

Limbaje de programare

Reprezentare internă. Operatori pe biți. Tablouri

31 octombrie 2011

Reprezentarea obiectelor în memorie

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

bit = cea mai mică unitate de memorare, are două valori (0 sau 1)

octet (byte) = grup de 8 biți, destul pentru a memora un caracter

E cea mai mică unitate de memorie *adresabilă* direct

(se poate citi/scrie independent: nu putem citi/scrie doar un bit)

Operatorul *sizeof*: dimensiunea *în octeți* a unui tip / unei valori

`sizeof(tip)` sau `sizeof expresie`

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

Un întreg are `sizeof(int)` *octeți* \Rightarrow `8*sizeof(int)` *biți*

`sizeof` e un *operator*, evaluat la compilare. NU e o funcție

sizeof și scrierea de programe portabile

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

ex. `sizeof(int)` poate fi 2, 4, 8, ... în funcție de sistem

⇒ folosim `sizeof` ca să aflăm dimensiunea unui tip / unei variabile
nu presupunem că ar fi 2, 4, etc., poate fi incorect pe alt sistem
(programul nu este *portabil* de pe o arhitectură pe alta)

Folosim `sizeof`:

- pentru a afla câți octeți ocupă o valoare

- pentru a afla ce tip de date poate cuprinde o anumită valoare
(de dimensiune dată în octeți)

- ca să alocăm cantitatea corectă de memorie pentru un obiect

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)

Domeniul de valori: de la 0 la $2^k - 1$ Ex: 11111111 e 255

$c_0 = 0 \Rightarrow$ număr par; $c_0 = 1 \Rightarrow$ număr impar

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

dacă bitul superior e 1, nr. se consideră negativ

\Rightarrow Domeniul de valori: de la -2^{k-1} la $2^{k-1} - 1$

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

Exemple (pe 8 biți):

11111111 e -1

111111110 e -2

10000000 e -128

Tipuri întregi

Înainte de `int` se pot scrie *calificatori* pentru:

dimensiune: `short`, `long` (în C99 și `long long`)

semn: `signed` (implicit, dacă e omis), `unsigned`

Le putem combina; putem omite `int`. Ex: `unsigned short`

`char`: `signed char` $[-128, 127]$ sau `unsigned char` $[0, 255]$

`int`, `short`: ≥ 2 octeți, min. $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$

`long`: ≥ 4 octeți, acoperă minim $[-2^{31} (-2147483648), 2^{31} - 1]$

`long long`: ≥ 8 octeți, acoperă minim $[-2^{63}, 2^{63} - 1]$

Tipurile cu și fără semn au aceeași dimensiune ($b = \text{nr. octeți}$)

`sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`

Limitele sunt constante (macro-uri) definite în `limits.h`

`INT_MIN`, `INT_MAX`, `UINT_MAX` (ex. 65535), la fel pt. `CHAR`, `SHRT`, `LONG`

C99: `stdint.h` are 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: se pot scrie în program doar în baza 8, 10, 16

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
'\v'	vert. tab	'\f'	form feed	'\r'	carriage return
'\"'	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).

O constantă caracter e convertită automat la int în expresii.

Reprezentarea numerelor reale

Numerele reale se reprezintă cu semn, exponent, și mantisă

$$(-1)^{\text{semn}} * 2^{\text{exp}} * 1.\text{mantisă}_{(2)}$$

Pe biți: S EEEEEEEEE MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

float: 4 octeți: 1+8+23 biți; double: 8 octeți: 1+11+52 biți

pt. $0 < E < 255$ obținem numerele $(-1)^S * 2^{E-127} * 1.M_{(2)}$

pt. $E = 0$, numere f. mici (denormalizate): $(-1)^S * 2^{-127} * 1.M_{(2)}$

mai avem: reprezentări pentru ± 0 , $\pm \infty$, erori (NaN)

Precizia numerelor reale e *relativă* la modulul lor (“virgulă mobilă”)

Ex: cel mai mic float > 1 e $1 + 2^{-23}$ (ultima poz. în mantisă 1)

La numere mari, imprecizia absolută crește:

De ex. $2^{24} + 1 = 2^{24} * (1 + 2^{-24})$, ultimul bit nu are loc în mantisă

\Rightarrow va fi rotunjit; nu toți întregii pot fi reprezentați ca float

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 pe 32 biți):

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
```

long double: pentru precizie și mai mare (12 octeți)

Constante reale: pot fi scrise în următoarele forme:

cu punct zecimal; optional semn și exponent (prefix e sau E)

în mantisă, partea reală sau zecimală pot lipsi: 2. .5

Tip implicit: double; sufix f, F: float; l, L: long double

Se recomandă double pentru precizie suficientă în calcule.

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

Atenție la depășiri și precizie!

int (chiar long) au domeniu de valori mic (pe 32 biți: ± 2 miliarde)

Pentru multe calcule cu întregi mari (factorial, etc.), e insuficient
 \Rightarrow folosim reali (double): domeniu de valori mare

Realii au 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 înlocuim `x==y` cu `fabs(x - y) < ceva foarte mic`
pentru *ceva foarte mic* ales în funcție de specificul problemei

Diferențe mai mici 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
facilități apropiate 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 *exact* 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 întregă)

Deci $1 \ll k$ ar 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)

$\sim x$ are tip de același semn, 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

\wedge cu 0 păstrează valoarea, \wedge cu 1 schimbă val. bitului în rezultat

$n \wedge (1 \ll k)$ *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: ȘI cu $\sim 0\dots011100_{(2)}$ $n \&= \sim 0x1C$
Setăm biții 1-4: SAU cu $11110_{(2)}$ $n = n | 0x1E$ $n |= 036$
Schimbăm biții 0-2 din n : XOR cu $0\dots0111_{(2)}$ $n = n \wedge 7$
 \Rightarrow alegem operația și *masca* (valoarea, scrisă ușor în hexa/octal)

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

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

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, 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 mărime mai mică e 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
!!! Partea dreaptă e evaluată întâi, independent de cea stângă
unsigned eur_rol = 43000, usd_rol = 31000 // curs valuta
double eur_usd = eur_rol / usd_rol; // rezultatul e 1 !!!
(împărțire întregă înainte de conversia prin atribuire la real)
Atribuind real la întreg, se trunchiază spre zero (partea fracționară)

Conversia explicită (type cast): (numetip) expresie
convertește expresia ca și prin atribuire la o valoare de tipul dat
eur_usd = (double)eur_rol / usd_rol // real/întreg dă real

Atenție la semn și depășire

ATENȚIE char poate fi signed sau unsigned, depinde de sistem
⇒ valori diferite dacă bitul 7 e 1, și în conversia la int
getchar/putchar lucrează cu unsigned char convertit la int

ATENȚIE: practic orice operație aritmetică poate provoca depășire!

```
printf("%d\n", 1222000333 + 1222000333); // -1850966630
```

(rezultatul are cel mai semnificativ bit 1, și e considerat negativ)

```
printf("%u\n", 2154000111u + 2154000111u); // trunchiat: 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)

Declararea tablourilor

Tablou (vector) = un șir de elemente de *același tip* de date

Tabloul x asociază la un *indice* n , o *valoare* $x[n]$

În matematică, același lucru face un *șir* x_n sau o *funcție* $x(n)$

Declarare: *tip nume-tablou[nr-elem];*

```
double x[20];   int mat[10][20];
```

Inițializare: între acolade, cu virgule:

```
int a[4] = { 0, 1, 4, 9 };
```

Dimensiunea tabloului (nr. de elemente) = o *constantă* pozitivă

C99 acceptă și dimensiuni variabile, cu valoare cunoscută în momentul declarării

```
void f(int n) { int tab[n]; /* n e cunoscut la apel */ }
```

Sintaxa declarației: *tip a[dim];* spune că $a[\text{indice}]$ are tipul *tip*

Folosirea tablourilor

Un *element* de tablou *nume-tab[indice]* e folosit ca orice *variabilă* are o valoare, poate fi folosit în expresii, poate fi atribuit

```
x[3] = 1; n = a[i]; t[i] = t[i + 1]
```

Indicele poate fi orice *expresie* cu valoare *întreagă*

ATENȚIE! În C, indicii de tablou sunt de la *zero* la *dimensiune - 1*

```
int a[4]; conține a[0], a[1], a[2], a[3], NU există a[4]
```

Exemplu de traversare și atribuire a unui tablou:

```
int a[10]; for (int i = 0; i < 10; ++i) a[i] = i + 1;
```


Constante simbolice ca dimensiuni de tablou

E util să folosim un nume de *constantă (macro)* pentru dimensiune

```
#define NUME val
```

Preprocesorul C înlocuiește NUME în sursă cu val înaintea compilării

```
#define LEN    30
double t[LEN];
for (int i = 0; i < LEN; ++i) { // tabelam sin cu pas 0.1
    t[i] = sin(0.1*LEN); printf("%f ", t[i]);
}
```

Programul e mai ușor de citit, e clar că LEN e lungimea tabloului.

Dacă vrem altă dimensiune, modificăm programul doar într-un loc

⇒ evităm greșelile din neatenție sau uitare

Exemplu: Calculul primelor numere prime

```
#include <stdio.h>
#define MAX 100          // preprocesorul inlocuieste MAX cu 100
int main(void) {
    unsigned p[MAX] = {2}; // primul element initializat cu 2
    unsigned cnt = 1, n = 3; // 2 e prim, 3 e urm. candidat
    do {
        for (int j = 0; n % p[j]; ++j) // cat timp nu gasim divizor
            if (p[j]*p[j] > n) { // nu mai pot fi alti divizori
                p[cnt++] = n; break; // memoreaza, iese din ciclu
            }
        n += 2; // trecem la numarul impar urm.
    } while (cnt < MAX); // pana nu e plin tabloul
    for (int j = 0; j < MAX; ++j)
        printf("%d\n", p[j]); // tiparim un element pe rand
    return 0;
}
```

Variabile și adrese

Orice variabilă x are o adresă: acolo e memorată valoarea ei

Operatorul prefix & dă adresa operandului: $\&x$ e adresa variabilei x
Operandul lui $\&$: orice *lvalue* (destinație de atribuire): variabile, elemente de tablou. NU au adrese: alte expresii, constantele

Numele unui tablou e chiar *adresa* tabloului.

Fie `int a[6];` Numele `a` reprezintă *adresa* tabloului.

Numele `a` NU reprezintă toate elementele împreună!

O adresă poate fi tipărită (în baza 16) cu formatul `%p` în `printf`

```
#include <stdio.h>
int main(void) {
    double d; int a[6];
    printf("Adresa lui d: %p\n", &d); // folosim operatorul &
    printf("Adresa lui a: %p\n", a); // a e adresa, nu trebuie &
    return 0;
}
```

Tablouri ca parametri la funcții

Declarația unui tablou alocă și memorie pentru elementele sale dar *numele* reprezintă *adresa* sa și nu tabloul ca tot unitar

⇒ numele tabloului *NU* poartă informații despre dimensiunea lui
excepție: `sizeof(numetab)` este $nr\text{-elem} * \text{sizeof}(tip\text{-elem})$

La funcții se transmit *numele* tabloului (*adresa*) și *lungimea* sa
NU scriem lungimea între [] la parametru, nu se ia în

considerare

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

```
void printtab(int t[], unsigned len) {  
    for (int i = 0; i < len; ++i) printf("%d ", t[i]);  
    putchar('\n');
```

```
}  
  
int main(void) {  
    int prim[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };  
    printtab(prim, 10); // ATENȚIE: NU prim[10], NU prim[]  
    return 0;  
}
```

Tablouri ca parametri la funcții

Transmiterea parametrilor în C se face *prin valoare*

⇒ un parametru tablou e transmis prin *valoarea adresei sale*

Având adresa, funcția poate *citi și scrie* elementele tabloului

```
void sumvect(double a[], double b[], double r[], unsigned len)
{
    for (unsigned i = 0; i < len; ++i) r[i] = a[i] + b[i];
}
#define LEN 3 // macro pt. lungimea tablourilor
int main(void) {
    double a[LEN] = {0, 1.41, 1}, b[LEN] = {1, 1.73, 1}, c[LEN];
    sumvect(a, b, c, LEN);
    return 0;
}
```

Inițializare

Tablourile neinițializate au elemente de valoare necunoscută.

Tablourile inițializate parțial au restul elementelor nule.

Tablouri și șiruri de caractere

```
char cuvant[20]; // tablou de caractere neinitializat
char msg[] = "test"; // 5 octeti, terminat cu '\0'
char msg[] = {'t','e','s','t','\0'}; // acelasi, scris altfel
char nume[3] = { 'E', 'T', 'C' }; // nu are '\0' la sfarsit !
char sir[20] = "test"; // restul pana la 20 sunt '\0'
```

În C, termenul *șir de caractere* înseamnă un tablou de caractere încheiat în memorie cu caracterul '\0'.

Tot cu '\0' se termină și *constantele șir*: "salut\n"
terminatorul '\0' nu se citește / tipărește

ATENȚIE: toate funcțiile standard pentru șiruri au nevoie de șiruri terminate cu '\0'

La șiruri inițializate, dar fără dimensiune dată (ex. msg mai sus) se alocă dimensiunea inițializatorului + 1 pt. caracterul '\0'

Tipul pointer

Rezultatul unei operații *adresă* are un tip, ca și orice expresie

Pentru o variabilă declarată *tip* x; *tipul adresei sale* &x e *tip* *
(*pointer la tip*, adică: adresă unde se află un obiect de acel *tip*)

Numele unui tablou are tipul pointer la tipul elementului

int a[4]; a are tipul int * char s[8]; s are tipul char *

În antetul funcției, void f(*tip* a[]) înseamnă void f(*tip* *a)

(de aceea dimensiunea: void f(*tip* a[6]) *nu contează*)

Tipul unei constante șir de caractere "sir" este char *:

adresa unde se găsește șirul în memorie

Valoarea NULL (0 de tip void *, adică adresă de tip neprecizat)
indică o adresă *invalidă*.

Funcții cu șiruri de caractere (string.h)

```
size_t strlen(const char *s); // returneaza lungimea sirului s
char *strchr(const char *s, int c); // cauta caract. c in s
// returneaza adresa unde e gasit sau NULL daca nu e gasit
char *strcpy(char *dest, const char *src); // copie src in dest
char *strcat(char *dest, const char *src); // concat src la dest
// pentru ambele e necesar ca la dest sa fie loc suficient
int strcmp (const char *s1, const char *s2); // compara
// returneaza intreg < 0 sau == 0 sau > 0 dupa cum e s1 fata de
char *strncpy(char *dest, const char *src, size_t n);
// copiaza cel mult n caractere din src in dest
char *strncat(char *dest, const char *src, size_t n);
// concateneaza cel mult n caractere din src la dest
int strncmp (const char *s1, const char *s2, size_t n);
// compara sirurile pe lungime cel mult n caractere
```

size_t: tip întreg fără semn pentru dimensiuni

const: specificator de tip: obiectul respectiv nu e modificat