

# Decizia. Iterația

Marius Minea

21 martie 2006

## Variabile locale

---

Un program  $C$  e o colecție de funcții  $\Rightarrow$  e scris *modular*: fiecare funcție rezolvă o subproblemă, și programul principal (`main`) le combină.

Numele *parametrilor* unor funcții diferite *nu* se influențează ca și în matematică putem avea  $f(x) = \dots$  și  $g(x) = \dots$

$\Rightarrow$  la fel pentru *variabilele* declarate în funcții (variabile locale)

*Domeniul de vizibilitate* al unui identicator = fragmentul de program unde poate fi utilizat (înțelesul său e cunoscut).

Variabilele declarate în funcții au domeniul de vizibilitate corpul funcției (din punctul declarației până la acolada de sfârșit).

Parametrii unei funcții au domeniul de vizibilitate tot corpul funcției  $\Rightarrow$  *nu* declarăm variabile cu același nume, ar fi un conflict

Variabilele locale au *durată de memorare automată*: ele sunt create la fiecare apel al funcției și distruse la încheierea acestuia (nu există și deci nu își păstrează valoarea între apeluri).

## Recapitulare: instrucțiuni și secvențiere

---

O *instrucțiune* e un element de limbaj ce specifică o *acțiune* care să fie executată de program. Am discutat:

– instrucțiunea *return*: `return expresie ;`

*evaluatează* expresia, valoarea ei devenind valoarea funcției și *încheie* execuția funcției (ea *returnează* valoarea calculată)

– instrucțiunea *expresie*: `expresie ;`

*evaluatează* expresia; esențiale sunt *efectele laterale*. Exemple:

`printf("salut");` valoarea (nefolosită) e numărul de caractere tipărite

`x = 2 + 3;` se *evaluatează* expresia din dreapta atribuirii; se atribuie la variabilă; valoarea *întregii expresii* e valoarea atribuită

Important: *efectul lateral* al atribuirii (modificarea valorii variabilei)

`2 + 3;` e o instrucțiune corectă, dar fără efect (valoarea nu e folosită)

Corpul unei funcții poate conține o *secvență* de mai multe instrucțiuni; ele se execută *secvențial*, în ordinea dată de program (exceptii ulterior)

*Secvențierea* e elementul de bază prin care scriem programe complexe

## Instrucțiunea compusă

---

E util să putem trata compunerea a mai multe elemente de limbaj ca un singur element de același fel.

(am văzut deja că  $2$  și  $n+1$  sunt expresii, la fel și  $2 * (n+1)$ )

O *instrucțiune* e o *acțiune* de executat de către program

Putem grupa mai multe instrucțiuni între acolade într-o *instrucțiune compusă (bloc)*:

```
instrucțiune-compusa ::= {  
                                instrucțiune  
                                ...  
                                instrucțiune  
                                }
```

Mai general, un bloc poate conține o secvență de *declarații și instrucțiuni*

## Exemplu: tipărirea cifră cu cifră a unui număr

---

Varianta 1: dorim să și numărăm câte cifre are

```
#include <stdio.h>
int printint(unsigned n) // tipărește și returnează numărul de cifre
{
    int r;                // pentru numărul de cifre
    r = n > 9 ? printint(n/10) : 0; // apel pentru n > 10
    putchar('0' + n % 10); // tipărește ultima cifră
    return r + 1;        // cu 1 mai mult decât n / 10
}
int main(void)
{
    printint(312);
    return 0;
}
```

## Exemplu: varianta fără returnarea unui rezultat

---

Putem defini funcții care *nu* întorc un rezultat, și sunt utile doar pentru efectul lateral (de exemplu o tipărire).

Tipul returnat de funcție e tipul `void` (tipul vid, fără valori)

– nu e necesar `return`, execuția se încheie la acolada `}`

(se *poate* folosi `return`; pentru a termina explicit execuția)

Pentru tipărirea pe cifre a unui număr:

– dacă numărul e  $> 9$

– tipărim pe  $n/10$

– tipărim ultima cifră

– dacă nu (numărul e  $< 10$ )

– tipărim unica cifră

Probleme: – pe una din ramuri dorim să efectuăm *două* lucruri.

– ele nu sunt neapărat *valori* diferite ale unei expresii (pt. care folosim *operatorul condițional*), ci *acțiuni* diferite (adică *instrucțiuni*)

## Instrucțiunea condițională (if)

---

Instrucțiunea **if** `if ( expresie ) instrucțiune1`

sau `if ( expresie ) instrucțiune1 else instrucțiune2`

- expresia trebuie să fie de tip scalar (întreg, real, enumerare)
- dacă expresia e adevărată se execută *instrucțiune1*, altfel *instrucțiune2*
- un `else` e asociat întotdeauna cu cel mai apropiat `if`

La origine, în C nu există un tip special pentru valori logice (boolene)

⇒ unde e necesară o expresie logică ( ? : , if ) se folosește convenția:

o valoare *nenulă* reprezintă *adevărat*, o valoare *nulă* reprezintă *fals*

Operatorii relaționali și de egalitate produc valorile 1 pentru adevărat, și 0 pentru fals.

## Operatorul și instrucțiunea condițională

---

Putem rescrie acum `return cond ? expr1 : expr2 ;` cu  
`if ( cond ) return expr1 ; else return expr2 ;`

Exemplu: sarcina J1: start 3, durată 7; sarcina J2: start la  $x$ , durată 5

```
int fin_J2 (int x) {  
    return x < -2 ? x + 5  
           : x < 10 ? x + 12  
           : x + 5;  
}
```

sau:

```
int fin_J2 (int x) {  
    if (x < -2) return x + 5;  
    else if (x < 10) return x + 12;  
    else return x + 5;  
}
```



## Operatorii logici && (ȘI), || (SAU), ! (NU)

---

Pentru a trata condiții complexe, folosim operatori pentru valori logice:

&& : adevărat (1) pt. ambii operanzi adevărați (nenuli), altfel fals (0)

|| : adev. (1) pt. cel puțin un operand adevărat (nenul), altfel fals (0)

! : adev. (1) pt. operand fals (0); fals pt. operand adevărat (nenul)

– ! are precedență maximă  $!(a || b) == !a \&\& !b$  (paranteze necesare)

– && e prioritar lui ||, ambii au precedență mai mică decât cei relaționali

⇒ se poate scrie natural  $(x < y + z \&\& y < z + x)$

– sunt evaluați de la stânga la dreapta

– *evaluarea se oprește (short-circuit)* când rezultatul e cunoscut (dacă primul argument al lui && (resp. ||) e fals (resp. adevărat)

Exemplu: `if (p != 0 && n % p == 0) { /* nu împarte la 0 */ }`

Ex.: `isdigit(c)` echivalent cu `c >= '0' && c <= '9'`

Ex.: `return x < -2 || x >= 10 ? x + 5 : x + 12;`

Ex.: `if (x < -2 || x >= 10) return x + 5; else return x + 12;`

## Ciclul cu test inițial (`while`)

---

Sintaxa: `while ( expresie ) instrucțiune`

Semantica: se evaluează expresia.

– dacă este adevărată (nenulă):

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

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

– altfel (dacă e falsă) nu se execută nimic

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

Semantica instrucțiunii `while` poate fi exprimată ea însăși recursiv, înlocuind (2) cu: “se execută instrucțiunea `while`”

```
void copyline(void) { // exemplu: copierea unei linii de la intrare
    int c = getchar();
    while (c != EOF && c != '\n') {
        putchar(c);
        c = getchar();
    }
    if (c == '\n') putchar('\n');
}
```

## Rescrierea recursivității ca iterație

---

```
unsigned fact_r(unsigned n,  
                unsigned r) {  
    return n != 0  
        ? fact_r(n - 1, n * r)  
        : 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 = n * r;  
        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 recursivă e scrisă cu acumularea valorii parțial calculate, transmisă mai departe ca parametru (în exemple,  $r$ ) (acesta devine variabilă locală în scrierea iterativă)
- testul de oprire și valoarea inițială pentru rezultat sunt aceleași
- varianta recursivă are propria valoare a parametrului  $r$ , calculată (în funcție de cea precedentă) la fiecare apel ( $n * r$ , respectiv  $x * r$ )
- în varianta iterativă, valoarea variabilei  $r$  e *actualizată* la fiecare iterație, după aceeași relație ( $n * r$ , respectiv  $x * r$ )
- la fel, folosirea altei valori ( $n - 1$ ) pentru  $n$  în apelul recursiv se rescrie ca actualizare a valorii lui  $n$  în corpul iterației
- la oprirea recursivității/iterației returnăm valoarea acumulată în  $r$