Limbaje de programare

Decizia. Atribuirea. Iterația

17 octombrie 2011

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!");` are un rezultat, dar se ignora;

(o folosim pentru *efectul lateral*, tipărirea)

Secvențierea

Instrucțiunile într-o funcție se scriu una după alta (*secvențial*)
$\Rightarrow$ împreună cu *decizia* și *recursivitatea* 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 și declarații: oriunde (C99)/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 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)

Fiecare ramură are *o singură* instrucțiune. Dacă sunt 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`, cu definițiile din fișierul `stdbool.h`
`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 nr. nat.
  if (n > 9)                // 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; }
```

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("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) {   // 1: e bisect, 0: nu e
  return an % 4 == 0 && (!(an % 100 == 0) || an % 400 == 0);
}    // se putea scrie şi (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

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

negație ! NU

e_1 && e_2	0	$\neq 0$
0	0	0
$\neq 0$	0	1

conjuncție && ȘI

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

disjuncție || SAU

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.
Evaluarea se oprește (*scurt-circuit*) când rezultatul e cunoscut:
  la &&, când primul argument e fals
  la ||, când primul argument e adevărat

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

⇒ Atenție la modul cum scriem testele compuse !

Atribuirea

Apelurile *recursive* creează *noi copii* de parametri cu *alte valori*
Dar uneori ajunge să *atribuim* (dăm) *o valoare nouă* unei variabile

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`

Poate apare în alte expresii: `if ((c = getchar()) != EOF) ...`
Atribuirea în lanț `a = b = x + 3` (`a` și `b` primesc aceeași valoare)

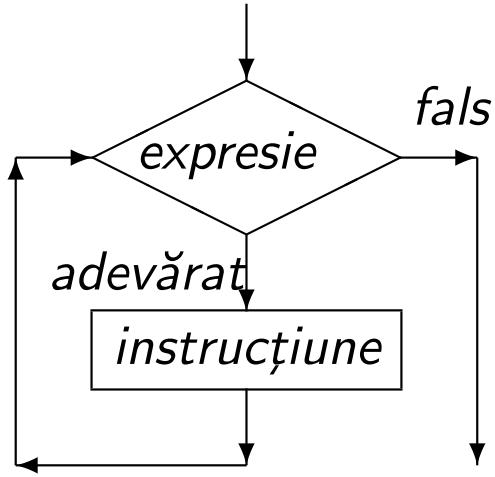
Orice *expresie* (apel de funcție, atribuire) cu `;` devine *instrucțiune*
`printf("salut");` `c = getchar();` `x = x + 1;`

O variabilă *se poate modifica doar prin atribuire*,
NU se modifică scriind alte expresii, sau transmisă ca parametru!!
`n + 1` `sqr(x)` `toupper(c)` calculează dar NU 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;
} // 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 apoi 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: 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

Exemplu: funcție care citește și ignoră până la un caracter dat; returnează acel caracter sau EOF dacă nu a apărut

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