

Reprezentarea realilor. Operatori. Expresii

12 octombrie 2005

Reprezentarea numerelor reale: Standardul IEEE 754-1985

simplă precizie (float): S EEEEEEEE MMMMM...MMMMM (1+8+23=32 biți)

de la cel mai semnificativ: semn, exponent (bias 127), mantisă

– E = 255, M != 0: NaN (not a number)

– E = 255, M = 0: $+\infty, -\infty$ (după bitul de semn)

– $0 < E < 255$: $(-1)^S * 2^{E-127} * 1.M$ (mantisă are implicit o unitate)

Ex.: 0 10000001 0100...00 = $2^{129-127} * 1.01_{(2)} = 4 * 1.25 = 5$

Max.: $2^{254-127} * 1.11...11_{(2)} \simeq 2^{127} * 2 \simeq 3 * 10^{38}$

Min. (in modul): $2^{1-127} * 1.00...00_{(2)} = 2^{-126} \simeq 10^{-38}$

– E = 0, M = 0: +0, -0 (in functie de bitul de semn)

– E = 0, M != 0: $(-1)^S * 2^{-126} * 0.M$: valori *nenormalizate* (f. mici)

Min. (in modul): $2^{-126} * 0.00...001_{(2)} = 2^{-149} \simeq 10^{-45}$

Numere reale in *dublă precizie* (double): 64 biți

aceleași reguli, 11 biți exponent (bias 1023), 52 biți mantisă

Tipuri reale

Numerele reale: reprezentate cu semn, mantisă, și exponent

⇒ domeniul de valori e simetric față de zero

⇒ precizia se definește relativ la modulul numărului

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

– pentru precizie suplimentară: `long double` (12 octeți)

Constante reale

– conțin mantisă, iar optional semn și exponent (prefix `e` sau `E`)

mantisa poate fi și în baza 16, iar exponentul în baza 2: prefix `p` sau `P`

– 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`

`3.14159265358979323846`

`1.175494e-38f`

Limitele tipului real

Constante limita definite în float.h

valori pt. gcc/i386/Linux (in paranteza cerințele minime standard)

```
FLT_DIG      6    // precizie in cifre zecimale pt float
DBL_DIG      15 (min. 10)    // precizie pentru double
FLT_MAX      3.40282347e+38F (min. 1E+37) // max. reprezentabil
FLT_MIN      1.17549435e-38F (max. 1E-37) // min. reprez. in modul
FLT_EPSILON  1.19209290e-07F (max. 1E-5)  // nr.min. cu 1+eps > 1
```

1 are mantisa: (1.)000...000 (23 biti de mantisa)

Urmatorul numar reprezentabil are mantisa: (1.)000...001

Din diferenta ($2^{-23} \simeq 10^{-7}$) rezulta valoarea FLT_EPSILON, si FLT_DIG

Precizia e *relativa*, data de FLT_EPSILON * marimea numarului

– float poate reprezenta numere mai mari decat int (si subunitare),

dar precizia de ex. pt numere de 10^9 e FLT_EPSILON * $10^9 \simeq 100$

– pe int32_t se pot reprezenta *toti* intregii dar numai pana la $\simeq \pm 2 * 10^9$

```
DBL_MAX      1.7976931348623157e+308 (min. 1E+37)
```

```
DBL_MIN      2.2250738585072014e-308 (max. 1E-37)
```

```
DBL_EPSILON  2.2204460492503131e-16 (max. 1E-9)
```

Atenție la precizie!

- `int` (chiar `long`): domeniu de valori mic (cca ± 2 miliarde)
- e insuficient pentru multe calcule care implică aparent întregi
- Ex. calculați $e^{-x} = 1 - x^1/1! + x^2/2! - \dots$ cu o precizie dată (10^{-5})
- Nu încercați: `long fact(long n) { /* ... */ }` (depășire pt. $n > 12$)
- mai bine: fără factorial, cu recurență între termeni: $t_n = t_{n-1} * x/n$
- până la $9E15$ tipul `double` distinge încă 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 $1.2_{(10)} = 1.(0011)_{(2)}$
- în calcule matematice, adeseori comparația `==` e insuficientă
(pot apare pierderi de precizie pe parcurs)
- mai bine: `fabs(x - y) < ceva_mic` (`fabs`: val. absolută, în `math.h`)

Operatori relaționali și logici

C nu are tip boolean; se folosește `int` (C99: `_Bool`, `stdbool.h`)

- operatorii logici produc 1 pt. *true*, 0 pt. *false*
- un întreg e interpretat ca *true* dacă e $\neq 0$ și ca *false* dacă e 0

Operatorii relaționali: precedența mai mică decât cei aritmetici

$x < y + 1$ înseamnă în mod natural $x < (y + 1)$

precedența: întâi `>`, `>=`, `<`, `<=`, apoi `==`, `!=` (egal, diferit)

Operatorii logici binari: `&&` (ȘI), prioritar lui `||` (SAU)

– precedență mai mică decât cei relaționali

⇒ se poate scrie natural $(x < y + z \ \&\& \ y < z + x)$

– sunt evaluați de la stânga la dreapta

– **evaluarea se oprește** (*short-circuit*) când rezultatul e cunoscut (dacă primul argument al lui `&&` (resp. `||`) e fals (resp. adevărat)

Exemplu: `if (p != 0 && n % p == 0) { /* nu împarte la 0 */ }`

Operatorul logic unar `!` (negație logică)

– cea mai ridicată prioritate (ca și toți operatorii unari)

– transformă operand non-zero în 0, și zero în 1

Ex: `if (!gasit)` e echivalent cu `if (gasit == 0)`

Conversia între tipuri

În expresii, operanzii de tipuri diferite sunt convertiți la un tip comun.

Conversia din real la întreg (ex. atribuire): prin trunchiere (înspre zero)

Conversiile aritmetice uzuale:

- operandul de dimensiune/precizie mai mică e convertit la tipul operandului de dim./prec. mai mare (în ordinea: `long double`, `double`, `float`)
- operanzii de tipuri de rang inferior lui `int` (`char`, `short`) sunt convertiți la tipurile `int` sau `unsigned` (după semn)
- dacă ambii operanzi au tipuri cu, resp. fără semn, se convertesc la tipul de rang (dimensiune) mai mare
- dacă semnele tipurilor sunt diferite, și unul din tipuri cuprinde toate valorile celuilalt, se face conversia la tipul cel mai cuprinzător
- dacă nu, operanzii se convertesc la tipul fără semn corespunzător operandului care are tip cu semn

Exemplu: între `int` și `unsigned`, conversie la `unsigned` (ultima regulă)

Probleme la conversie

ATENȚIE: în funcție de arhitectură, char poate fi signed sau unsigned
⇒ determină semnul caracterelor cu bitul 7 pe 1, și implicit semnul la conversia char → int

ATENȚIE la conversia/comparația între int și unsigned !!
valorile > INT_MAX sunt considerate negative ca int
⇒ rezultate incorecte / surprinzătoare / neintuitive

```
int i; unsigned u = 3000000000; /* u > INT_MAX */
i = u + 5; /* bitul de semn 1, deci i e considerat negativ */
if (i > u) printf("%d > %u\n", i, u);
/* tipareste: -1294967291 > 3000000000 !!! */
```

Pentru a compara int i cu unsigned u
– înlocuiți (i < u) cu (i < 0 || i < u)
– înlocuiți (i > u) cu (i > 0 && i > u)

Conversia între tipuri (cont.)

Conversia la atribuire: partea dreaptă convertită la tipul părții stângi
– e posibilă trunchierea dacă atribuim la un tip de dimensiune mai mică
⇒ mesaje de avertizare de la compilator

Exemplu: `int i; char c;`

```
i = c; c = i; /* valoarea se păstrează */
```

```
c = i; i = c; /* biții superiori se pierd */
```

Atentie: partea dreaptă e evaluată independent de tipul părții stângi!

```
unsigned eur_rol = 36000, usd_rol = 30000;
```

```
float eur_usd;
```

```
eur_usd = eur_rol / usd_rol; // 1 !!!
```

Operatorul de conversie explicită (engl. *type cast*)

Sintaxa: `(nume_tip) expresie`

expresia este convertită ca în atribuirea unei variabile de tipul dat

```
eur_usd = (double) eur_rol / usd_rol; // 1.2
```

```
int n; sqrt((double)n); /* double sqrt(double) in math.h */
```

Operatori de atribuire

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

⇒ 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 $x = x + expr$

vezi ulterior și pentru operatorii pe biți & | ^ << >>

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

```
int x=2, y, z; y = x++; /* y=2,x=3 */; z = ++x; /* x=4,z=4 */
```

Câteva exemple

– numără caracterele din șirul `s` în variabila `i`

```
for (i = 0; s[i] != '\0'; i++); /* șirul se termina cu '\0' */
```

sau, cu un test implicit de valoare nonzero, și preincrement:

```
for (i = -1; s[++i]; ); // ultima ; inseamna corp vid pentru for
```

– copiază șirul `src` în șirul `dest`; expresia atribuită servește și pt. test

```
for (i = 0; dest[i] = src[i]; ++i);
```

– copiază max. `N` caractere; când primul test e fals, se omite al doilea (deci nu se mai execută atribuirea)

```
for (i = 0; i < N && dest[i] = src[i]; ++i );
```

– rezultatul unei funcții e atribuit și testat în aceeași expresie:

```
for (i = 0; i < N-1 && (c = getchar()) != EOF; ) s[i++] = c;
```

Expresii: valoare și efect lateral

Orice expresie are o *valoare*, definită prin semantica operatorilor.

Unele expresii au și *efect lateral*: modifica starea programului:

atribuirea modifica variabile; citirea/scrierea modifica intrarea/iesirea

Exemplu: `++i` și `i++` au același efect lateral (incrementează pe `i`)

dar dau prin evaluare valori diferite (deja incrementată / încă nu)

ATENȚIE: Limbajul C *nu specifica ordinea de evaluare* a operanzilor unei expresii (depinde de implementare). Excepții: `&&` `||` `?:` ,

⇒ expresii cu mai multe efecte laterale pot avea efect nedeterminat.

```
int i = 0; printf("%d %d", i++, i++); // scrie 0 1 sau 1 0
```

(argumentele unei funcții se pot evalua în orice ordine !!)

```
while (s[i] < s[++i]); // pana unde e crescator sirul ?
```

(dar dacă `s[++i]` e evaluat întâi, îl comparăm cu el însuși !!)

```
if ((c = getchar()) == '*' && (c = getchar()) == '/')
```

(daca intram pe 'else', s-a evaluat si partea a doua sau nu ?)

ATENȚIE! NU folosiți mai multe efecte laterale în aceeași expresie sau în partea a doua a lui `&&` și `||`. Nu scrieți cod compact, ci cod corect!

Alți operatori

Operatorul condițional

Sintaxa: $expr1 ? expr2 : expr3$

- dacă $expr1$ e adevărată, rezultatul e dat de evaluarea lui $expr2$;
- dacă $expr1$ e falsă, rezultatul e dat de evaluarea lui $expr3$
- mai concisă decât `if ... else ...`

Exemple: `m = (a > b) ? a : b; /* max(a, b) */`

`printf("Numărul este %s\n", (n < 0) ? "negativ" : "nenegativ");`

`putchar(v > 9 ? v - 10 + 'A' : v + '0');` // scrie cifra hexa

Operatorul secvențial

Sintaxa: $expr1 , expr2$ /* operatorul este virgula */

- se evaluează $expr1$, apoi $expr2$; rezultatul e dat de $expr2$
- se folosește când e nevoie de mai multe evaluări, dar sintaxa prevede o singură expresie (de ex. în `if`, `for`, `while`)

Exemple: `for (p = 1, i = j = 0; i < n; i++, j++) { /* ... */ }`

`while (printf("Numărul?"), scanf("%d", &n) == 1) { /* ... */ }`

Precedența și asociativitatea operatorilor

Precedența (descrescătoare ↓)

() [] -> .

! ~ ++ -- - (*tip*) * & (pt.adrese) sizeof

* / %

+ -

<< >>

< <= > >=

== !=

&

^

|

&&

||

? :

= += -= etc.

,

Asociativitate

→

←

→

→

→

→

→

→

→

→

→

→

←

←

→

ATENȚIE: În caz de dubiu, și pentru lizibilitate, folosiți parantezele !

Atenție la precedență!

În multe situații frecvent întâlnite în programe trebuie paranteze!

– dacă vrem să atribuim o valoare și apoi să o testăm:

```
while ((c = s[++i]) != '\0') { /* prelucram c cat e nenul */ }
```

dar: `c = s[++i] != '\0'` îi dă lui `c` o valoare booleană (0 sau 1)

– dacă vrem să deplasăm pe biți și apoi să adunăm:

```
n = (hi << 8) + lo /* facem un int din doi octeți */
```

dar: `hi << 8 + lo` deplasează pe `hi` la stânga cu `10+8` biți

– dacă vrem să testăm valoarea unui grup de biți dintr-un număr

```
if ((n & mask) == val) { /* testeaza bitii selectati de mask */ }
```

dar: `n & mask == val` face ȘI cu booleanul `mask == val` (0 sau 1)

Funcții matematice standard (declarate în `math.h`)

Funcții de conversie

`double fabs(double x);` valoarea absolută a lui `x`
`double floor(double x);` partea întreagă $\lfloor x \rfloor$ a lui `x`, ca `double`
`double ceil(double x);` cel mai mic întreg $\lceil x \rceil$ nu mai mic de `x`
`double trunc(double x);` trunchează argumentul la întreg, înspre 0

Funcții de rotunjire (Obs: direcția de rotunjire poate fi controlată cu `fgetround()` și `fsetround()` din `fenv.h`, detalii în standard)

`double nearbyint(double x);` rotunjesc în direcția curentă cu/
`double rint(double x)` /fără excepție de argument *inexact*
(implementarea/tratarea excepțiilor e definită în standard, v. `fenv.h`)
`double round(double x);` rotunjește jumătățile în direcția opusă lui zero
`long int lrint(double x);` `long int lround(double x);`
ca și `rint()`, `round()` dar rezultat întreg; nedefinit în caz de depășire

Funcțiile din `math.h` au variante cu sufixele `f` și `l` cu argumente și rezultate `float` sau `long double`. Exemple: `float fabsf(float);` `long double fabsl(long double);`

Funcții standard din `math.h` (cont.)

Funcții de exponențiere și logaritmice

`double exp(double x);` returnează e^x
`double exp2(double x);` returnează 2^x
`double log(double x);` returnează logaritmul natural $\ln x$
`double log10(double x);` `double log2(double x);` log. în baza 10 și 2
`double pow(double x);` returnează x^y
`double sqrt(double x);` returnează \sqrt{x}

Funcții trigonometrice și hiperbolice

`acos`, `asin`, `atan`, `cos`, `sin`, `tan`, `acosh`, `asinh`, `atanh`, `cosh`, `sinh`, `tanh`
(valori unghiulare în radiani; inversele returnează valori principale)
`double atan2(double y, double x);` returnează $\arctg(y/x)$ în intervalul $[-\pi, \pi]$, determină cadranul după semnele ambelor argumente