Limbaje de programare

# Decizia. Atribuirea. Iterația

15 octombrie 2012

# Recapitulare: expresia condițională

*condiție* ? *expr1* : *expr2* totul e o *expresie*

*expr1* sau *expr2* pot fi și ele expresii condiționale
(dacă trebuie mai multe întrebări pentru a afla răspunsul)

$$f(x) = \begin{cases} -6 & x < -3 \\ 2 \cdot x & x \in [-3, 3] \\ 6 & x > 3 \end{cases}$$

```
double f(double x)
{
  return x < -3 ? -6        // altfel, ştim că x >= -3
                : x <= 3 ? 2*x : 6;
}
```

sau: `x >= -3 ? (x <= 3 ? 2*x : 6) : -6`

dacă $x \geq -3$ încă nu știm răspunsul, întrebăm $x \leq 3$ ?

sau: `x < -3 ? -6 : (x > 3 ? 6 : 2*x)`

dacă $x$ nu e $< -3$ și nici $> 3$, înseamnă că $x \in [-3, 3]$

# Expresia condițională (cont.)

Expresia condițională se poate folosi *oriunde* trebuie o expresie

Exemplu: expresie de tip șir în `printf`
(programul afișează ce fel de caracter a fost introdus):

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

int main(void)
{
  int c = getchar();    // var. iniţializată cu caract. citit
  printf(isupper(c) ? "e litera mare"
         : islower(c) ? "e litera mica"
         : isdigit(c) ? "e cifra"
         : "nu e nici litera nici cifra");
  putchar('\n');
  return 0;
}
```

# Recapitulare: Recursivitatea

Permite să rezolvăm o problemă:

1) găsind un caz de bază, când putem da răspunsul direct
2) un pas în care rezolvăm *aceeași* problemă *cu date mai simple* (*aceeași* funcție cu *alți parametri*)

```
unsigned sum_to(unsigned n)      // suma de la 1 la n
{
  return n == 0 ? 0              // nu avem ce aduna: 0
    : sum_to(n - 1) + n;         // (1 + ... + n-1) + n
}
```

# Expresii și instrucțiuni

Expresia: efectuează un calcul

operații aritmetice: `x + 1`

apel de funcție: `fact(5)`

Instrucțiunea: execută o acțiune

`return n + 1;`

Orice *expresie* la care se adaugă `;` devine `instrucţiune`

`n + 3;` (calculează, dar nu face nimic cu rezultatul)

`printf("hello!");` nu folosim *rezultatul* lui `printf`

ci ne interesează *efectul lateral*, tipărirea

`printf` returnează un `int`: numărul de caractere scrise (rar folosit)

# Secvențierea

Instrucțiunile se scriu și execută una după alta (*secvențial*)

Cu *decizie*, *recursivitate* și *secvențiere* putem scrie orice program.

*Instrucțiunea compusă*: mai multe instrucțiuni între *acolade* `{ }`
*Corpul unei funcții* e o instrucțiune compusă (*bloc*) .

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

Instrucțiunea compusă e considerată *o singură instrucțiune*.
Poate conține declarații: oriunde (C99/C11)/la început (ANSI C).
Orice instrucțiune care *nu e* compusă se termină cu *punct-virgulă* `;`

Pentru expresii *operatorul de secvențiere* e *virgula*: *expr1* `,` *expr2*
Se evaluează *expr1*, se ignoră, valoarea expresiei e cea a lui *expr2*

# Instrucțiunea condițională (`if`)

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

*Instrucțiunea condițională* selectează între *instrucțiuni* de executat

*Sintaxa:*

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

sau

```
if ( expresie )
   instrucțiune1
```

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

# Instrucțiunea condițională (`if`)

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

*Instrucțiunea condițională* selectează între *instrucțiuni* de executat

*Sintaxa:*

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

sau

```
if ( expresie )
   instrucțiune1
```

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

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

*Parantezele* `( )` din jurul condiției sunt obligatorii.

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`

Tipărirea soluțiilor ecuației de gradul II:

```
void printsol(double a, double b, double c)
{
  double delta = b * b - 4 *a * c;
  if (delta >= 0) {
    printf("solutia 1: %f\n", (-b-sqrt(delta))/2/a);
    printf("solutia 2: %f\n", (-b+sqrt(delta))/2/a);
  } else printf("nu are solutie\n");
}
```

Putem rescrie *operatorul condițional* `? :` cu *instrucțiunea* `if`

```
int abs(int x)
{
  return x > 0 ? x : -x;
}
```

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

# Exemple cu `if`: tipărirea unui număr

```
#include <stdio.h>

void printnat(unsigned n) { // tipareste recursiv nr. nat.
  if (n >= 10)              // daca are mai multe cifre
    printnat(n/10);         // scrie si prima parte
  putchar('0' + n % 10);    // oricum, scrie ultima cifra
}

int main(void)
{
  printnat(312);
  return 0;
}
```

# Expresii cu valoare logică în limbajul C

*Condiția* din instrucțiunea `if` sau operatorul `? :` e deobicei 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`, cu definițiile din fișierul `stdbool.h`
`bool` (pentru `_Bool`), `true` (pentru `1`) și `false` (pentru `0`)

# Operatori logici

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

| *expr* | ! *expr* |
|---|---|
| 0 | 1 |
| $\neq 0$ | 0 |

negație ! NU

| $e_1$ && $e_2$ | $e_2$ = 0 | $e_2$ $\neq 0$ |
|---|---|---|
| $e_1$ = 0 | 0 | 0 |
| $e_1 \neq 0$ | 0 | 1 |

conjuncție && ȘI

| $e_1$ \|\| $e_2$ | $e_2$ = 0 | $e_2$ $\neq 0$ |
|---|---|---|
| $e_1$ = 0 | 0 | 1 |
| $e_1 \neq 0$ | 1 | 1 |

disjuncție || SAU

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

# Exemplu: an bisect

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)        // 1: e bisect, 0: nu e
{
  return an % 4 == 0 && (!(an % 100 == 0) || an % 400 == 0);
}
```

`!(an % 100 == 0)` e echivalent cu `(an % 100 != 0)`

# Precedența operatorilor logici

*Operatorul logic* unar `!` (negație logică): precedență cea mai mare
`if (!gasit)` e la fel ca `if (gasit == 0)` (nul e fals)
`if (gasit)` e la fel ca `if (gasit != 0)` (nenul e adevărat)

*Operatorii relaționali*: precedența mai mică decât cei aritmetici
$\Rightarrow$ putem scrie natural `x < y + 1` pentru `x < (y + 1)`
Precedența: `>` `>=` `<` `<=`, apoi `==` `!=` (egal, diferit)

*Operatorii logici* binari: `&&` (ȘI) e prioritar lui `||` (SAU)
Au precedență mai mică decât cei relaționali
$\Rightarrow$ putem scrie natural `x < y + z && y < z + x`

# Evaluarea în scurt-circuit

Evaluarea expresiilor logice se face *de la stânga la dreapta*.
(pentru alți operatori în general, ordinea de evaluare nu e precizată)
*Evaluarea se oprește* (*scurt-circuit*) când rezultatul e cunoscut:
  la `&&`, când primul argument e fals (nu se mai evaluează restul)
  la `||`, când primul argument e adevărat

```
if (p != 0 && n % p == 0)
  printf("p divide pe n");
```

```
if (p != 0)                  // doar daca pe e nenul
  if  (n % p == 0)           // atunci testeaza restul
    printf("p divide pe n");
```

$\Rightarrow$ Atenție la modul cum scriem testele compuse !

Ordinea de evaluare și precedența sunt două noțiuni diferite!
În `2 * f(x) + g(x)` : înmulțirea înainte de adunare (*precedența*)
*NU* e precizat dacă care parte a sumei se *evaluează* întâi (`f` sau `g`)

# Atribuirea

În funcțiile recursive, nu a trebuit să *modificăm* valoarea variabilelor
stil de programare folosit în *limbajele funcționale* (pure)
Apelurile *recursive* creează *noi copii* de parametri cu *alte valori*.

În *programarea imperativă*, folosim:
*variabile* pentru a reprezenta un obiect din rezolvarea problemei
(caracter curent; rezultat parțial; număr rămas de prelucrat)
*atribuirea*, pentru a da o *valoare nouă* unei variabile
(exprimăm un pas de calcul făcut în program)

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

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

# Atribuirea (cont.)

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

Atribuirea în lanț: `a = b = x+3` (a și b primesc aceeași valoare)

Orice *expresie* (apel de funcție, atribuire) cu `;` devine *instrucțiune*

```
printf("salut");   c = getchar();   x = x + 1;
```

O variabilă *se poate modifica doar prin atribuire*,
*NU* se modifică prin alte expresii, sau transmisă ca parametru!

`n + 1`  `sqr(x)`  `toupper(c)`  calculează dar *NU modifică!*

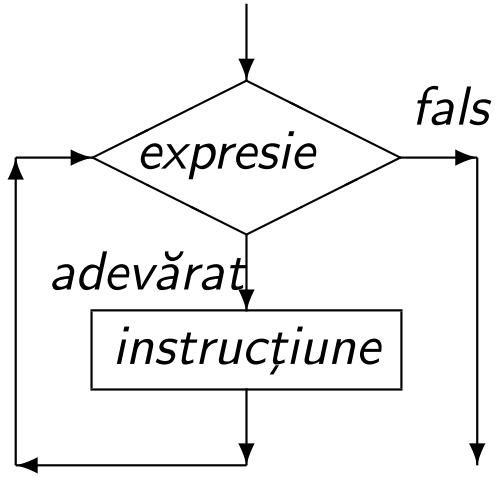
`n = n + 1`  `x = sqr(x)`  `c = toupper(c)`  *modifică*

*ATENȚIE!*  `=` operator de atribuire  `==` operator de comparare.

# Iterația. Ciclul cu test inițial

Am scris funcții recursive ca să *repetăm* prelucrări.
Putem exprima repetiția unei instrucțiuni, cu o condiție:

*Sintaxa*:
`while` ( *expresie* )
  *instrucțiune*




*ATENȚIE!* Parantezele ( ) sunt obligatorii la expresie!

*Semantica*: 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.
$\Rightarrow$ corpul se execută repetat *atât timp* cât condiția e adevărată

# Iterație și recursivitate

Putem defini iterația recursiv:

```
while ( expresie )
   instrucțiune
```

are același efect ca:

```
if ( expresie ) {
   instrucțiune
   while ( expresie )
      instrucțiune
}
```

# Rescrierea recursivității ca iterație

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

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

```
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)
```

```
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 recursivă e scrisă cu acumularea rezultatului parțial `r`, transmis ca parametru (tail recursion)

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*: și recursivitatea și iterația repetă prelucrări
$\Rightarrow$ într-o prelucrare 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 citit la următoarea citire, de ex. cu `getchar()`

# Citirea caracter cu caracter: filtre

Ex.: funcție care citește și tipărește până la un caracter dat; returnează acel caracter sau EOF dacă nu a apărut

```
int printto(int stopchar)       // pana la ce caracter
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar)
    putchar(c);
  return c;
}
```

*NU uitați parantezele:* `(c=getchar())!=EOF` (atribuie, apoi compară)

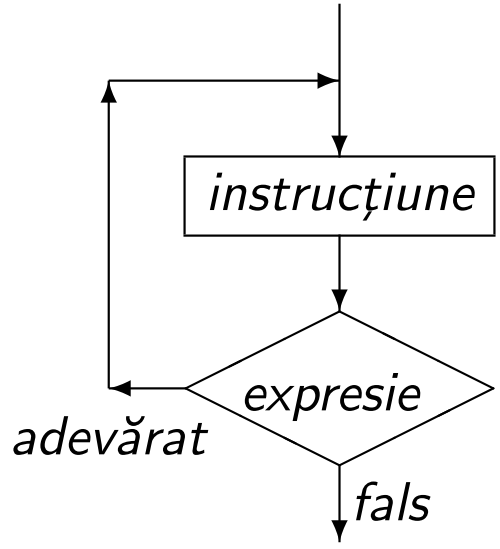
```
int skipto(int stopchar) // ignoră până la stopchar
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar);
  return c;
}
```

`;` după `while(...)` e *instrucțiunea vidă* (nu face nimic)

*NU puneți* `;` în alte cazuri (din greșeală)

# 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
```