

## Reprezentarea intregilor. Operatori pe biti

11 octombrie 2005

Programarea calculatoarelor 2. Curs 2a

Marius Minea

Reprezentarea intregilor. Operatori pe biti

## Tipuri de bază (fundamentale)

2

Un *tip*: determină mulțimea valorilor pe care le poate lua o variabilă, și operațiile care pot fi efectuate pe aceste valori.

– reprezentate pe un număr *finit* de octeți ⇒ set *finit* de valori (chiar dacă în matematică, domeniile pentru întregi și reali sunt nelimitate) ⇒ Atenție la depășiri !!!

Limbajul C are doar câteva tipuri de bază.

- **char**: caractere, reprezentate pe 1 octet (8 biți)
  - signed char: -128..127 unsigned char: 0..255
  - char poate corespunde la oricare din ele, conform standardului
- **int**: numere întregi
- **float**: numere reale (virgulă mobilă), în precizie simplă
- **double**: numere reale, în dublă precizie

Domeniul de valori pentru întregi și reali e dependent de arhitectură (de obicei, corespunde natural cu dimensiunea regiștrilor procesorului)

Programarea calculatoarelor 2. Curs 2a

Marius Minea

Reprezentarea intregilor. Operatori pe biti

## Reprezentarea binară a numerelor

3

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)

Exemplu (pe 8 biți):  $11111111(2) == 255$

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

dacă bitul superior e 1, numărul 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):  $11111111(2) == -1$ ;  $10000000(2) == -128$

Adunarea se poate face normal, bit cu bit, indiferent de reprezentare.

Test de depășire: `int x, y, s; scanf("%d%d", &x, &y); s = x + y;`

`if (x < 0 && y < 0 && s >= 0 || x > 0 && y > 0 && s < 0)`

`printf("depasire");`

Programarea calculatoarelor 2. Curs 2a

Marius Minea

Reprezentarea intregilor. Operatori pe biti

## Tipuri întregi

4

Tipul **int** poate primi ca prefix calificatori care specifică:

- **dimensiunea**: **short**, **long** (în C99 și **long long**)
- **semnul**: **signed** (implicit, în caz de omisiune), **unsigned**

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

Standardul prevede următoarele limite:

- **int**, **short**:  $\geq 2$  octeți, minim  $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$
- **long**:  $\geq 4$  octeți, acoperă minim  $[-2^{31}, 2^{31} - 1]$
- **long long**:  $\geq 8$  octeți, acoperă minim  $[-2^{63}, 2^{63} - 1]$
- **unsigned** păstrează dimensiunea; între 0 și  $2^b - 1$  ( $b = \text{nr. octeți}$ )
- $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$

Constante definite în `<limits.h>` (valori minime cerute de standard)

SHRT_MIN, INT_MIN	-32767	SHRT_MAX, INT_MAX	32768
LONG_MIN	-2147483647	LONG_MAX	2147483647
USHRT_MAX, UINT_MAX	65535	ULONG_MAX	4294967295

Programarea calculatoarelor 2. Curs 2a

Marius Minea

Reprezentarea intregilor. Operatori pe biti

## Constante de tipuri întregi

5

**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' null
  - '\a' alarm
  - '\b' backspace
  - '\t' tab
  - '\v' vertical tab
  - '\n' linie nouă
  - '\r' carriage return
  - '\f' form feed
  - '\'' apostrof (ghilimea)
  - '\\' backslash
- caractere scrise în octal (max. 3 cifre), ex: '\14'
- caractere scrise în hexazecimal (prefix x), ex. '\xff'

Programarea calculatoarelor 2. Curs 2a

Marius Minea

Reprezentarea intregilor. Operatori pe biti

## Tabela de caractere ASCII

6

ASCII = American Standard Code for Information Interchange

Caracterele sunt memorate ca și cod numeric = indicele în acest tabel ex. '0' == 48, 'A' = 65, 'a' = 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 { | } ~

– caracterele < 0x20 (spațiu): caractere de control

– caracterele cu cod > 0x7f (127): nu fac parte din setul ASCII

(diacritice, diverse variante naționale standardizate de ISO, etc.)

Programarea calculatoarelor 2. Curs 2a

Marius Minea

## Funcții de clasificare a caracterelor (în ctype.h)

```

int isalnum(int c) (isalpha(c) || isdigit(c))
int isalpha(int c) ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
int isblank(int c) (c == ' ' || c == '\t')
int iscntrl(int c) // caracter de control, valoare: 0 - 31
int isdigit(int c) ('0' <= c && c <= '9')
int isgraph(int c) // caracter tipăribil, exceptând spațiu
int islower(int c) ('a' <= c && c <= 'z')
int isprint(int c) // caracter tipăribil, inclusiv spațiu
int ispunct(int c) // tipăribil și nu alnum și nu spațiu
int isspace(int c) (c == ' ' || c == '\t' || c == '\n' ||
/* "spatii albe" */ c == '\v' || c == '\f' || c == '\r')
int isupper(int c) ('A' <= c && c <= 'Z')
int isxdigit(int c) // cifră hexazecimală: 0-9, A-F, a-f
int tolower(int c) // A - Z -> a - z, restul neschimbat
int toupper(int c) // a - z -> A - Z, restul neschimbat

```

## Dimensiunea tipurilor

Operatorul `sizeof` dă numărul de *octeți* de memorie ocupați de operand (un tip de date sau o expresie – în particular o variabilă).  
– dacă operandul e un tip, trebuie pus între ( paranteze )  
– dacă e o expresie, rezultatul e dat de tipul său (fără evaluarea ei)

```

#include <stdio.h>
int main (void)
{
    printf("char\t\t%d\n",sizeof(char));
    printf("short\t\t%d\n",sizeof(short));
    printf("int\t\t%d\n",sizeof(int));
    printf("long\t\t%d\n",sizeof(long));
    return 0;
}

```

C99 definește *tipuri întregi de dimensiune data* în `<stdint.h>` :

```
int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t
```

## Operatori aritmetici

## Operatori aritmetici

– operatorii uzuali binari: +, -, \*, / pentru numere întregi și reale  
ATENȚIE: pentru întregi, / înseamnă împărțire cu rest  
– operatorul % (numai pentru întregi): modulo (restul la împărțire)  
9/-5== -1 9%-5==4 -9/5== -1 -9%-5== -4 -9/-5==1 -9%-5== -4  
(restul are semnul deîmpărțitului)  
– operatorul unar - (minus; nu există plus unar).

În expresii aritmetice, caracterele sunt considerate ca și întregi

(indicele caracterului respectiv în tabela ASCII)

Exemple: '7' - '0' == 7, 'a' + 5 == 'f'

(cifrele, literele mari, literele mici: 3 grupuri continue în tabela ASCII)

Conversii: lit\_mica = lit\_mare + 'a' - 'A'; cifra = valoare + '0';

Precedența: - unar, apoi \* / %, apoi + -

## 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 ~ complementare bit cu bit
| SAU bit cu bit << deplasare la stânga
^ SAU exclusiv bit cu bit >> deplasare la dreapta

```

De obicei testam, setam, resetam sau inversam bitul *k* dintr-un număr.  
Cream un număr ( $2^k$ ) care are 1 pe bitul *k* și în rest 0 : `1 << k`

Testare `n & (1 << k)` nenul dacă bitul *k* din *n* e 1

Setare `n = n | (1 << k)` bitul *k* din *n* devine 1, ceilalți neschimbați

Resetare `n = n & ~(1 << k)` bitul *k* din *n* devine 0, ceilalți neschimbați

Inversare `n = n ^ (1 << k)` bitul *k* din *n* se schimbă, ceilalți nu

## Exemple cu operatori pe biți

- `n & 0xF` are ultimii 4 biți mai puțin semnificativi la fel ca *n*, restul 0 (pentru *n* fără semn, echivalent cu `n % 16`; `n & 1 == n % 2`)
- `n | 0200` are bitul 7 pe 1, și toți ceilalți biți la fel ca *n*
- `n ^ 1` are ultimul bit schimbat față de *n*, toți ceilalți la fel
- `~0 == -1` : întreg *cu semn* cu toți biții 1
- `~0u == UINT_MAX` : întreg *fără semn* (unsigned) cu toți biții 1
- `n < 0` (pentru *signed n*) – echivalent cu testarea bitului de semn
- `~0xf` are ultimii 4 biți pe 0 și restul pe 1
- `n << 3` are biții lui *n* deplasați 3 poziții la stânga, și ultimii 3 biți 0
- `n >> 2` are biții lui *n* deplasați 2 poziții la dreapta; primii 2 biți devin:
  - 0, dacă *n* este unsigned, sau signed dar nenegativ
  - dependent de implementare (tipic: 1), dacă *n* este signed negativ
- **ATENȚIE** la efectul deplasării la dreapta în funcție de semn !
- `<< și >>` : ca și înmulțiri/împărțiri cu puterile lui 2 (mai eficiente)

## Operatori pe biți (cont.)

Operatori compuși de atribuire (pt. cei binari): `&=` `|=` `^=` `<<=` `>>=`  
`n &= a` e o notatie prescurtata pentru `n = n & a` etc.

Exemple: extragerea unei porțiuni din reprezentarea unui număr:  
creem o *mască* (un tipar) în care biții respectivi sunt pe 0 (sau 1)

```

~0 << k are ultimii k biți pe 0, restul pe 1
~(~0 << k) are ultimii k biți pe 1, restul pe 0
~(~0 << k) << p are k biți pe 1, începând de la bitul p, și restul 0
(n >> p) & ~(~0 << k)

```

are pe ultimele poziții cei *k* biți ai lui *n* începând cu bitul *p*, și în rest 0  
`n & (~0 << k) << p`  
are cei *k* biți începând cu bitul *p* la fel ca ai lui *n*, și restul biților pe 0