Programarea calculatoarelor

Introducere

Marius Minea

28 februarie 2006

Despre limbajul C

– dezvoltat și implementat în 1972 la AT&T Bell Laboratories de Dennis Ritchie `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`
– nevoia unui limbaj pentru scrierea de sisteme de operare și utilitare (strâns legat de *sistemul de operare UNIX* dezvoltat la Bell Labs):
– C dezvoltat inițial sub UNIX; în 1973, UNIX rescris în totalitate în C
– cartea de referință: Brian Kernighan, Dennis Ritchie:
The C Programming Language (1978)
– în 1988 (vezi K&R ediția II) limbajul a fost standardizat de ANSI (American National Standards Institute)
– versiunea curentă: C99 (standard ISO 9899)

De ce folosim C ?

– foarte versatil: acces la reprezentarea binară a datelor, mare libertate în lucrul cu memoria, bună interfață cu hardware
– limbaj matur, bază mare de cod (biblioteci pt. multe scopuri),
– compilatoare bune, generează cod eficient (compact, rapid)

Calcule, funcţii si programe

Primul rol al programelor: de a efectua *calcule* (matematice)

În matematică, efectuăm calcule cu ajutorul *funcţiilor*:

- *cunoaştem* diverse funcţii (sin, cos, etc.)
- *definim* funcţii noi (depinzând de problemă)
- *combinăm* funcţiile existente şi definite de noi
- şi le *folosim* într-o anumită *ordine*

Toate aceste aspecte le întâlnim şi în programare

Funcţii în matematică şi în C

Exemplu: funcţia de ridicare la pătrat

$$sqr : \mathbb{Z} \to \mathbb{Z}$$

$$sqr(x) = x \cdot x$$

```
int sqr(int x)
{
  return x * x;
}
```

– *antetul* funcţiei: specifică domeniul de valori (întregi), numele funcţiei şi parametrii acesteia (un singur parametru, intreg)
– *corpul* funcţiei: aici, o singură *instrucţiune* (`return`): indică o *expresie* care dă valoarea funcţiei (pornind de la parametri)

Limbajul are *reguli* precise de scriere (*sintaxa*):
– diversele elemente scrise într-o anumită *ordine*;
– se folosesc *separatori* pentru a le delimita precis: `( )` `;` `{ }`

O a doua funcție

Ridicarea la pătrat pentru numere *reale*

$$sqr2 : \mathbb{R} \to \mathbb{R}$$

$$sqr2(x) = x \cdot x$$

```
float sqrf(float x)
{
  return x * x;
}
```

– o altă funcție decât cea dinainte: alt domeniu de definiție și de valori
– trebuie să-i dăm alt nume dacă o folosim in același program
– strict vorbind, și operația $\hat{*}$ e alta, fiind definită pe altă mulțime

Cuvintele `int`, `float` denotă *tipuri*.

Un *tip* e o *mulțime de valori* împreună cu un set de *operații* permise pentru aceste valori.

Întregi, reali și operații matematice

Există *diferențe importante* între tipurile numerice în limbajul C și corespondentul lor ideal, matematic. În matematică, $\mathbb{Z} \subseteq \mathbb{R}$. În C:

– modul de scriere a unei constante îi determină tipul:

  `2` e un întreg, `2.0` e un real

  putem scrie un real și în notație științifică: `1.0e-3` în loc de `0.001`

  sunt echivalente scrierile: `1.0` și `1.` respectiv `0.1` și `.1`

– unele operații sunt definite diferit pentru întregi și reali:

  Împărțirea întreagă e *împărtire cu rest* !!!

`7 / 2` dă valoarea 3, pe când `7.0 / 2.0` dă valoarea `3.5`

`-7 / 2` dă valoarea -3, deci la fel cu `- (7 / 2)`

Operatorul *modulo* (scris %) e permis doar pentru întregi.

`9 / 5` este `1` `9 % 5` este `4` `9 / -5` este `-1` `9 % -5` este `4`

`-9 / 5` este `-1` `-9 % 5` este `-4` `-9 / -5` este `1` `-9 % -5` este `-4`

semnul restului e același cu semnul deîmpărțitului

e valabilă egalitatea `a == a / b * b + a % b`

Puţină terminologie

– *cuvinte cheie*: au un înţeles predefinit (nu poate fi schimbat) pentru instrucţiuni (ex. `return`), *tipuri* (ex. `int`), etc.

– *identificatori* (de ex. `sqr`, `x`) aleşi de programator pentru a denumi *funcţii*, *parametri*, *variabile*, etc.

Un identificator e o secvenţă de caractere formată din litere (mari şi mici), liniuţa de subliniere _ şi cifre, care nu începe cu o cifră şi nu este un cuvânt cheie.
Exemple: `x3, a12_34, _exit, main, printf, int16_t`

– *semne de punctuaţie*, cu diverse semnificaţii:
  * `*` e un *operator*
  * `;` delimitează sfârşitul unei instrucţiuni
  * parantezele `( )` grupează parametrii unei funcţii sau o subexpresie
  * acoladele `{ }` grupează instrucţiuni sau declaraţii etc.

Funcţii cu mai mulţi parametri

Exemplu: discriminantul ecuaţiei de gradul II $a \cdot x^2 + b \cdot x + c = 0$

```
float discrim(float a, float b, float c)
{
  return b * b - 4 * a * c;
}
```

– între parantezele rotunde ( ) din antetul funcţiei putem specifica oricâţi parametri, fiecare cu tipul propriu, separaţi prin virgulă.

Apelul de funcție

Până acum, am *definit* funcții, fără să le *folosim*.
Valoarea unei funcții poate fi folosită într-o expresie cu aceeași sintaxă ca și în matematică: $functie(parametru, parametru, \ldots, parametru)$
Exemplu: în discriminantul dinainte, puteam scrie:

```
return sqrf(b) - 4 * a * c;
```

Sau putem defini:

```
int cube(int x)
{
  return x * sqr(x);
}
```

IMPORTANT: înainte de a *folosi* orice identificator (nume) în C, el trebuie să fie *declarat* (trebuie să știm ce reprezintă)
⇒ Exemplele sunt corecte dacă `sqrf`, respectiv `sqr` sunt definite *înainte* de `discrim`, respectiv `cube` în program.

Un prim program C

```
int main(void)
{
  return 0;
}
```

– cel mai mic program: nu face nimic !
– orice program conţine funcţia *main* şi e executat prin apelarea ei
(programul poate conţine şi alte funcţii)
– în acest caz: funcţia nu are parametri (`void`)
Cf. standard: `main` returneaza un cod intreg catre sistemul de operare
(conventie: 0 == terminare cu succes, != 0: cod de eroare)

Un program comentat

```
/* Acesta este un comentariu */
int main(void) // comentariu până la capăt de linie
{
  /* Acesta e un comentariu pe mai multe linii
     obisnuit, aici vine codul programului */
  return 0;
}
```

– programele pot conţine *comentarii*, înscrise între /* şi */
sau începând cu // şi terminându-se la capătul liniei (ca în C++)
– orice conţinut între aceste caractere nu are nici un efect asupra generării codului şi execuţiei programului
– programele *trebuie* comentate
  – pentru ca un cititor sa le înţeleagă (alţii, sau noi, mai târziu)
  – ca documentaţie şi specificaţie: funcţionalitate, restricţii, etc.
  – ce reprezinta variabilele, parametrii functiilor, rezultatul, ce conditii sunt necesare, cum se comporta la eroare, etc.

Să scriem ceva!

```
#include <stdio.h>

int main(void)
{
  printf("hello, world!\n"); // tipăreste un text
  return 0;
}
```

– prima linie: obligatorie pentru orice program care citește sau scrie
= o *directivă de preprocesare*, include fișierul `stdio.h` care conține *declarațiile* (NU implementarea) funcțiilor standard de intrare/ieșire
= informațiile (nume, parametri) necesare compilatorului pt. a le folosi
– implementarea (cod obiect, compilat): într-o bibliotecă inclusă (linkeditată) la compilarea programului utilizator
– `printf` ("print formatted"): o *funcție standard*
– N.B.: `printf` *nu* este o instrucțiune sau cuvânt cheie
– e apelată aici cu un parametru șir de caractere
– șirurile de caractere: incluse între ghilimele duble `"`
– `\n` este notația pentru caracterul de linie nouă

Tipărirea unei valori numerice

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  printf("cos(0) = ");
  printf("%f", cos(0));
  return 0;
}
```

```
#include <stdio.h>
int sqr (int x) { return x * x; }
int main(void)
{
  printf("2 ori -3 la patrat e ");
  printf("%d", 2 * sqr(-3));
  return 0;
}
```

Pentru a tipări valoarea unei expresii, `printf` ia două argumente:

– un șir de caractere (specificator de format):
  `%d` (întreg), `%f` (real)

– expresia, al cărei tip trebuie să fie compatibil cu cel indicat
  (verificarea cade în sarcina programatorului !!!)

Funcţii definite pe cazuri

$abs : \mathbb{Z} \to \mathbb{Z}$ $\qquad abs(x) = \begin{cases} x & x \geq 0 \\ -x & \text{altfel } (x < 0) \end{cases}$

Cu cele discutate pâna acum, nu putem defini această funcţie în C.

Valoarea funcţiei nu e dată de o singură expresie, ci de una din doua expresii diferite (x sau $-x$), în funcţie de o *condiţie* ($x \geq 0$ sau nu)

$\Rightarrow$ e necesară o facilitate de limbaj pentru a *decide* valoarea pe care o ia o expresie în funcţie de valoarea unei condiţii (adevărat/fals)

Operatorul condiţional ? : în C

O *expresie condiţională* în C are sintaxa: *condiţie* ? *expr1* : *expr2*
– dacă condiţia e adevărată, se evaluează *expr1* şi întreaga expresie ia valoarea acesteia
– dacă condiţia e falsă, se evaluează *expr2* şi întreaga expresie ia valoarea acesteia

```
int abs(int x)
{
  return x >= 0 ? x : -x;
}
```

Operatori de comparaţie în C: == (egalitate), != (diferit), <, <=, >, >=

IMPORTANT! Testul de egalitate în C e == şi nu = simplu !!!

Obs.: Funcţia `abs` există ca funcţie standard, declarată în `stdlib.h`

Definirea funcţiilor pe mai multe cazuri

$sgn : \mathbb{Z} \to \{-1, 0, 1\}$ $\qquad$ $sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$

Chiar cu operatorul condiţional nu putem transcrie funcţia direct în C. (el permite doar decizia pe *două* ramuri (adevărat/fals), nu pe un număr mai mare de condiţii / ramuri)

$\Rightarrow$ trebuie să descompunem problema de calcul a lui sgn (de fapt decizia asupra valorii parametrului x)

– *descompunerea în subprobleme mai mici*: principiu *foarte important* în rezolvarea de probleme

Rescriem funcţia cu *o singură* decizie la fiecare moment:

$$sgn(x) = \begin{cases} \text{daca } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{daca } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

Definirea pe mai multe cazuri (cont.)

```
int sgn (int x)
{
  return x < 0 ? -1 :
         x == 0 ? 0 : 1;
}
```

– putem grupa arbitrar de mulți operatori ? :
– într-o expresie scrisă corect, un : corespunde univoc unui ?

Gruparea eficientă a testelor

Considerăm următoarea funcţie definită pe intervale:

$$f : \mathbb{R} \to \mathbb{R} \qquad f(x) = \begin{cases} -x^2 & x < -1 \\ x & -1 \le x < 0 \\ x^2 & 0 \le x < 1 \\ x^3 & 1 \le x \end{cases}$$

testând fiecare caz secvenţial

```
float f(float x) {
  return x < -1 ? -x * x
         : x < 0 ? x
         : x < 1 ? x * x
         : x * x * x;
}
```

despărţind în *subprobleme* cu număr cât mai echilibrat de cazuri

```
float f(float x) {
  return x < 0 ?
           x < -1 ? -x * x : x
         : x < 1 ? x*x : x*x*x;
}
```

varianta a doua: doar două teste, pentru orice valoare a lui `x`

$\Rightarrow$ *căutarea binară* e o modalitate eficientă