Variabile. Efecte laterale

Marius Minea

14 martie 2006

Recursivitate: putere cu înjumătățirea exponentului

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{n/2})^2 & n \text{ par} \\ x \cdot (x^{n-1/2})^2 & n \text{ impar} \end{cases}$$

```
double sqr(double x) { return x*x; }
double pow2(double x, int n)
{
  return n == 0 ? 1 :
    n % 2 == 0 ? sqr(pow2(x, n/2)) :
    x * sqr(pow2(x, n/2));
}
```


Structura apelurilor recursive

Corpul oricărei funcții poate conține o *secvență* de instrucțiuni, care se execută succesiv
– prin inserarea de tipăriri (`printf`) vizualizăm ordinea de execuție

```
double sqr(double x) { return x*x; }
double pow2(double x, unsigned n)
{
  printf("exponent %d\n", n);
  return n == 0 ? 1 :
    n % 2 == 0 ? sqr(pow2(x, n/2)) :
    x * sqr(pow2(x, n/2));
}
```

Pentru exponent n = 19 avem:

```
exponent 21
exponent 10
exponent 5
exponent 2
exponent 1
exponent 0
```


Apeluri și calcule repetate

Dacă am fi scris în schimb:

```
double pow2(double x, unsigned n)
{
  printf("exponent %d\n", n);
  return n == 0 ? 1 :
    n % 2 == 0 ? pow2(x, n/2) * pow2(x, n/2) :
    x * pow2(x, n/2) * pow2(x, n/2);
}
```

Cu exponent n = 3:

```
exponent 3
exponent 1
exponent 0
exponent 0
exponent 1
exponent 0
exponent 0
```

Cele două expresii `pow2(x, n/2)` se evaluează succesiv, independent. Compilatorul nu caută sa vadă că sunt egale pt. a nu recalcula
În cazul anterior, pentru expresia de forma `y * y` s-a scris o *funcție*
Expresia `pow2(x, n/2)` se evaluează o singură dată, ca argument al funcției `sqr`, înainte de a o apela pe aceasta.

Folosirea functiilor declarate in fisiere antet

Arbore binar = frunză sau rădăcină cu doi subarbori.
Ni se dă un fișier `arbore.h` cu *declarații* de funcții pt. arbori
Funcțiile sunt implementate în alt fișier de tip `.c` (nu-l discutăm) (la fel, funcția `printf` e *declarată* în `stdio.h` dar implementată altundeva, într-o bibliotecă)

```
typedef struct a *arbore;       // defineste tipul arbore
// discutam mai tarziu astfel de definitii de tip
arbore a_l(arbore a);   // returneaza arborele stang
arbore a_r(arbore a);   // returneaza arborele drept
int a_val(arbore a);    // returneaza intregul din radacina
arbore c_arb(int n, arbore l, arbore r);    // creeaza arbore
                        // cu radacina data si doi subarbori
arbore c_frunza(int n); // creeaza o frunza cu intregul dat
int e_frunza(arbore a); // adevarat daca arborele e frunza
// adevarat in C inseamna o valoare nenula
```


Recursivitate: arbore binar

Folosim funcțiile din `arbore.h` tot cu directiva `#include`
(sintaxa cu ghilimele caută uzual și în directorul curent)
Programul trebuie compilat impreuna cu codul pentru funcțiile folosite (aflat într-un fișier C (de ex. `arbore.c`) sau într-o bibliotecă)

```
#include "arbore.h"
#include <stdio.h>
int a_sum(arbore a)  // suma tuturor numerelor
{
  return e_frunza(a) ? a_val(a) :
         a_val(a) + a_sum(a_l(a)) + a_sum(a_r(a));
}
int main(void) {
  printf("%d\n", a_sum(c_arb(5,
                 c_arb(3, c_frunza(2), c_frunza(7)),
                 c_frunza(-4))));
  return 0;
}
```

Caractere. Codul ASCII

ASCII = American Standard Code for Information Interchange
Caracterele sunt memorate ca și cod numeric = indicele în acest tabel
ex. `0` are codul 48, `A` are codul 65, `a` are codul 97, etc.

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
-------------------------------------------------------
0x0    \0                  \a \b \t \n \v \f \r
0x10:
0x20:     !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
0x30:  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
0x40:  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
0x50:  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
0x60:  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
0x70:  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

Am scris cu prefixul `0x` constante hexazecimale (în baza 16).
– caracterele cu cod $<$ 0x20 (spațiu): caractere de control
– caracterele cu cod $>$ 0x7f (127): nu fac parte din setul ASCII (diacritice, diverse variante nationale standardizate de ISO, etc.)

Tipul caracter în C

Tipul standard `char` reprezintă caractere (codul lor ASCII – un întreg)
⇒ în C, tipul `char` e un *tip întreg*, dar cu domeniu de valori mai restrâns decât `int` sau `unsigned`

Există tipul `signed char`, cu valori de la -128 la 127
și tipul `unsigned char`, cu valori de la 0 la 255.
Standardul permite tipului `char` să fie oricare din cele două
(caracterele din setul ASCII sunt codificate la fel în oricare variantă)

În program, *constantele caracter* se scriu între ghilimele simple `' '`
Ele au valori întregi (= cu codul ASCII). Se permit operațiile pe întregi:
`'7' == '0' + 7` `'E' - 'A' == 4` `'f' == 'a' + 5`

Reprezentări în program pentru caractere speciale:

`'\0'`	null	`'\n'`	linie nouă
`'\a'`	alarm	`'\r'`	carriage return
`'\b'`	backspace	`'\f'`	form feed
`'\t'`	tab	`'\''`	apostrof (ghilimea)
`'\v'`	vertical tab	`'\\'`	backslash

Citirea și scrierea unui caracter

`int getchar(void)`
– funcție fără parametri, returnează valoarea caracterului (codul ASCII), ca și `unsigned char` convertit la `int`
– returnează valoarea specială `EOF` (nu face parte din tipul `unsigned char`) dacă nu s-a putut citi un caracter (la sfârșit de fișier)

IMPORTANT! Programul nu are control asupra datelor introduse la citire ⇒ trebuie întotdeauna verificat ce s-a citit/testate erorile

`int putchar(int c)`
– scrie un `unsigned char` dat ca și `int`; returnează valoarea scrisă

```
#include <stdio.h>
int main(void) {
  putchar(':');       // scrie caracterul :
  putchar(getchar()); // citeşte şi scrie un caracter
  putchar(getchar()); // citeşte şi scrie inca unul
  return 0;
}
```


Noțiunea de efect lateral

Apelul repetat al unei funcții (în matematică, sau din cele scrise până acum: `sqr`, `fact`, etc.) cu aceiași parametri produce același rezultat.

La fiecare apel al lui `getchar()` se returnează *alt* caracter din intrare; cel anterior a fost consumat.
– dacă dorim sa facem mai multe operații cu valoarea citită, devine necesară *memorarea ei*

O modificare în starea mediului de execuție a programului se numește *efect lateral* (ex. citire, scriere; atribuire (discutăm ulterior))

Noțiunea de variabilă. Atribuirea

O variabilă e un obiect caracterizat prin *tip* și *valoare*.
Valoarea unei variabile poate fi modificată prin *atribuire*.
O variabilă își păstrează valoarea până la următoarea atribuire.

Declararea variabilelor în C
– cel mai simplu: *tip nume_variabilă* ; `double x;`
– mai multe variabile de același tip: `int a, b, c;`
– cu inițializare: `int n = 2, x; // n initializat, x nu`

Corpul { } unei funcții în C conține o *secvență de declarații și instrucțiuni*
– în C99, pot aparea în orice ordine
– în standardele anterioare, declarațiile preced instrucțiunile

Exemplu: citirea unui intreg

```
#include <ctype.h>
#include <stdio.h>

int readint(int v)      // v: valoarea citita pana acum
{
  int c;
  c = getchar();       // citim urmatorul caracter
  return isdigit(c) ? readint(10*v + c - '0') : v;
  // apelam cu valoarea curenta actualizata
  // sau la incheiere (nu mai e cifra), returnam pe cea curenta
}

int main(void)
{
  printf("%d\n", readint(0));   // initial, valoare curenta 0
  return 0;
}
```