

Tipuri de bază (fundamentale)

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

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

Limbajul C are doar câteva tipuri de bază.

- **char**: caractere, reprezentate pe 1 octet (8 biți)
- **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)

Tipuri. Operatori. Expresii

19 octombrie 2004

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 == 255; $c_0 = 0 \Rightarrow$ nr. par; $c_0 = 1 \Rightarrow$ nr. impar

Întregi *cu semn*: reprezentate 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 == -1; 10000000 == -128

Numerele reale (**float**: semn, exponent, mantisă)

S EEEEEEEE MMMMMMMMMMMMMMMMMMM (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)

Tipuri întregi

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 (definiții în `<limits.h>`)

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

Dimensiunea în octeți a acestor tipuri variază în funcție de implementare

(Turbo C sub Windows, gcc sub Linux, procesoare diferite de x86)

\Rightarrow folosiți **sizeof** pentru a scrie programe *portabile*

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:
 - '\n' linie nouă
 - '\r' carriage return
 - '\b' backspace
 - '\'' apostrof (ghilimea)
 - '\t' tab
 - '\'\' backslash
- caractere scrise în octal (max. 3 cifre), ex: '\14'
- caractere scrise în hexazecimal (prefix x), ex. '\xff'

Tabela de caractere ASCII

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, etc. – diverse variante standardizate de ISO)

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)
 – 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

Constante definite în fișierele antet standard

limits.h: valori minime cerute de standard

```
SHRT_MIN, INT_MIN    -32767          SHRT_MAX, INT_MAX    32768
LONG_MIN             -2147483647       LONG_MAX             2147483647
USHRT_MIN, UINT_MIN  65535           ULONG_MAX            4294967295
```

Obs: pe gcc/i386/Linux, int are aceleași dimensiuni ca și long

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

```
FLT_DIG    6          DBL_DIG    15 (min. 10) /* precizie zecimala */
FLT_MIN    1.17549435e-38F (max. 1E-37)
FLT_MAX    3.40282347e+38F (min. 1E+37)
FLT_EPSILON 1.19209290e-07F (max. 1E-5) /* nr.min. cu 1+eps > 1 */
DBL_MIN    2.2250738585072014e-308 (max. 1E-37)
DBL_MAX    1.7976931348623157e+308 (min. 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})
 Dacă se calculează factorialul ca întreg, va da 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 → 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) < epsilon (fabs: val. absolută, în math.h)
 FLT_EPSILON (DBL_EPSILON) în float.h: cel mai mic x cu $1 + x > 1$

Dimensiunea tipurilor

Operatorul sizeof returnează numărul de octeți de memorie ocupați de un tip de date (sau o variabilă sau constantă):

```
#include <stdio.h>
void 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));

    printf("float\t\t%d\n", sizeof(float));
    printf("double\t\t%d\n", sizeof(double));
    printf("long double\t\t%d\n", sizeof(long double));
}
```

Caracterul '\t' (tab) sare la alinierea următoare (tipic: 8 caractere)

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)

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

(cifrele, respectiv literele ocupă spațiu continuu în tabela de caractere)

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

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 ≠ 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: && (SI), 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)

În expresii, operandii 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)
- operandii de tipuri de rang inferior lui int (char, short) sunt convertiți la tipurile int sau unsigned (după semn)
- dacă ambii operandi 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, operandii se convertesc la tipul fără semn corespunzător operandului care are tip cu semn

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

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

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 = 38400, usd_rol = 32700;
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; /* 1.17... */
int n; sqrt((double)n); /* double sqrt(double) in math.h */
```

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

– numără caracterele din șirul s în variabila i

```
for (i = 0; s[i] != '\0'; i++); /* sirul se termina cu '\0' */
sau, cu un test implicit de valoare nonzero, și preincrement:
for (i = -1; s[++i]; ); /* corpul lui for este vid */
```

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

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

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

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

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

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

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

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 => i e considerat negativ */
if (i > u) printf("%d > %u\n", i, u);
/* tiparește: -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)

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

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 dinaintea de atribuire

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

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

Orice expresie are o *valoare*, definită prin înțelesul operatorilor.
Atribuirile au și un *efect lateral*: modifică valoarea expresiei atribuite
Exemplu: ++i și i++ au același efect lateral (incrementează pe i) dar returnează valori diferite (valoarea deja incrementată / încă nu)

ATENȚIE: În C, ordinea de evaluare a operandilor unei expresii nu e specificată (depinde de implementare). Excepții: && || ?: ,
⇒ o expresie care conține mai mulți operatori cu efect lateral poate avea rezultat / efect nedeterminat. Exemple *erone*:

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

(argumentele unei funcții se pot evalua în orice ordine)

```
while (s[i] < s[i+1]); /* in ordine crescătoare ? */
```

(dar dacă s[i+1] e evaluat întâi, îl comparăm cu el însuși)

Atenție la efectele laterale, nu scrieți cod compact cu orice preț!

Utilizarea și Programarea calculatoarelor 2. Curs 3

Marius Minea

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

Exemple:

```
n & 0xF are ultimii 4 biți (mai puțin semnificativi) la fel ca n, restul 0
  (pentru n fără semn, echivalent cu n % 16)
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
  (dacă n pozitiv, echivalent cu n-1 pt. n impar, n+1 pt. n par)
~0 == -1 (toți biții pe 1, indiferent de dimensiunea în octeți)
~0xf are ultimii 4 biți pe 0 și restul pe 1 (indiferent de dimensiune)
```

Operatori pe biți (cont.)

$n \ll 3$ are biții lui n deplasați 3 poziții la stânga, și ultimii 3 biți 0
 $n \gg 2$ are biții lui n deplasați 2 poziții la dreapta, și primii 2 biți 0
 (sau pentru *signed*, 1 (bitul de semn), în funcție de arhitectură)
 \ll și \gg : ca și înmulțiri/împărțiri cu puterile lui 2, uneori mai rapide
 (dacă \gg inserează la stânga biți de semn, e valabil și pt. nr. negative)

Operatori compuși de atribuire (pt. cei binari): $\&=$ $|=$ $\^{}=$ $\ll=$ $\gg=$

Exemple: extragerea unei porțiuni din reprezentarea unui număr:
 creem o *maskă* (un tipar) în care biții respectivi sunt pe 0 (sau 1)
 $\sim 0 \ll k$ are ultimii k biți pe 0, restul pe 1
 $\sim(\sim 0 \ll k)$ are ultimii k biți pe 1, restul pe 0
 $\sim(\sim 0 \ll k) \ll p$ are k biți pe 1, începând de la bitul p , și restul 0
 $(n \gg p) \& \sim(\sim 0 \ll k)$ are pe ultimele poziții cei k biți ai lui n începând
 cu bitul p , și în rest 0
 $n \& (\sim(\sim 0 \ll k) \ll p)$ are cei k biți începând cu bitul p la fel ca ai lui
 n , și restul biturilor pe 0

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");
```

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 ↓)	Asociativitate
() [] -> .	→
! ~ ++ -- - (tip) * & (pt.adrese) sizeof	←
* / %	→
+ -	→
<< >>	→
< <= > >=	→
== !=	→
&	→
^	→
	→
&&	→
	→
? :	←
= += -= etc.	←
,	→

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 lo+8 biți
```

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

```
if ((n & mask) == val) { /* testeaza bitii selectati de mask */ }
dar: n & mask == val face ȘI cu booleanul mask == val (0 sau 1)
```