

Reprezentare și operatori pe biți

Marius Minea

4 aprilie 2006

Reprezentarea obiectelor în memorie

un *bit* = cea mai mică unitate de memorare, suficientă pentru două valori (identificate convențional cu 0 și 1)

un *octet* (byte) = grup de 8 biți, suficient pentru a memora un caracter – e cea mai mică unitate *adresabilă* direct (care se poate citi sau scrie independent din/în memorie)

Toate valorile sunt reprezentate în calculator pe număr finit de octeți. Operatorul `sizeof`: ne dă dimensiunea *în octeți* a unei expresii sau a unui tip: `sizeof(tip)` sau `sizeof expresie`
Exemplu `sizeof(char)` e 1: un caracter ocupă un octet

Reprezentarea binară a numerelor

În memoria calculatorului, numerele se reprezintă în binar (baza 2).

Valoarea unui *întreg fără semn*, cu k cifre binare (biți):
 $c_{k-1}c_{k-2}\dots c_1c_0 (2) = c_{k-1} * 2^{k-1} + \dots + c_1 * 2^1 + c_0 * 2^0$

c_{k-1} = bitul *cel mai semnificativ* (superior)

c_0 = bitul *cel mai puțin semnificativ* (inferior)

Exemple: $11111111 (2)$ e 255; c_0 e 0 \Rightarrow nr. par; c_0 e 1 \Rightarrow nr. impar

Întregi *cu semn*: reprezentați *în complement de 2*

dacă bitul superior e 1, nr. se consideră negativ

valoarea: translatată cu 2^k în jos față de interpretarea fără semn.

$1c_{k-2}\dots c_1c_0 (2) = -2^{k-1} + c_{k-2} * 2^{k-2} + \dots + c_1 * 2^1 + c_0 * 2^0$

Exemple (pe 8 biți): 11111111 e -1; 10000000 e -128

Numerale reale (altă reprezentare: semn, exponent, mantisă)

S EEEEEEE MMMMMMMMMMMMMMMMMMM (pt. float: 1+8+23 biți)

pt. $0 < E < 255$: $(-1)^S * 2^{E-127} * 1.M(2)$

plus alte cazuri pentru 0, $\pm\infty$, numere foarte mici, erori (NaN)

Operatori pe biți

– oferă acces direct la reprezentarea binară a datelor în memorie, cu posibilități apropiate limbajului de asamblare

– pot fi aplicați doar operanzilor de tipuri întregi, cu sau fără semn

- & ȘI bit cu bit (1 doar dacă ambii biți sunt 1)
- | SAU bit cu bit (1 dacă cel puțin un bit e 1)
- ^ SAU exclusiv bit cu bit (1 dacă exact un bit e 1)
- ~ complementare bit cu bit (1 devine 0, 0 devine 1)

<< deplasare la stânga cu număr indicat de biți
(se introduc la dreapta biți de 0, cei din stânga se pierd)

>> deplasare la dreapta cu număr indicat de biți
(se introduc la stânga biți de 0 dacă numărul e fără semn
altfel depinde de implementare (ex. se repetă bitul de semn))

Proprietăți ale operatorilor pe biți

$n \ll k$ are valoarea $n \cdot 2^k$ (dacă nu apare depășire)

$n \gg k$ are valoarea $n/2^k$ (împărțire întregă; pentru n fără semn)

Deci, $1 \ll k$ e 2^k (pentru exponenți mai mici de $8 * \text{sizeof(int)}$)

0 are toți bitii 0, ~ 0 e numărul cu toți biții 1

$\sim(1 \ll k)$ are doar bitul k pe 0, restul pe 1

& cu 1 păstrează valoarea unui bit, & cu 0 pune pe 0 valoarea unui bit

$n \& (1 \ll k)$ testează (e nenul) dacă bitul k din n e 1

$n \& \sim(1 \ll k)$ pune pe 0 bitul k în rezultat

| cu 0 păstrează valoarea unui bit, | cu 1 pune pe 1 valoarea unui bit

$n | (1 \ll k)$ pune pe 1 bitul k în rezultat

^ cu 0 păstrează valoarea unui bit, ^ cu 1 schimbă valoarea unui bit

$n \wedge (1 \ll k)$ schimbă valoarea bitului k în rezultat

Crearea și selectarea unor tipare de biți

– fie scriind constantele în hexazecimal sau octal, ex: 5 biți de 1:

0x1f sau 0x1f etc. (hexazecimal) sau 037 (octal, cu 0 în față)

– fie aplicând operatorii pe biți

întregul cu toți biții 1: ~ 0 (sau $\sim 0u$ pentru număr fără semn)

întregul cu k biți din dreapta 0, restul 1: $\sim 0 \ll k$

întregul cu k biți din dreapta 1, restul 0: $(1 \ll k) - 1$ sau $\sim(\sim 0 \ll k)$

$\sim(\sim 0 \ll k) \ll p$ are k biți pe 1, începând de la bitul p , și restul 0

Selectarea anumitor biți dintr-un număr

$(n \gg p) \& \sim(\sim 0 \ll k)$

deplasăm pe n cu p poziții la dreapta și păstrăm doar ultimii k biți

$n \& (\sim(\sim 0 \ll k) \ll p)$

păstrăm doar k biți din n , începând cu cel de ordin p

Constante întregi în baza 10: scrise obișnuit; ex. -5
 în baza 8: prefix zero; ex. 0177 în baza 16: prefix 0x sau 0X; ex. 0xA9
 sufix u sau U: unsigned, ex. 65535u sufix l sau L: long ex. 0177777L

Constante de tip caracter

– caractere tipăribile, între ghilimele simple: '0', '1', 'a'
 – caractere speciale: '\0' null '\t' tab
 '\b' backspace '\n' linie nouă '\r' carriage return
 '\'' apostrof (ghilimea) '\\' backslash
 – caractere scrise în octal (max. 3 cifre), ex: '\14'
 – caractere scrise în hexazecimal (prefix x), ex. '\xff'

Constante reale

– conțin mantisă, iar optional semn și exponent (prefix e sau E)
 – în mantisă, partea reală sau zecimală poate lipsi, dar nu amândouă
 – implicit, orice constantă reală e considerată double
 – sufix f sau F pentru float; l sau L pentru long double
 Exemple: 1.0 sau 1. sau .1e1 3.1415926 1.175494e-38f

În fața tipului `int` se pot pune cuvinte cheie care specifică:

– **dimensiunea**: `short`, `long` (în C99 și `long long`)
 – **semnul**: `signed` (implicit, dacă nu apare), `unsigned`
 Cele două se pot combina; `int` poate fi omis (ex. `unsigned short`)

Dimensiuni **minime** în octeți (`sizeof`), și domenii **minime** de valori:
`int`, `short`: ≥ 2 octeți; `long`: ≥ 4 octeți; `long long`: ≥ 8 octeți
 $[-2^{15}, 2^{15}-1]$ $[-2^{31}, 2^{31}-1]$ $[-2^{63}, 2^{63}-1]$
 cu **unsigned**: aceeași dimensiune; valori între 0 și $2^{8*\text{sizeof}(tip)} - 1$

Numerale reale: reprezentate cu semn, mantisă, și exponent
 \Rightarrow domeniu de valori simetric față de zero; precizie relativă la mărime
Exemple de dimensiuni (compilator gcc pe i386, sub Linux):
 – `float`: 4 octeți, între cca. 10^{-38} și 10^{38} , 6 cifre semnificative
 – `double`: 8 octeți, între cca. 10^{-308} și 10^{308} , 15 cifre semnificative
 – `long double`: 12 octeți (pentru precizie suplimentară)

Pentru **dimensiunea corectă**, folosim `sizeof(typ)`, nu 2, 4, 8, etc.

limits.h: valori minime cerute de standard
 SHRT_MIN, INT_MIN -32767 SHRT_MAX, INT_MAX 32768 USHRT_MIN, UINT_MIN 65535
 LONG_MIN -2147483647 LONG_MAX 2147483647 ULONG_MAX 4294967295

Obs: pe gcc/i386/Linux, `int` are aceleași dimensiuni ca și `long`
 Reținem: întregi cu semn pe 4 octeți: până la cca 2 miliarde

float.h: valori pt. gcc/i386/Linux (și cerințele standard)

FLT_DIG 6 DBL_DIG 15 // cifre precizie zecimala
 FLT_MIN 1.17e-38F FLT_MAX 3.4+38F // min/max în modul
 FLT_EPSILON 1.19e-07F DBL_EPSILON 2.22e-16 // nr.min.cu 1+eps>1
 DBL_MIN 2.22e-308 DBL_MAX 1.79e+308 // min/max în modul
 Reținem: `float`: cca $10^{\pm 30}$, 6 cifre; `double`: cca $10^{\pm 300}$, 15 cifre

Numere cu / fără semn: fiți consecvenți (probleme la conversie)
`unsigned char u = 255; signed char i = u;` dacă `i == -1` (aceiași biți)
 sau: `printf("%u", -1);` scrie 4294967295 (aceeași reprezentare)

Pentru a compara corect `unsigned u` cu `int i`:

`i >= 0 && i > u` în loc de `i > u` (întâi i trebuie să fie negativ)

Fiți consecvenți în folosirea tipurilor cu/fără semn !

La adunare/scădere pot apărea depășiri: `int i1, i2; unsigned u1, u2;`
`if (i1>0 && i2>0 && i1+i2<0 || i1<0 && i2<0 && i1+i2>0) //depasire`

– `int` (chiar `long`): domeniu de valori mic (cca ± 2 miliarde)
 – e insuficient pentru multe calcule cu întregi (ex. factorial mare)
 – folosim reali (`double`: domeniu de valori mare, dar precizie limitat, dincolo de $1E16$ tipul `double` nu mai distinge doi întregi consecutivi)

– o valoare citită de la intrare nu e reprezentată neapărat precis!
`float x; scanf("%f", &x); printf("%.7f", x);` 4.2 \rightarrow 4.1999998
 fracții exacte în baza 10 pot fi periodice în baza 2: $1.2 = 1.(0011)_2$

– în calcule matematice, apar pierderi de precizie pe parcurs
 \Rightarrow e mai robust să testăm `fabs(x - y) < epsilon` decât `x==y`
 – diferențe dincolo de limita preciziei nu se pot reprezenta:

\Rightarrow pentru `x < DBL_EPSILON` (resp. `FLT_EPSILON`) avem `1 + x == 1`

Atribuirea propriu-zisă: `var = expr` (un operator ca oricare altul)

\Rightarrow o expresie de atribuire poate fi folosită în altă expresie compusă
 (și valoarea ei e chiar cea a expresiei atribuite)

`a = b = c /* asociativ la dreapta, a = (b = c) */`
`if ((c = getchar()) != '\n') { /* folosim rezultatul în test */ }`

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 compuși de atribuire: `+= -= *= /= %= >>= <<= &= ^= |=`
`x += expr` e o formă mai scurtă de a scrie `x = x + expr`

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 */`

```

                                exp-init;
for (exp-init ; exp-test ; exp-cont) while (exp-test) {
    instrucțiune                    instrucțiune;
                                exp-cont;
}

```

* excepție: instrucțiunea `continue`, vezi ulterior

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

În C99 în loc de `exp-init` e permisă o **declarație** de variabile (inițializate) cu domeniu de vizibilitate toată instrucțiunea (dar nu după).

`for (int i = 0; i < 10; i = i + 1) { /* corpul ciclului */ }`

ATENȚIE: Instrucțiunea ; e caz particular al instrucțiunii *expresie* ;
 cu expresia lipsă, deci nu face nimic !

NU scriem ; după paranteza) de la `while` sau `for` decât dacă vrem un ciclu cu corp vid (doar cu test, iar la `for` și cu expresia 3)