: `+=` `--` `*=` `/=` `%=`
`x += expr` e o formă mai scurtă de a scrie `x = x + expr`
vezi ulterior și pentru operatorii pe biți `>>` `<<` `&` `^` `|`

Operatori de incrementare/decrementare prefix/postfix: `++` `--`
`++i` incrementare cu 1, valoarea expresiei este cea de *după* atribuire
`i++` incrementare 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).

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

ATENȚIE Atribuim doar variabile, nu definim cu `=` valoarea funcției.
INCORECT: `int fact(int n) {fact(0) = 1; fact(n) = n*fact(n-1);}`
INUTIL: `c = toupper(c); return c;` Suficient: `return toupper(c);`

- produce ieșirea din corpul ciclului *imediat înconjurător*
- folosită dacă nu dorim să continuăm restul prelucrărilor din ciclu
- de regulă: `if (conditie) break;`

```
#include <ctype.h>
#include <stdio.h>
int main(void) {                                // numără cuvintele din intrare
    int c;
    unsigned nrw = 0;
    while (1) {                                  // condiție adevărată, 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); // cuvaâtul
    }
    printf("%u\n", nrw);
    return 0;
}
```

Exemplu: rescrie fiecare cuvânt cu majusculă

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
    int c;
    for (;;) {                // condiție adevărată, iese doar cu break;
        while (isspace(c = getchar())) // cât timp citește spații
            putchar(c);        // se scriu 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
        }                    // și reia ciclul for
    }
    return 0;
}
```

Instrucțiunea for

`for (expr-init ; expr-test ; expr-actualiz)`
instrucțiune

```
expr-init;
while (expr-test) {
    instrucțiune
    expr-actualiz;
}
```

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)

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

Cel mai des folosit: pentru a *număra* (repetă de un număr fix de ori)

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

ATENȚIE Instrucțiunea ; e caz particular al instrucțiunii *expresie* ; cu expresia vidă: nu face nimic! Scriem ; după) la `while` sau `for` doar dacă vrem ciclu cu corp vid (doar cu `test`, iar la `for` și cu *expr-actualiz*)

```
while (isspace(c = getchar())); (consumă secvență de spații)
```

Scrierea ciclurilor

În conceperea programelor care conțin cicluri

- identificăm ce variabilă se modifică în fiecare iterație
- identificăm care e condiția de oprire
- nu uităm instrucțiunea care modifică acea variabilă (altfel ciclul continuă la infinit)

Definim precis ce știm despre program când iese dintr-un ciclu.

- la ieșirea dintr-un ciclu, condiția e falsă
- ⇒ ne spune ceva despre valorile posibile ale variabilelor din condiție
- Folosim* această informație pentru a gândi mai departe programul.

Verificăm programul:

- mental, executându-l “cu creionul pe hârtie” (întâi pe cazuri simple)
- apoi la rulare, cu teste tot mai complexe, și pentru situații limită