Programarea calculatoarelor

Decizia. Atribuirea. Iterația

17 martie 2009

Secvențierea

= construcție fundamentală în programare (cu decizia și recursivitatea)
Într-o funcție, instrucțiunile se execută una după alta (*secvențial*)

Instrucțiunea compusă: mai multe instrucțiuni grupate cu acolade { }
– poate conține și declarații: oriunde (C99)/doar la început (ANSI C)
– considerată *o singură instrucțiune* ⇒ putem folosi oriunde e nevoie
Un exemplu e de instrucțiune compusă (*bloc*) e chiar corpul unei funcții.

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

Cu *operatorul condițional* `?` `:` alegem din două *valori de expresii*
Cu *instrucțiunea condițională* alegem din două *instrucțiuni* de executat

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

Dacă expresia e adevărată se execută *instrucțiune1*, altfel se execută *instrucțiune2* (respectiv nimic, în varianta scurtă)
Fiecare ramură are *o singură* instrucțiune. Ea *poate fi compusă* { }
Expresia trebuie să fie de tip *scalar* (întreg, real, enumerare)

O valoare se consideră *adevărată* dacă e *nenulă* și *falsă* dacă e *nulă* (atunci când e folosită ca și condiție: în `?` `:` , `if` , `while` etc.)
Corespunzător: *Operatorii de comparație* (`==` `!=` `<` etc.) întorc în C valorile *întregi* 1 (pentru *adevărat*) sau 0 (pentru *fals*)

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 recursivă a unui număr natural:

```
#include <stdio.h>
void printnat(unsigned n) {
  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");
}
```

Putem rescrie (mai puțin concis) operatorul condițional `?` `:` cu `if`

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


Operatori logici

Chiar în deciziile cu doar două răspunsuri, condițiile pot fi compuse. Cu operatorii logici, putem scrie totul cu doar două ramuri de decizie:

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

C nu are tip boolean; se folosește `int` (C99: `_Bool` din `stdbool.h`)
– operatorii logici produc 1 pt. *true*, 0 pt. *false*
– un întreg e interpretat ca *true* dacă e ≠ 0 și ca *false* dacă e 0

expr	! *expr*
0	1
≠ 0	0

a) negație ! NU

e1	*e2* 0	*e2* ≠ 0
0	0	0
≠ 0	0	1

b) conjuncție && ȘI

e1	*e2* 0	*e2* ≠ 0
0	0	1
≠ 0	1	1

c) disjuncție || SAU

Precedența și evaluarea în scurt-circuit

Operatorul logic unar ! (negație logică): precedență cea mai ridicată
Ex: `if (!gasit)` e echivalent cu `if (gasit == 0)` (nul e fals)
Ex: `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)`
Evaluare: de la stânga la dreapta. *Evaluarea se oprește* (*scurt-circuit*) când rezultatul se știe din primul argument (fals la && , adev. la ||)

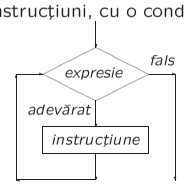
```
if (p != 0 && n % p == 0)      if (p != 0)         // doar atunci fa
  printf("p e divizor");         if (n % p == 0)   // si testul 2
                                   printf("p e divizor");
```

Iteraţia. Ciclul cu test iniţial

Am scris funcţii recursive ca să *repetăm* prelucrări – ceva esenţial.
Adesea, putem controla direct repetiţia unei instrucţiuni, cu o condiţie:

Sintaxa:

`while (` *expresie* `)`
instrucţiune



ATENŢIE! Parantezele `( )` sunt obligatorii la expresie!

Semantica: se exaluează expresia. Dacă e adevărată (nenulă):
- (1) se execută instrucţiunea (*corpul* ciclului)
- (2) se revine la începutul lui `while` (evaluarea expresiei)

Altfel (dacă condiţia e falsă/nulă) nu se execută nimic.
⇒ corpul se execută repetat *atât timp* cât condiţia e adevărată

Putem defini iteraţia recursiv. Pct. (2) = "execută instrucţiunea `while`"

Atribuirea

Iteraţia se opreşte când condiţia devine falsă ⇒ trebuie să se *modifice*
În *recursivitate* fiecare apel creează *noi copii* de parametri cu *alte valori*
În iteraţie, nu se mai creează altă variabilă la fiecare ciclu
⇒ trebuie să modificăm *valoarea unei variabile* (din condiţie)

Sintaxa: *variabilă* `=` *expresie* Totul e o *expresie (de atribuire)*.
Se evaluează expresia; valoarea se atribuie variabilei (şi e valoarea întregii expresii). Ex. `c = getchar()` `n = n-1` `r = r * n`

Poate fi folosită în alte expresii: `if ((c = getchar()) != EOF) ...`
inclusiv atribuire în lanţ `a = b = x + 3` (`a` şi `b` primesc aceeaşi valoare)

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

O variabilă *se poate modifica doar prin atribuire*, *nu* prin transmiterea ca parametru la funcţii, sau prin alte expresii!
`n + 1` `sqr(x)` `toupper(c)` calculează ceva, NU modifică nimic!
ATENŢIE! `=` operatorul de atribuire `==` operatorul de comparare.

Rescrierea recursivităţii ca iteraţie

```
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 recursivităţii ca iteraţie

– se face mai direct dacă funcţia e *recursivă la dreapta*: e scrisă cu acumularea rezultatului parţial, transmis mai departe ca parametru (`r`)
– testul de oprire şi valoarea iniţială pentru rezultat rămân aceleaşi
– în varianta recursivă, fiecare apel creează *copii noi* de parametri, cu valori proprii (în funcţie de cele vechi): ex. `n * r`, `n - 1`, `x * r`, etc.
– varianta iterativă, *actualizează (atribuie)* la fiecare iteraţie valorile variabilelor, după aceleaşi relaţii. Ex. `r = n * r, n = n - 1, r = x * r`
– ambele variante returnează valoarea acumulată a rezultatului

ATENŢIE: recursivitatea şi iteraţia produc ambele prelucrări *repetate*.
⇒ în probleme simple folosim una sau cealaltă, rareori amândouă!

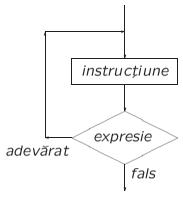
Citirea iterativă a unui număr, cifră cu cifră

```
#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()`

Ciclul cu test final

`do`
instrucţiune
`while (` *expresie* `);`



– uneori ştim sigur că un ciclu trebuie executat cel puţin o dată (citim cel puţin un caracter, un număr are măcar o cifră, etc.)
– ca şi ciclul cu test iniţial, execută *instrucţiune* atât timp cât execuţia expresiei e nenulă (adevărată)
– expresia se evaluează însă *după* fiecare iteraţie

– echivalent cu:

instrucţiune
`while (` *expresie* `)`
instrucţiune

Citirea caracter cu caracter: filtre

Frecvent: prelucrăm intrarea şi extragem / calculăm ceva.

```
void skipspace(void) {
  int c;
  while (isspace(c = getchar()));
  ungetc(c, stdin);
}
```

```
void skipspace(void) {
  int c;
  do
    c = getchar();
  while (isspace(c));
  ungetc(c, stdin);
}
```

Ciclul are corpul ; (instrucţiunea *vidă*)
ATENŢIE! Nu puneţi ; din greşeală!

```
int wordlen(void) {         // lungimea unui cuvânt citit
  int c, l = 0;
  while ((c = getchar()) != EOF && !isspace(c)) l++;
  return l;
}
```

ATENŢIE: Testaţi întotdeauna sfârşitul intrării, poate apărea oricând! Fără acest test, ciclul *s-ar bloca* când `c` e `EOF` (care nu e spaţiu)

Operatori de atribuire

ATENŢIE: Nu greşiţi folosind atribuirea în loc de test de egalitate!!
`if (x = y)` testează dacă valoarea lui `y` (atribuită şi lui `x`) e nenulă.

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

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 acelaşi *efect lateral* (atribuirea) dar *valoare* diferită

```
int x=2, y, z; y = x++; /* y=2,x=3 */; z = ++x; /* x=4,z=4 */
```

ATENŢIE Evitaţi expresii compuse cu mai multe efecte laterale!
(nu e precizat care se execută întâi).
Ex. INCORECT: `i = i++` (două atribuiri în aceeaşi expresie: `=` şi `++`)

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

Instrucţiunea `break`

– produce ieşirea 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) {          // condiţie adevărată, iese doar cu break;
    while (isspace(c = getchar()));           // consumă spaţiile
    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 (;;) {             // condiţie adevărată, iese doar cu break;
    while (isspace(c = getchar()))       // cât timp citeşte spaţii
      putchar(c);                           // se scriu şi spaţiile
    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 spaţiu iese
    }                                         // şi reia ciclul for
  }
  return 0;
}
```


Instrucţiunea `for`

`for (`*expr-init* ; *expr-test* ; *expr-actualiz*`)`
  instrucţiune

e echivalentă* cu:

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

* excepţie: instrucţiunea `continue`, vezi ulterior

– oricare din cele 3 expresii poate lipsi (dar cele două ; rămân)
– dacă *expr-test* lipseşte, e tot timpul adevărată (ciclu infinit)

În C99 în loc de *expr-init* e permisă o *declaraţie* de variabile (iniţializate) cu domeniu de vizibilitate întreaga instrucţiune (dar nu şi după)

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

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

ATENŢIE Instrucţiunea ; e caz particular al instrucţiunii *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` şi cu *expr-actualiz*)

```
while (isspace(c = getchar()));   (consumă secvenţă de spaţii)
```


Scrierea ciclurilor

În conceperea programelor care conţin cicluri
– identificăm ce variabilă se modifică în fiecare iteraţie
– identificăm care e condiţia de oprire
– nu uităm instrucţiunea care modifică acea variabilă
(altfel ciclul continuă la infinit)

Definim precis ce ştim despre program când iese dintr-un ciclu.
– la ieşirea dintr-un ciclu, condiţia e falsă
⇒ ne spune ceva despre valorile posibile ale variabilelor din condiţie
Folosim această informaţie 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, şi pentru situaţii limită