

Programarea calculatoarelor

Decizia. Variabile și atribuirea. Iterația

Marius Minea

20 martie 2007

Programarea calculatoarelor. Curs 4

Marius Minea

Programarea calculatoarelor. Decizia. Variabile și atribuirea. Iterația

Secvențierea pentru expresii

3

După citirea unui număr vrem ca următorul caracter (care nu e cifră) să nu fie consumat din intrare, pentru a fi folosit ulterior. Ne trebuie: – o funcție standard (de bibliotecă) care să "pună înapoi" caracterul – cum să adăugăm asta la codul funcției

```
int ungetc(int c); // declarată în stdio.h
pune înapoi caracterul cu valoarea c în intrarea standard
(va fi returnat de următorul apel de citire, de ex. getchar())
```

Operatorul , (de secvențiere) `expr1 , expr2`
– evaluează `expr1`, ignoră rezultatul ei, dă ca rezultat valoarea lui `expr2`
– are precedență mică ⇒ grupăm toată expresia în paranteze

```
unsigned readnat_rc(unsigned r, int c)
{
    return isdigit(c) ? readnat_rc(r*10 + (c-'0'), getchar())
        : (ungetc(c, stdin), r);
} // stdin: identificator pentru intrarea standard
```

Programarea calculatoarelor. Curs 4

Marius Minea

Programarea calculatoarelor. Decizia. Variabile și atribuirea. Iterația

Exemple pentru instrucțiunea if

5

Tipărirea recursivă a unui număr natural:

```
#include <stdio.h>
void printnat(unsigned n) {
    if (n > 9)
        printnat(n/10); // tipareste si prima parte
    putchar('0' + n % 10); // oricum, tipareste si 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");
}
```

Putem rescrie `int abs(int x) { return x > 0 ? x : -x; }` cu `if`:

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

Programarea calculatoarelor. Curs 4

Marius Minea

Decizia și secvențierea

– construcții fundamentale în scrierea programelor (+ recursivitatea)
– am folosit: *decizia* în expresii (`? :`) și secvențierea instrucțiunilor

Mai multe instrucțiuni pot forma o singură *instrucțiune compusă*:

```
{
    instrucțiune                printf("scriu ceva\n");
    ...                          functia_mea(3, 5);
    instrucțiune                printf("scriu alta\n")
}
```

poate apărea oriunde sintaxa cere o instrucțiune (în corpul unei funcții)
Un exemplu e de instrucțiune compusă (*bloc*) e chiar corpul unei funcții.

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

Discutăm: cum putem evalua două expresii una după alta (secvențiere) și cum executăm instrucțiuni diferite în funcție de o condiție (decizie)

Programarea calculatoarelor. Curs 4

Marius Minea

Programarea calculatoarelor. Decizia. Variabile și atribuirea. Iterația

Instrucțiunea condițională (if)

4

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

– dacă expresia e adevărată se execută *instrucțiune1*, altfel se execută *instrucțiune2* (resp. nimic, în varianta scurtă)

– fiecare ramură are o *singură* instrucțiune (care poate fi compusă { })
– expresia trebuie să fie de tip scalar (întreg, real, enumerare)

Obs. În C, operatorii de comparație (`==`, `!=`, `<`, etc) întorc valorile *întregi* 1 (pentru adevărat) sau 0 (pentru fals)

o valoare se consideră *adevărată* dacă e nenulă și falsă dacă e nulă (atunci când e folosită ca și condiție: `în ? :, if, etc.`)

– o ramură `else` aparține întotdeauna de cel mai apropiat `if`

Programarea calculatoarelor. Curs 4

Marius Minea

Programarea calculatoarelor. Decizia. Variabile și atribuirea. Iterația

Operatori logici

6

Uneori apar decizii cu condiții compuse (chiar când există doar două variante de răspuns). Putem scrie programul mai simplu, fără a separa explicit toate ramurile de decizie, folosind direct operatorii logici: Un

an e bisect dacă: $\frac{an}{4} \in \mathbb{Z}$ și $\frac{an}{100} \notin \mathbb{Z}$ sau $\frac{an}{400} \in \mathbb{Z}$

```
int e_bisect(unsigned an)
{
    return an % 4 == 0 && (!(an % 100 == 0) || an % 400 == 0);
}
// se putea scrie și (an % 100 != 0)
```

Tabelele de adevăr pentru cei trei operatori sunt:

$expr$	$!expr$	$e1 \ \&\& \ e2$	$e1 \ \ e2$
0	1	0	0
1	0	1	1
0	1	0	1
1	0	1	0

a) negație b) conjuncție c) disjuncție

Programarea calculatoarelor. Curs 4

Marius Minea

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

- C nu are tip boolean; se folosește `int` (C99: `_Bool`, `stdbool.h`)
 - operatorii logici produc 1 pt. `true`, 0 pt. `false`
 - un întreg e interpretat ca `true` dacă e $\neq 0$ și ca `false` dacă e 0
- Operatorii relaționali:** precedența mai mică decât cei aritmetici
 - $x < y + 1$ înseamnă în mod natural $x < (y + 1)$
 - precedența: întâi `>`, `>=`, `<`, `<=`, apoi `==`, `!=` (egal, diferit)
- Operatorii logici** binari: `&&` (ȘI), prioritar lui `||` (SAU)
 - 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 t (dacă primul argument al lui `&&` (resp. `||`) e fals (resp. adevărat t)
 - Exemplu: `if (p != 0 && n % p == 0) { /* nu împarte la 0 */ }`
- Operatorul logic** unar `!` (negație logică)
 - cea mai ridicată prioritate (ca și toți operatorii unari)
 - transformă operand non-zero în 0, și zero în 1
 - Ex: `if (!gasit)` e echivalent cu `if (gasit == 0)`

Declararea variabilelor

- Când rezolvăm o problemă (scriem o funcție), deosebim: ce se dă (parametrii) și ce se cere (rezultatul). Uneori, e nevoie de rezultate/valori intermediare ⇒ *declaram variabile*
- Ex: în funcțiile de citire de numere, am transmis caracterul curent `c`, care nu face parte din enunțul problemei ⇒ funcția și-I poate citi singur:
- ```
unsigned readnat_r(unsigned r) { // declaram variabila c
 int c = getchar(); // pentru a retine rezultatul lui getchar
 if (isdigit(c)) return readnat_r(10*r+c-'0');
 else { ungetc(c, stdin); return r; }
}
```
- O **variabilă** e un obiect cu un *nume* și un *tip*. Se folosește la memorarea unor valori (altele decât parametrii de funcție) necesare în calcule.
- Declarația de variabile:** una sau mai multe variabile de același tip,, ex: `double x; int a = 1, b, c;` (a e inițializat cu 1, retul nu)
- Declaram variabile când e nevoie să reținem rezultate (de exemplu returnate de funcții) pentru folosire ulterioară.

## Despre variabile

- Un program C e o colecție de funcții ⇒ e scris *modular*: fiecare funcție rezolvă o subproblemă; programul principal `main` le apelează/combină.
- Numele *parametrilor* unor funcții diferite *nu* se influențează; ca și în matematică putem avea  $f(x) = \dots$  și  $g(x) = \dots$
- ⇒ la fel pentru variabilele declarate în funcții (*variabile locale*)
- Domeniul de vizibilitate** al unui identificator (de ex. variabilă) = partea de program unde poate fi utilizat (înțelesul său e cunoscut).
- Parametrii și variabilele declarate în funcții au domeniul de vizibilitate corpul funcției ⇒ *nu* sunt vizibile în exteriorul funcției.
- Variabilele locale au *durată de memorare* automată: sunt create la fiecare apel al funcției și distruse la încheierea acestuia (între apeluri nu există și deci nu își păstrează valoarea).
- Corpul `{ }` unei funcții C conține o *secvență de declarații și instrucțiuni*
- în C99, declarațiile și instrucțiunile pot apărea în orice ordine
  - în standardele anterioare: întâi declarații, apoi instrucțiuni

## Iterația. Ciclul cu test inițial

- Sintaxa: `while ( expresie )  
instrucțiune`
- 
- Semantica: se 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
- ATENȚIE !** Parantezele `( )` în jurul `( expresiei )` sunt obligatorii !
- Obs: Iterația și recursivitatea sunt strâns legate. Putem rescrie recursiv definiția iterației înlocuind (2) cu "se execută instrucțiunea `while`"

## Atribuirea

- O iterație e corectă dacă se oprește la un moment dat ⇒ condiția trebuie să devină falsă ⇒ să se *modifice*
- ⇒ condiția trebuie să conțină o funcție cu efect lateral, ex. citire, `c = getchar()`, sau o *variabilă modificată* în ciclu, ex. `n = n - 1`
- Atribuirea** = operația prin care se modifică valoarea unei variabile
- Sintaxa: `variabilă = expresie` (se evaluează expresia; se atribuie variabilei; aceasta e și valoarea întregii expresii de atribuire)
- poate fi folosită în alte expresii: `if ((c = getchar()) != EOF) ...` inclusiv atribuire în lanț `a = b = x + 3` (a și b primesc aceeași valoare)
- Doar prin atribuire* putem modifica o variabilă, *nu* prin simpla scriere de alte expresii sau transmiterea ca parametru la funcții!
- `n + 1` `sqr(x)` `toupper(c)` NU modifică nimic!

## 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 = 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 ( $x$ )
- testul de oprire și valoarea inițială pentru rezultat rămân aceleași
- varianta recursivă are propria valoare a parametrilor calculată la fiecare apel (în funcție de cea precedentă): ex.  $n * r, n - 1$ , etc.
- în varianta iterativă, valorile variabilelor sunt *actualizate* la fiecare iterație, după aceleași relații (ex.  $r = n * r, n = n - 1, r = x * r$ )
- în ambele cazuri se returnează valoarea acumulată a rezultatului

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

## 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

## 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 compusi 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).

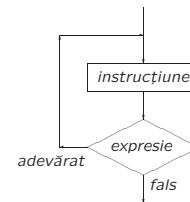
Ex. INCORECT: i = i++ (două atribuirii în aceeași expresie: = și ++)

**ATENȚIE** Atribuire 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);

## 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
```

## Instrucțiunea break

- produce ieșirea 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) {
 int c = getchar();
 unsigned nrw = 0;
 while (1) { // condiție adevărată, ciclu infinit
 while (isspace(c)) c = getchar();
 if (c == EOF) break; // se iese din ciclu
 nrw = nrw + 1;
 do c = getchar(); while (c != EOF && !isspace(c));
 }
 printf("%u\n", nrw);
 return 0;
}
```

## Instrucțiunea for

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

e echivalentă\* cu:

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

\* 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)