

Programarea calculatoarelor

Reprezentare internă. Operatori pe biți

Marius Minea

1 aprilie 2008

Reprezentarea obiectelor în memorie

Orice *valoare* (parametru, variabilă) din program ocupă loc în memorie.

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

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

Operatorul `sizeof`: dimensiunea *în octeți* a unui tip / unei valori
`sizeof(tip)` sau `sizeof expresie` (evaluat la compilare)

Exemplu: `sizeof(char)` e 1: un caracter ocupă (de obicei) un octet

Folosim `sizeof`: – pentru a determina ce tip de date e suficient pentru a cuprinde o valoare de dimensiune dată în octeți
– când vrem să alocăm cantitatea corectă de memorie pentru un obiect

Dimensiunea tipurilor *depinde de sistem* (procesor, compilator):

ex. `sizeof(int)` poate fi 2, 4, 8, ... \Rightarrow nu scriem programul bazat pe o valoare anume ci folosim `sizeof` unde avem nevoie de dimensiune.

Tipuri întregi

Alegem tipul potrivit: înainte de `int` se pot scrie *calificatori* pentru:

- *dimensiune*: `short`, `long` (în C99 și `long long`)
- *semn*: `signed` (implicit, în caz de omisiune), `unsigned`

Cele două se pot combina; `int` poate fi omis: (ex. `unsigned short`)

- `char`: poate fi `signed char` $[-128, 127]$ sau `unsigned char` $[0, 255]$
- `int`, `short`: ≥ 2 octeți, minim $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$
- `long`: ≥ 4 octeți, acoperă minim $[-2^{31} (-2147483648), 2^{31} - 1]$
- `long long`: ≥ 8 octeți, acoperă minim $[-2^{63}, 2^{63} - 1]$
- `unsigned` are dimensiunea tipului cu semn: $[0, 2^{8b} - 1]$ ($b = \text{nr. octeți}$)
- `sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`

Aflăm limitele pe un sistem: constante (macrouri) definite în `limits.h`:
`INT_MIN`, `INT_MAX`, `UINT_MAX` (ex. 65535), la fel cu `CHAR`, `SHRT`, `LONG`

C99: `stdint.h` definește tipuri de dimensiune precizată (cu/fără semn)
`int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

Constante de tipuri întregi

Constante întregi

- în baza 10: scrise obișnuit; ex. -5
- în baza 8: cu prefix cifra zero; ex. 0177 (127 zecimal)
- în baza 16: cu prefix 0x sau 0X; ex. 0xA9 (169 zecimal)
- sufix u sau U pentru unsigned, ex. 65535u
- sufix l sau L pentru long ex. 0177777L

Constante de tip caracter

- caractere tipăribile, între ghilimele simple: '0', '!', 'a'
- caractere speciale:

'\0'	nul	'\a'	alarm
'\b'	backspace	'\t'	tab
'\n'	newline	'\r'	carriage return
'\v'	vert. tab	'\f'	form feed
'\"'	ghilimele	'\''	apostrof
'\\'	backspace		
- caractere scrise în octal (max. 3 cifre), ex: '\14'
- caractere scrise în hexazecimal (prefix x), ex. '\xff'

Tipul caracter e tot un tip întreg (de dimensiuni mai mici).

○ constantă caracter folosită în expresii e convertită automat la `int`

Tipuri reale

Numerele reale: reprezentate ca $semn \cdot (1 + mantisa) \cdot 2^{exponent}$

- domeniul de valori e simetric față de zero
- precizia e *relativă* la mărimea numărului (în modul)

Exemple de *limite* (`float.h`, compilator gcc / i386 / 32 biți / Linux):

- float: 4 octeți, între cca. 10^{-38} și 10^{38} , 6 cifre semnificative

```
FLT_MIN      1.17549435e-38F      FLT_MAX      3.40282347e+38F
```

```
FLT_EPSILON  1.19209290e-07F      // nr.min. cu 1+eps > 1
```

- double: 8 octeți, între cca. 10^{-308} și 10^{308} , 15 cifre semnificative

```
DBL_MIN  2.2250738585072014e-308  DBL_MAX  1.7976931348623157e+308
```

```
DBL_EPSILON  2.2204460492503131e-16  // nr.min. cu 1+eps > 1
```

- pentru precizie suplimentară: long double (12 octeți)

Constante reale

- cu punct zecimal; optional semn și exponent (prefix e sau E)
- în mantisă, fie partea reală fie cea zecimală poate lipsi: 2. .5
- implicit: de tip double; sufix f sau F: float; l sau L: long double
- se recomandă double pt. suficientă precizie la calcul;

funcțiile din `math.h` au tip double, și variante cu sufix: `sin`, `sinf`, `sinl`

Atenție la depășiri și precizie!

- `int` (chiar `long`): domeniu de valori mic (pe 32 biți: cca ± 2 miliarde)
- e insuficient pentru multe calcule cu întregi mari (factorial, etc.)
- folosim reali (`double`): domeniu de valori mare, dar precizie limitată: dincolo de $1E16$ tipul `double` nu mai distinge doi întregi consecutivi !
- o valoare zecimală nu e reprezentată neapărat precis în baza 2, poate fi o fracție periodică: $1.2_{(10)} = 1.(0011)_{(2)}$
`printf("%f", 32.1f);` va scrie 32.099998
- în calcule: pierderi de precizie \Rightarrow rezultatul poate diferi de cel exact
 \Rightarrow decât `x==y` e mai robust să testăm `fabs(x - y) < epsilon` pentru `epsilon` foarte mic, în funcție de specificul problemei
- diferențe dincolo de limita preciziei nu se pot reprezenta
 \Rightarrow pentru `x < DBL_EPSILON` (cca. 10^{-16}) avem `1 + x == 1`

Operatori pe biți

- oferă acces la reprezentarea binară a datelor în memorie
 - facilitează limbajului mașină (de asamblare)
 - pot fi folosiți doar pentru operanzi de orice tip întreg
- | | | |
|----|---|--|
| & | ȘI bit cu bit | (1 doar când ambii biți sunt 1) |
| | SAU bit cu bit | (1 dacă cel puțin un bit e 1) |
| ^ | SAU exclusiv bit cu bit | (1 dacă <i>exact</i> unul din biți e 1) |
| ~ | complement bit cu bit | (valoarea opusă: 1 pt. 0, 0 pt. 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)
⇒ cod neportabil pe alt sistem, nu folosiți pt. nr. cu semn! |

Toți operatorii lucrează simultan pe *toți* biții operanzilor.

nu modifică operanzii, ci dau un rezultat (ca și alți operatori uzuali)

Proprietăți ale operatorilor pe biți

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

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

$1 \ll k$ are doar bitul k pe 1 \Rightarrow e 2^k pentru $k < 8 * \text{sizeof}(\text{int})$

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

0 are toți biții 0, ~ 0 are toți biții 1 (nr. cu semn = -1)

complementul păstrează semnul tipului, deci $\sim 0u$ e fără semn (UINT_MAX)

$\&$ cu 1 păstrează valoarea, $\&$ cu bitul 0 e întotdeauna 0

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

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

$|$ cu 0 păstrează valoarea, $|$ cu bitul 1 e întotdeauna 1

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

\sim cu 0 păstrează valoarea, \sim cu 1 schimbă valoarea în bitului în rezultat

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

Crearea și selectarea unor tipare de biți

& cu 1 nu schimbă, & cu 0 face 0 | cu 0 nu schimbă, | cu 1 face 1

Valoarea dată de biții 0-3 din n: ȘI cu $0\dots01111_{(2)}$ $n \& 0xF$

Resetăm biții 2, 3, 4 din n: ȘI cu $\sim 0\dots011100_{(2)}$ $n \&= \sim 0x1C$

Setăm biții 1-4 din n: SAU cu $11110_{(2)}$ $n = n | 0x1E$ $n |= 036$

Schimbăm biții 0-2 din n: XOR cu $0\dots0111_{(2)}$ $n = n \wedge 7$

⇒ cu operația și *masca* potrivită din 1 și 0 (scrisă ușor în hexa/octal)

Pentru lucrul cu număr de biți dați de o variabilă:

Întregul cu toți biții 1: ~ 0 (cu semn) sau $\sim 0u$ (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 pe 0

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

n deplasat cu p poziții și ștergem toți biții mai puțin ultimii k

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

ștergem toți biții în afară de k biți începând cu cel de ordin p

Conversii explicite și implicite de tip

Conversii implicite: În expresii

- char, short se convertesc la int
- tipul de dimensiune mai mică: convertit la cel de mărime mai mare
- la dimensiuni egale, tipul cu semn e convertit la tipul fără semn
- în expresii mixte întreg-real, întregii sunt convertiți la reali

Conversii la atribuire: se trunchiază când membrul stâng e mai mic !

```
char c; int i;    c = i; // pierde biții superiori din i
```

ATENȚIE: partea dreaptă e evaluată întâi, independent de cea stângă

```
unsigned eur_rol = 37000, usd_rol = 24000 // curs de schimb
```

```
double eur_usd = eur_rol / usd_rol; // rezultatul e 1 !!!
```

(împărțirea întregă e *înainte* de a face conversia prin atribuire la real)

Atribuind real la întreg, se trunchiază spre zero (partea fracționară)

Conversia explicită (type cast): `(numetip) expresie`

expresia e convertită ca și cum ar fi atribuită la o valoare de tipul dat

```
ex. eur_usd = (double)eur_rol / usd_rol // real/intreg dă real
```

Atenție la semn și depășire

ATENȚIE în funcție de sistem, char poate fi signed sau unsigned
⇒ determină semnul dacă bitul 7 e 1, și valoarea în conversia la int
getchar/putchar lucrează cu valori convertite din unsigned char la int

ATENȚIE: practic orice operație aritmetică poate provoca depășire !
`printf("%d\n", 1222000333 + 1222000333); // -1850966630`
(rezultatul are cel mai semnificativ bit setat, și e considerat negativ)
`printf("%u\n", 2154000111u + 2154000111u); // trunchiat la 4032926`

ATENȚIE la comparații și conversii cu semn / fără semn
`if (-5 > 4333222111u) printf("-5 > 4333222111 !!!\n");`
pentru că -5 convertit la unsigned are valoare mai mare !

Comparații corecte între int i și unsigned u:

`if (i < 0 || i < u) respectiv if (i >= 0 && i >= u)`
(compară i cu u doar dacă i e nenegativ)