

Considerații teoretice

Formatori:

Tutor: [Stângaciu Valentin](#)  

Tutor: [Belu Claudiu-Marcel](#)  

+3

Data de începere a cursului:

 25.09.2023

 [Utilizatori înscrși](#)

 [Calendar](#)

 [Note](#)

 [Cursurile mele](#) [S1-L-AC-CTIRO1-PC](#) [Laborator 2: Reprezentarea informației în calculat...](#) [Considerații teoretice](#)

Considerații teoretice

Variable

De multe ori este necesar să se memoreze anumite valori, care nu se știu de la început sau care sunt calculate pe parcurs. Evident, pentru memorarea acestor valori va trebui desemnat un anumit loc în memorie, unde ele vor fi depuse. Deoarece se permite ca într-o asemenea zonă de memorie să se memoreze oricând o nouă valoare, noua valoare înlocuind-o pe cea veche, acestea se numesc **variabile**. Deci, în C, o variabilă este un nume care identifică o zonă de memorie ce conține o valoare.

În limbajul C, o variabilă are un tip bine definit și ea poate conține doar valori specifice aceluia tip. În funcție de tipul lor, unele variabile au nevoie de mai multă sau de mai puțină memorie pentru memorarea valorii lor. Totodată, operațiile care pot fi realizate cu anumite tipuri diferă de cele care pot fi realizate cu alte tipuri. Din aceste motive, compilatorului trebuie să i se specifice fiecare variabilă ce tip are. Sintaxa de definire a unei variabile este:

```
tip nume_variabila;
```

Se poate remarca faptul că tipul întotdeauna se scrie înaintea numelui variabilei, iar la sfârșit se pune punct și virgulă (;). Rămâne în sarcina compilatorului să aloce o zonă de memorie care va fi accesată prin intermediul acestui nume.

```
#include <stdio.h>
int main() {
    int a;
    a = 10; // @1
    printf("a=%d", a);
    return 0;
}
```

În acest exemplu s-a declarat variabila *a*, având tipul **int**. Cuvântul **int** este prescurtarea de la integer și specifică numere întregi cu semn. De obicei, pentru compilatoarele actuale pe Windows și Linux, în variabile de tip **int** se pot ține valori în intervalul [-2 147 483 648, 2 147 483 647]. La compilatoarele mai vechi sau la cele pentru microcontrolere, **int** poate lua valori din intervalul [-32 768, 32 767].

Pentru numere reale (cu virgulă) se folosește tipul de date **float**. Tipul **float** conține numere reale în intervalul [$\pm 1.18 \times 10^{-38}$; $\pm 3.4 \times 10^{38}$], cu o precizie de 7 cifre zecimale (după virgula zecimală). Când se dorește precizie mai mare, se poate folosi tipul **double**, care conține numere reale din intervalul [$\pm 2.23 \times 10^{-308}$; $\pm 1.80 \times 10^{308}$], cu o precizie de 16 cifre zecimale.

Atenție: în C, pentru scrierea cifrelor zecimale se folosește **punctul** (.), nu **virgula** (,). De exemplu, numărul PI se va scrie 3.14, nu 3,14.

La instrucțiunea *@1* se poate vedea cum variabilei *a* i-a fost atribuită valoarea 10 (s-a stocat valoarea 10 în locația de memorie desemnată de *a*). După orice atribuire trebuie pus **punct și virgulă** (;).

Pentru afișarea valorii unei variabile se folosește în cadrul șirului de caractere din *printf* un așa-numit „placeholder”, care va fi înlocuit la afișare chiar cu valoarea respectivei variabile. Pentru fiecare placeholder, după șirul de caractere va trebui să urmeze variabila care va înlocui placeholderul respectiv, de la stânga la dreapta. Un placeholder începe cu caracterul procent (%), urmat fără spațiu de o literă. Iată în continuare câteva placeholder:

Placeholder Folosire

%d	<i>decimal</i> - afișează o variabilă de tip int
%f	<i>floating point</i> - afișează a variabilă de tip float sau double și tipărește întotdeauna și o parte zecimală, deși valoarea poate fi întreagă
%e	<i>exponential</i> - afișează a variabilă de tip float sau double în format științific (mantisă+exponent). Este util în special pentru numere foarte mari sau foarte mici. Exemplu: -1.327e9
%g	<i>general</i> - afișează o variabilă de tip float sau double într-un fel cât mai familiar pentru utilizator: numerele întregi le afișează fără zecimale, iar numerele foarte mari sau foarte mici le afișează cu exponent.
%%	afișează chiar caracterul procent (%)

```
// afisarea unui numar intreg
int numar_intreg;
numar_intreg = 10;
printf("numarul intreg este %d\n", numar_intreg);
// afisarea unui numar real
float numar_real;
numar_real = 10.52;
printf("numarul real este %f\n", numar_real);
```

La primul *printf*, după șirul de caractere se poate observa că urmează o virgulă și după ea variabila a cărei valoare va înlocui placeholderul din cadrul șirului de caractere (%d). Dacă am fi avut de afișat două valori, s-ar fi folosit două placeholderuri și după șirul de caractere ar fi urmat două variabile, separate între ele prin virgulă.

```
// afisarea mai multor numere cu un singur printf
int marker_cantitate, whiteboard_cantitate; // se pot declara mai multe variabile de același tip, separate prin ,
float marker_pret, whiteboard_pret;
marker_cantitate = 15;
whiteboard_cantitate = 2;
marker_pret = 4.83;
whiteboard_pret = 95.99;
printf("Vom achizitiona pentru laborator %d markere in valoare de %f lei fiecare si %d whiteboard-uri in valoare de %f lei fiecare.", marker_cantitate, marker_pret, whiteboard_cantitate, whiteboard_pret);
```

Functia *printf* (*print formatted*) își trage numele tocmai de la folosirea acestor placeholderuri.

Citirea unor valori de la tastatură

Dacă se dorește ca utilizatorul să poată introduce noi valori de la tastatură, acestea pot fi citite folosind funcția *scanf* (*scan formatted*), ca în exemplul de mai jos:

```
// citire de la tastatura cu scanf
int k;
printf("Introduceti k: "); // @1
scanf("%d", &k); // @2
printf("Ati introdus valoarea: %d\n", k);
```

Instrucțiunea @1 afișează un text explicativ pentru utilizator. Nu este obligatoriu, dar este recomandat să se utilizeze asemenea texte, pentru ca utilizatorul să știe ce i se cere. Instrucțiunea @2 citește efectiv valoarea de la tastatură. Funcția *scanf* este foarte asemănătoare cu *printf*, în sensul că folosește aproape aceleași placeholderuri (în acest caz, %d pentru a se citi o valoare de tip **int**) și, la fel ca *printf*, cere ca fiecare variabilă citită să fie separată cu virgulă. La fel ca la *printf*, pentru fiecare variabilă vom avea câte un placeholder. Deoarece funcția *scanf* nu afișează nicio valoare pe ecran, șirul ei de caractere va conține doar placeholderuri.

```
// citirea mai multor numere de diferite tipuri
int marker_cantitate, whiteboard_cantitate;
float marker_pret, whiteboard_pret;
printf("Cate markere sunt necesare: ");
scanf("%d", &marker_cantitate);
printf("Pretul unui marker: ");
scanf("%f", &marker_pret);
printf("Cate whiteboards sunt necesare, urmat de pretul unui whiteboard: ");
scanf("%d%f", &whiteboard_cantitate, &whiteboard_pret);
printf("Vom achizitiona pentru laborator %d markere in valoare de %f lei fiecare si %d whiteboard-uri in valoare de %f lei fiecare.", marker_cantitate, marker_pret, whiteboard_cantitate, whiteboard_pret);
```

Spre deosebire de *printf*, care folosește %g, %f și %e atât pentru **float**, cât și pentru **double**, *scanf* folosește %lg, %lf și %le (l = long) pentru citirea valorilor de tip **double**.

Baze de numerație

O bază de numerație este numărul de unități necesar pentru a se obține o unitate de ordin imediat superior. De exemplu, în baza de numerație 10, sunt necesare 10 unități simple pentru a se obține o unitate de zeci. Cifrele disponibile într-o bază b sunt în intervalul $0..b-1$. De exemplu, pentru baza 10, cifrele disponibile sunt $0..9$. Orice sistem de numerație pozițional (în care pozițiile cifrelor denotă diverse ranguri, de exemplu unități, zeci, sute, mii, ...) se bazează implicit pe folosirea bazelor de numerație.

Deoarece inițial omul a numărat pe degete, cel mai comod a fost să folosească baza de numerație 10. După ce număra folosind toate degetele, ținea minte că a făcut o numărătoare completă (deci adăuga o unitate la rangul zecilor) și reîncepea să numere unitățile.

Explicitând baza de numerație b în care este scris, un număr format din n cifre $c_{n-1}c_{n-2}...c_1c_0$ se poate reprezenta astfel:

$$c_{n-1}c_{n-2}...c_1c_0 = c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + ... + c_1 \cdot b^1 + c_0 \cdot b^0$$

Exemplu: numărul 4905 în baza 10 se poate scrie: $4905 = 4 \cdot 10^3 + 9 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0$. Constatăm că de fapt atunci când citim un număr, noi spunem pe rând, de la stânga la dreapta, numărul de unități (cifrele) corespunzătoare diferitelor ranguri: 4 *mii*, 9 *sute*, 0 *zeci* (nu îl rostim fiindcă nu contribuie la număr), 5.

Baza de numerație 2

Deoarece funcționarea calculatoarelor se bazează pe fenomene electromagnetice, cel mai simplu a fost să se implementeze memoria acestora folosind fenomene de genul prezența sau absența unui curent sau a unei tensiuni. Astfel, se poate nota prezența tensiunii cu 1 și absența sa cu 0. O unitate de informație care poate fi doar 0 sau 1 se numește **bit**. Grupând 8 biți (en: *bits*) obținem un **octet** (en: *byte*). Deoarece fiecare cifră din numărul memorat de un calculator poate avea doar două valori, devine natural pentru calculatoare să folosească baza de numerație 2 sau, altfel spus, să folosească sistemul de numerație *binar*. Un sistem electronic care folosește doar semnale care au semnificație numerică (ex: 0 și 1) se numește *digital*, deoarece transferă digiți (cifre) de informație. Dacă de exemplu codificăm 1 ca fiind 5V (volți) și 0 ca fiind 0V, într-un sistem digital vom avea doar aceste două tensiuni, în intervale specificate de toleranță.

Exemplu: Un octet cu biții având valorile 10010101 reprezintă numărul 149 în baza 10 ($1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 128 + 16 + 4 + 1 = 149$).

În anumite domenii din informatică, precum programarea microcontrolelor (microcontrollers, ex: Arduino) sau interacțiunea cu dispozitive periferice (ex: roboți, imprimante 3D, ...) se folosește foarte des baza de numerație 2. În aceste domenii se preia/trimite semnalul (biți de informație) de la/către diverși pini ai cipurilor respective și programatorul trebuie să implementeze comportamentul fiecărui pin.

În asemenea domenii și în general pe partea de hardware, lucrând mult cu biți, vom ajunge să memorăm puterile uzuale ale lui 2 și să lucrăm în mod firesc cu ele. La început este util să avem un tabel cu aceste puteri, pentru a nu le calcula de fiecare dată.

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2 ^N	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536

Dacă avem un număr N de biți, vom putea reprezenta pe ei 2^N valori, în intervalul 0..2^N-1. De exemplu pe 3 biți vom putea avea 2³=8 valori în intervalul 0..7: 000, 001, 010, 011, 100, 101, 110, 111.

Cum s-a văzut din exemplul de mai sus, pentru a se transforma un număr dintr-o bază oarecare în baza 10, se scrie numărul sub formă pozițională cu baza explicită și apoi se fac calculele în baza 10. Pentru a ști în ce bază scriem un număr, vom postfixa acel număr cu baza sa, dată ca indice.

$$1100_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 12_{10}$$

Dacă știm puterile lui 2, această transformare este simplă, procedând astfel: luăm pe rând cifrele numărului, de la dreapta la stânga și, de fiecare dată când avem un 1, adunăm la numărul final puterea lui 2 corespunzătoare acelei poziții, cea mai din dreapta poziție fiind 0.

Pentru 1100₂, la poziția unităților (0) este 0, deci această poziție nu contribuie la număr. Pentru poziția 1 iarăși avem 0, deci trecem și peste ea. Pentru poziția 2 avem 1, deci adunăm 2²=4 la numărul final (inițial 0), rezultând 4. În final, pentru poziția 3 avem 1, deci adunăm și 2³=8 la rezultat: 8+4=12. Acesta este rezultatul final.

Transformarea din baza 10 în baza 2 se face prin împărțiri succesive, până când obținem câtul 0. Împărțirea se va efectua în numere întregi (nu cu virgulă), obținând un cât și un rest. Numărul va fi format din resturile parțiale, în sens invers de cum le-am obținut. Pentru a completa până la multiplu de octet, se pun în față 0-uri ne semnificative.

Exemplu: transformarea numărului 13₁₀ în baza 2:

13 : 2 = 6 | 1 - cel mai puțin semnificativ bit (LSB=Least Significant Bit)

6 : 2 = 3 | 0

3 : 2 = 1 | 1

1 : 2 = 0 | 1 - cel mai semnificativ bit (MSB=Most Significant Bit)

preluând resturile împărțirilor, de jos în sus: 13₁₀ = 1101₂ = 00001101₂ (completat cu 0-uri ne semnificative)

Bitul cel mai din stânga al unui număr se numește *bitul cel mai semnificativ* (MSB=Most Significant Bit), deoarece el singur contribuie cu ½ din valoarea întregului număr. Bitul cel mai din dreapta al unui număr se numește *bitul cel mai puțin semnificativ* (LSB=Least Significant Bit), deoarece el contribuie cel mai puțin la valoarea numărului, doar cu o unitate.

Bazele de numerație 16 (hexazecimal) și 8 (octal)

De multe ori devine destul de laborios să se scrie și să se opereze pe toți biții unui octet, deoarece avem de scris 8 valori și cu fiecare dintre aceste valori avem de făcut diverse operații. Cu atât mai mult, dacă dorim să reprezentăm numere pe 2, 4 sau 8 octeți, va trebui pentru fiecare număr să scriem respectiv câte 16, 32 sau 64 de valori. Pentru a reduce acest efort, putem considera baze de numerație mai mari, în care o singură cifră să corespundă la mai mulți biți.

Ținând cont că pe 4 biți putem reprezenta 2⁴=16 valori, dacă am reprezenta numerele în baza 16 (hexazecimal), fiecare număr din această bază va corespunde la 4 biți. Pentru a reprezenta un octet avem nevoie doar de 2 cifre hexazecimale.

Cifrele hexazecimale sunt:

Hexazecimal	Zecimal	Binar
0	0	0000
1	1	0001
2	2	0010

3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
a, A	10	1010
b, B	11	1011
c, C	12	1100
d, D	13	1101
e, E	14	1110
f, F	15	1111

Deoarece în alfabet nu avem cifre decât pentru baza 10 (0...9), prin convenție vom folosi pentru cifrele 10...15 din hexazecimal primele litere din alfabet (a...f), indiferent dacă sunt mari sau mici.

Exemplu: Numărul $7af_{16} = 7 \cdot 16^3 + 10 \cdot 16^2 + 15 \cdot 16^1 + 2 \cdot 16^0 = 31474_{10}$

Ținând cont că unei cifre în hexazecimal îi corespund 4 biți, transformarea din hexazecimal în binar este foarte simplă: fiecare cifră hexazecimală va fi înlocuită cu 4 biți.

Exemplu: $7af_{16} = 0111\ 1010\ 1111\ 0010_2$

Transformarea din binar în hexazecimal este la fel de simplă: facem perechi de câte 4 biți, de la dreapta la stânga și înlocuim fiecare pereche cu cifra hexazecimală corespunzătoare.

Exemplu: $0010\ 1101\ 1001\ 0110_2 = 2d96_{16}$

Tot pentru simplificarea scrierii, dar mult mai rar, se folosește și baza 8 (octal). În această bază avem cifre de la 0...7 și fiecare cifră în octal corespunde exact la 3 biți.

Exemplu: $715_8 = 7 \cdot 8^2 + 1 \cdot 8^1 + 5 \cdot 8^0 = 461_{10} = 111\ 001\ 101_2$

Atât în hexazecimal cât și în octal, operațiile aritmetice, precum și transformările între diverse baze, rămân identice cu ce s-a prezentat anterior.

Tipuri de date întregi în C

Limbajul C pune la dispoziția programatorului o gamă bogată de tipuri de date întregi, cu sau fără semn, pentru ca programatorul să-și poată alege tipurile cele mai potrivite pentru aplicația sa. Dimensiunea în octeți a acestor tipuri nu este standardizată, astfel încât ea poate să varieze în funcție de compilator, platforma hardware etc.

Un tip de date fără semn se declară punând cuvântul **unsigned** în fața tipului întreg. Tipurile de date **unsigned** se afișează cu *printf* folosind placeholderul **%u**, în loc de **%d** care este pentru întregi. Pentru tipul **long** se consideră **int** ca fiind implicit, deci nu mai trebuie pus **int** după **long** (**long** este echivalent cu **long int**).

În tabelul de mai jos s-au reprezentat toate tipurile întregi din C. Cu cifre bold s-au marcat valorile mai uzuale, pe care le implementează și compilatoarele de C din laborator. Intervalele de valori date corespund acestor valori mai uzuale.

Tip	Dimensiune (octeți)	Interval de valori	Placeholder printf și scanf (simplificat, doar pentru printf)
char	1	-128...127 sau 0..255 (în C programatorul poate selecta tipul <i>char</i> să fie cu sau fără semn)	%hhd (%d) %c - va afișa valoarea ca pe un caracter
unsigned char	1	0..255	%hhu (%u)
signed char	1	-128...127	%hhd (%d)
short	2	-32 768...32 767	%hd (%d)
unsigned short	2	0...65 535	%hu (%u)
int	2 sau 4	-2 147 483 648...2 147 483 647	%d
unsigned int	2 sau 4	0...4 294 967 295	%u
long	4 sau 8	-2 147 483 648...2 147 483 647	%ld

unsigned long	4 sau 8	0...4 294 967 295	%lu
long long	4 sau 8	-9 223 372 036 854 775 808...9 223 372 036 854 775 807	%lld
unsigned long long	4 sau 8	0...18 446 744 073 709 551 615	%llu

În C, când scriem o constantă numerică în program (ex: 108), ea implicit este de tipul **int**. Dacă acea constantă este folosită în contexte în care este nevoie de alt tip, compilatorul ne va atenționa de o posibilă greșeală. Pentru a se evita aceasta, putem să postfixăm constantele cu sufixul **"l"** pentru a fi considerate de tip **long** sau cu sufixul **"ll"**, pentru a fi considerate de tip **long long**.

```
printf("%lld\n", 7); // atenționare: %lld cere un argument de tipul long long, iar 7 este considerat de tipul int
printf("%lld\n", 7ll); // corect: acum constanta 7 este considerată de tip long long
```

Pe lângă baza 10, limbajul C ne permite să scriem un număr direct în bazele cele mai uzuale (2, 8, 16), conform următoarei sintaxe:

Baza Exemple	Sintaxa	Afișare cu printf
2 0b 11101 0b 1011 0b 10	numărul se prefixează cu 0b	nu există placeholder
8 071 0262 0117	numărul se prefixează cu 0 . Atenție: în C un număr întreg care începe cu 0, se consideră ca fiind în baza 8, nu în baza 10.	%o
10 1729 1 423	numărul trebuie să înceapă cu o cifră între 1...9	%d
16 0xBAF 0x72 0x2a1f	numărul se prefixează cu 0x	%x - litere mici %X - litere mari

În exemplul de cod de mai jos, mai multe variabile de tip **int** primesc valori într-o anumită bază și sunt apoi afișate în diverse feluri, folosind **printf**.

```
#include <stdio.h>

int main() {
    // număr format din cifre de la 0 la 10, în baza 10
    int decimal_num = 91;

    // număr format din 0 și 1, în baza 2, începe cu 0b
    // atenție, sintaxa aceasta nu este suportată de toate compilatoarele
    int binary_num = 0b1011011;

    // număr format din cifrele de la 0 la 7, în baza 8, începe cu 0
    int octal_num = 0133;

    // număr format din cifre de la 0 la 9 și litere de la A la F, în baza 16, începe cu 0x
    int hexadecimal_num = 0x5B;

    printf("%d %d %d %d\n", decimal_num, binary_num, octal_num, hexadecimal_num);
    printf("%o %o %o %o\n", decimal_num, binary_num, octal_num, hexadecimal_num);
    printf("%x %x %x %x\n", decimal_num, binary_num, octal_num, hexadecimal_num);

    return 0;
}
```

Operații logice

În limbajul C, orice valoare diferită de 0 (0 întreg) și de 0.0 (0 real) este considerată ca având valoarea logică **adevărat** (1, true). Singurele valori care sunt considerate ca fiind **fals** logic (0, false) sunt 0 și 0.0.

Exemple: valori adevărate: 1, -49, 0.5, 3.14, 0.001; valori false: 0, 0.0.

Pentru a opera cu valori logice, avem la dispoziție următorii operatori:

Operator	Efect
!a	negație logică
a && b	ȘI logic
a b	SAU logic

Acești operatori returnează o valoare întreagă nenulă (poate fi diferită de 1) pentru **adevărat** logic și returnează 0 pentru **fals** logic. Deși tabelele de adevăr sunt la fel pentru operatorii logici ca și pentru cei pe biți, comportamentul lor este diferit: **operatorii logici** consideră operanzii ca fiind fiecare câte o singură valoare logică (câte un un bit); **operatorii pe biți** operează individual, în paralel, asupra biților corespondenți din cei doi operanzi. Următorul exemplu ilustrează această diferență:

a	0	1	1	0	0	1	0	1
b	1	0	0	0	1	0	1	0
a & b	0	0	0	0	0	0	0	0
a && b	0	0	0	0	0	0	0	1

Operația $a \& b$ a avut ca rezultat 0, deoarece nicăieri nu au fost biți corespondenți simultan pe 1. Operația $a \&\& b$ a avut ca rezultat 1 (sau orice altă valoare *true*), deoarece luat ca întreg a este *true* și la fel și b este *true*.

Operatorii relaționali (de comparație)

În C există următorii operatori relaționali. Pentru a-i memora mai ușor, semnul = (egal) este întotdeauna pus după semnele < sau >. Acești operatori returnează *adevărat* (o valoare diferită de 0, nu neapărat 1) ca valoare de tip **int** dacă cei doi operanzi sunt în relația specificată, altfel returnează *fals* (0).

Operator	Relație
$a == b$	egal
$a != b$	inegal
$a < b$	mai mic
$a <= b$	mai mic sau egal
$a >= b$	mai mare sau egal
$a > b$	mai mare

Atenție: una dintre cele mai frecvente erori de codare în C este folosirea *atribuirii* (=) în loc de test de *egalitate* (==) și invers. Din punct de vedere sintactic nu este o greșală (eventual compilatorul va emite doar o atenționare) și atunci uneori această greșală este mai greu de depistat.

Exemplu: se cere să se citească 3 numere întregi de la tastatură, a , b , x și să se afișeze o valoare de *adevărat* sau de *fals* dacă x este sau nu în interiorul intervalului închis $[a, b]$.

```
int a,b,x;
printf("a=");scanf("%d",&a);
printf("b=");scanf("%d",&b);
printf("x=");scanf("%d",&x);
printf("rezultat=%d",a<=x && x<=b);
```

Instrucțiunea de decizie: if

De multe ori este necesar să se continue execuția unui algoritm în funcție de o condiție (de exemplu rezultatul unei comparații), care se poate sau nu îndeplini. Pentru aceasta se folosește instrucțiunea **if**, care are următoarea sintaxă:

```
if(condiție) {
    instrucțiuni_pentru_condiție_adevărată;
} else {
    instrucțiuni_pentru_condiție_falsă;
}
```

Această secvență de program se citește astfel:

DACĂ (**if**) condiție este adevărată, se execută *instrucțiuni_pentru_condiție_adevărată*,
ALTFEL (**else**) se execută *instrucțiuni_pentru_condiție_falsă*.

De exemplu, pentru calculul valorii maxime dintre două numere, se poate folosi programul de mai jos:

```
#include <stdio.h>

int main() {
    int x, y, max;
    printf("x = "); scanf("%d", &x);
    printf("y = "); scanf("%d", &y);
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    printf("Maximul este: %d", max);
    return 0;
}
```

Partea **else** este opțională (poate să lipsească). Acoladele se pun întrucât compilatorul are nevoie să știe unde începe și unde se termină zona de cod care se execută în cazul în care condiția este adevărată (ramura **if**) sau falsă (ramura **else**). Acoladele pot însă să lipsească dacă avem o singură instrucțiune pe ramura respectivă, astfel încât puteam scrie codul de mai sus și astfel:

```
#include <stdio.h>

int main() {
    int x, y, max;
    printf("x = "); scanf("%d", &x);
    printf("y = "); scanf("%d", &y);
    if (x > y)
        max = x;
    else
        max = y;
    printf("Maximul este: %d", max);
    return 0;
}
```

În continuare este un exemplu în care avem nevoie de mai multe instrucțiuni pentru ramura **if**. În acest caz, suntem obligați să punem acolade. Tot ceea ce este între acolade este tratat ca fiind o singură instrucțiune. Pe ramura **else** în schimb este necesară o singură instrucțiune, deci acoladele pot lipsi. Cu toate acestea, pentru simetrie, ceea ce duce la creșterea lizibilității codului, vom pune acolade și pe ramura **else**.

```
#include <stdio.h>

int main() {
    float x, y, catul;
    printf("x = "); scanf("%g", &x);
    printf("y = "); scanf("%g", &y);
    if (y != 0) {
        catul = x / y;
        printf("x/y=%g", catul);
    } else {
        printf("Nu se poate face impartirea la zero!");
    }
    return 0;
}
```

Este bine să indentăm (deplasăm) la dreapta toate instrucțiunile care fac parte din corpul altei instrucțiuni (în acest caz **if**), deoarece devine vizual mai simplu să citim codul.

Atenție: una dintre cele mai frecvente greșeli este să nu se pună între acolade instrucțiunile care depind de **if** sau de **else**-ul asociat. În acest caz, chiar dacă aceste instrucțiuni sunt indentate la dreapta și se creează vizual impresia că depind de **if**, ele totuși se vor executa independent de acesta, ca în exemplul următor:

```
#include <stdio.h>

int main() {
    float x, y, catul;
    printf("x = "); scanf("%g", &x);
    printf("y = "); scanf("%g", &y);
    if (y != 0)
        catul = x / y;
        printf("x/y=%g", catul); // GREȘIT: chiar dacă este indentată, această linie nu este condiționată de if. Ea se va
executa întotdeauna, indiferent de rezultatul condiției
    return 0;
}
```

Cum s-a discutat la operatori, în C o expresie se consideră ca fiind adevărată dacă ea are orice valoare diferită de 0. Doar valoarea 0 se consideră ca fiind falsă. Putem astfel să folosim următoarea secvență pentru a testa dacă un număr este 0 sau nu:

```
if(k)
    printf("k nu este 0\n");
else
    printf("k este 0\n");
```

Operatorii relaționali sunt: <, <=, >=, >, ==, !=. Reamintim faptul că o greșeală foarte frecventă este folosirea operatorului de atribuire (=) în loc de cel de comparație (==) și invers. „a=b” înseamnă că i se atribuie lui a valoarea lui b, pe când „a==b” înseamnă că se compară a cu b pentru a se vedea dacă acestea sunt egale.

Dacă avem mai multe condiții, le putem combina folosind operatorii logici: && || !. Nu se vor folosi operatorii pe biți pentru operații logice, chiar dacă au aceleași tabele de adevăr, deoarece rezultatele pot fi neașteptate și erorile sunt greu de depănat.

Un exemplu de folosire al operatorilor logici este următorul: într-o aplicație bancară, un client poate intra în contul său folosind un număr de cont (număr întreg) și o parolă (tot număr întreg). Să se scrie un program care testează dacă utilizatorul a introdus corect ambele informații, valorile cerute fiind cont=1546 și parola=9746.

```
#include <stdio.h>
int main() {
    int cont, parola;
    printf("cont: "); scanf("%d", &cont);
    printf("parola: "); scanf("%d", &parola);
    if (cont == 1546 && parola == 9746) {
        printf("Autentificare reușită!");
    } else {
        printf("Autentificare eșuată!");
    }
    return 0;
}
```

Atenție: în C, scrierea în serie a operatorilor relaționali de obicei nu are același efect ca în matematică. De exemplu, în matematică, dacă dorim să scriem că x aparține intervalului (1, n), putem scrie „ $1 < x < n$ ”, ceea ce se citește ca „ $1 < x$ ȘI $x < n$ ”. În C, expresia respectivă se citește ca „ $(1 < x) < n$ ”, adică „**rezultatul lui $1 < x$ (care poate fi de exemplu 1 sau 0) se compară cu n**”, deci în final nu se va compara pe x cu n, ci un rezultat intermediar cu n. Corect este să scriem „ $1 < x \&\& x < n$ ”.

Operatorul condițional (ternar)

if este o instrucțiune și prin urmare nu returnează o valoare. Din acest motiv, **if** nu poate fi folosit în cadrul unei expresii. Pentru a se returna în cadrul unei expresii valori în funcție de rezultatul unei condiții, se folosește operatorul condițional: **?:**. Sintaxa acestui operator este:

condiție ? valoare_true : valoare_false

Efectul operatorului condițional este următorul: dacă *condiție* este adevărată, se returnează *valoare_true*, altfel se returnează *valoare_false*. Operatorul condițional se poate folosi oriunde într-o expresie: în locul unei variabile sau constante, ca argument pentru apeluri de funcții, etc. Dacă se folosește în expresii mai complicate, este util să se pună între paranteze, pentru a se evita unele erori care țin de precedența operatorilor. Se pot imbrica mai mulți operatori condiționali unul în celălalt.

Exemplu: programul anterior, de obținere a maximului dintre două numere, se poate rescrie astfel, folosind operatorul condițional:

```
#include <stdio.h>
int main() {
    int x, y, max;
    printf("x = "); scanf("%d", &x);
    printf("y = "); scanf("%d", &y);
    printf("Maximul este: %d", x > y ? x : y);
    return 0;
}
```


Ordinea (precedența) și asociativitatea operatorilor

La fel ca și în matematică, operatorii din C se execută într-o anumită ordine. De exemplu, în expresia $a+b*7$, înmulțirea se execută prima, deși apare după adunare. Această ordine de execuție se numește **precedența** operatorilor. Întotdeauna se vor executa primii operatorii cu precedența cea mai mare. Pentru a se modifica ordinea de evaluare a operatorilor, se folosesc paranteze.

Totodată, pentru operatori se definește și **asociativitatea** acestora. Această proprietate se referă la ordinea de execuție (de la stânga la dreapta → asociativitate *stângă*, sau de la dreapta la stânga → asociativitate *dreaptă*) atunci când avem o secvență de mai mulți operatori cu aceeași precedență. De exemplu, expresia $18/3/2$ dacă împărțirea ar avea asociativitate stângă s-ar scrie $(18/3)/2 \rightarrow 3$, iar dacă împărțirea ar avea asociativitate dreaptă, s-ar scrie $18/(3/2) \rightarrow 18$.

În tabelul de mai jos se află precedența și asociativitatea operatorilor din C, de la precedența cea mai mare la cea mai mică. Toți operatorii dintr-o celulă au aceeași precedență. Unii dintre acești operatori vor fi predați ulterior, în următoarele laboratoare.

Operator	Nume	Asociativitate
++ --	incrementare/decrementare postfix	stângă
()	apel de funcție	
[]	indexare în vector	
.	accesul membrului unei structuri	
->	accesul membrului unei structuri folosind un pointer	
++ --	incrementare/decrementare prefix	dreaptă
+ -	plus sau minus unar	
! ~	NOT logic și pe biți	
(tip)	conversie de tip (cast)	
*	indirectare (dereferențiere)	
&	adresa	
sizeof	dimensiunea în octeți	
* / %	înmulțire, împărțire, modulo	stângă
+ -	adunare, scădere	stângă
< < > >	deplasare pe biți la stânga și la dreapta	stângă
< < = > > =	mai mic, mai mic sau egal, mai mare, mai mare sau egal	stângă
= = ! =	egal, inegal	stângă
&	ȘI pe biți	stângă
^	SAU EXCLUSIV pe biți	stângă
	SAU pe biți	stângă
&&	ȘI logic	stângă
	SAU logic	stângă
?:	operatorul condițional (ternar)	dreaptă
=	atribuire	dreaptă
+ = - =	atribuire cu sumă, diferență	
* = / = % =	atribuire cu produs, împărțire, modulo	
< < = > > =	atribuire cu deplasare pe biți la stânga, la dreapta	
& = ^ = =	atribuire cu ȘI, SAU EXCLUSIV, SAU pe biți	
,	virgulă (secvență)	stângă

În general este bine, mai ales dacă lucrăm în echipă cu alți programatori, să punem paranteze pentru a se evidenția ordinea operațiilor mai puțin folosite, chiar dacă aceste paranteze nu sunt necesare. Ele măresc în schimb lizibilitatea programului: $((a' <= c \& \& c <= 'z')) || ((A' <= c \& \& c <= 'Z'))$

În special când se combină operații pe biți cu operații aritmetice, trebuie avută multă atenție la ordinea operațiilor, altfel putând rezulta erori greu de depistat. De exemplu, dacă un programator rescrie expresia $a+b*2$ ca $a+b<<1$, pentru a-i optimiza viteza, programul rezultat va fi greșit, deoarece această din urmă expresie de fapt este echivalentă cu $(a+b)<<1$. Corect ar fi trebuit să se scrie $a+(b<<1)$.