Decizia. Iteraţia

Marius Minea

21 martie 2006

Variabile locale

Un program C e o colecţie de funcţii ⇒ e scris *modular*: fiecare funcţie rezolvă o subproblemă, şi programul principal (`main`) le combină.

Numele *parametrilor* unor funcţii diferite *nu* se influenţează
ca şi în matematică putem avea $f(x) = \ldots$ şi $g(x) = \ldots$
⇒ la fel pentru *variabilele* declarate în funcţii (variabile locale)

Domeniul de vizibilitate al unui identificator = fragmentul de program unde poate fi utilizat (înţelesul său e cunoscut).
Variabilele declarate în funcţii au domeniul de vizibilitate corpul funcţiei (din punctul declaraţiei până la acolada de sfârşit).
Parametrii unei funcţii au domeniul de vizibilitate tot corpul funcţiei
⇒ *nu* declarăm variabile cu acelaşi nume, ar fi un conflict

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

Recapitulare: instrucţiuni şi secvenţiere

O *instrucţiune* e un element de limbaj ce specifică o *acţiune* care să fie executată de program. Am discutat:
– instrucţiunea `return`: `return` *expresie* `;`
evaluează expresia, valoarea ei devenind valoarea funcţiei şi *încheie* execuţia funcţiei (ea *returnează* valoarea calculată)
– instrucţiunea *expresie*: *expresie* `;`
evaluează expresia; esenţiale sunt *efectele laterale*. Exemple:
`printf("salut");` valoarea (nefolosită) e numărul de caractere tipărite
`x = 2 + 3;` se evaluează 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 instrucţiune corectă, dar fără efect (valoarea nu e folosită)

Corpul unei funcţii poate conţine o *secvenţă* de mai multe instrucţiuni; ele se execută *secvenţial*, în ordinea dată de program (excepţii ulterior)
Secvenţierea e elementul de bază prin care scriem programe complexe

Instrucţiunea compusă

E util să putem trata compunerea a mai multe elemente de limbaj ca un singur element de acelaşi fel.
(am văzut deja că 2 şi `n+1` sunt expresii, la fel şi `2 * (n+1)`)
O *instrucţiune* e o *acţiune* de executat de către program
Putem grupa mai mai multe instrucţiuni între acolade într-o *instrucţiune compusă* (*bloc*):
instructiune-compusa ::= {
instrucţiune
...
instrucţiune
}
Mai general, un bloc poate conţine o secvenţă de *declaraţii* şi *instrucţiuni*

Exemplu: tipărirea cifră cu cifră a unui număr

Varianta 1: dorim să şi numărăm câte cifre are

```
#include <stdio.h>
int printint(unsigned n) // tipăreşte şi 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);          // tipăreşte 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 funcţii care *nu* întorc un rezultat, şi sunt utile doar pentru efectul lateral (de exemplu o tipărire).
Tipul returnat de funcţie e tipul `void` (tipul vid, fără valori)
– nu e necesar `return`, execuţia se încheie la acolada `}`
(se *poate* folosi `return;` pentru a termina explicit execuţia)

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 condiţional*), ci *acţiuni* diferite (adică *instrucţiuni*)

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`