

# Expresii în C/C++

## (II)

<https://docs.microsoft.com/en-us/cpp/cpp/expressions-cpp>

### 5 B. Operatori unari prefixați.

Niv	Nr	Simbol	Operator / Operație	Apelare	As.	RV	LV	SE	EO
III	1	<b>++ --</b>	incrementare/decrementare	<b>++c</b>	<<<<	cto	L*	SE	-
	2	<b>&amp;</b>	op. adresă	<b>&amp;l-valoare</b>	<<<<	ptr	-	-	-
	3	<b>*</b>	op. țintă	<b>*pointer</b>	<<<<	cto	Lv	-	-
	4	<b>+</b>	păstrare semn	<b>+l-valoare</b>	<<<<	cto	-	-	-
	5	<b>-</b>	schimbare semn	<b>-l-valoare</b>	<<<<	cto	-	-	-
	6	<b>~</b>	negare pe biți	<b>~expresie</b>	<<<<	int	-	-	-
	7	<b>!</b>	negare logică	<b>!expresie</b>	<<<<	bool	-	-	-
	8	<b>sizeof</b>	aflarea spațiului de stocare	<b>sizeof (tip)</b>	<<<<	int	-	-	-
	9	<b>(tip)</b>	op. de conversie explicită	<b>(tip) expresie</b>	<<<<	<i>tip</i>	-	-	-

B1. *Operatorii de incrementare/decrementare prefixați* au ca rezultat valoarea numerică a operandului incrementată/decrementată cu o unitate. Modificarea valorii operandului, care trebuie să aibe l-valoare, survine într-un moment neprecizat între începutul evaluării expresiei curente și utilizarea rezultatului incrementat/decrementat.

```
#include<iostream>
using namespace std;
int main ()
{
    int i=1;
    double w=10.3;
    cout<<"i  ="<<i<<"    w="<<w<<endl;
    cout<<"i++="<<i++<<"  ++w="<<w<<endl;
    cout<<"i  ="<<i<<"    w="<<w<<endl;
    return 0;
}
/*
    REZULTAT
i  =1    w=10.3
i++=1 ++w=11.3
i  =2    w=11.3
Press any key to continue
*/
```

Pentru evitarea ambiguităților, este indicat să nu se folosească de mai multe ori în aceeași expresie acești operatori aplicați asupra aceleiași variabile.

```

#include<iostream>
using namespace std;
int main (){
    int tab[3]={10,11,12};
    int i=0;
    tab[++i]=++i;
    for(i=0;i<3;i++) cout<<tab[i]<<endl;
    return 0;
}
/* REZULTAT
10
11
2
Press any key to continue . . .*/

```

B2. *Operatorul adresă* (operatorul *ampersand*) se aplică asupra unei l-valori și are ca rezultat adresa locației desemnate. Mai precis rezultatul este o constantă de tip pointer către locația dată.

B3. *Operatorul țintă* (operatorul *steluță*) se aplică asupra unui pointer și are ca rezultat atât r-valoarea cât și l-valoarea țintei.

```

#include<iostream>
using namespace std;
int main()
{
    int i=108,j;
    cout<<"valoarea lui i este "<<i<<endl;
    cout<<"adresa lui i este "<<&i<<endl;

    int* p; //declaratie de pointer catre int
    p=&i; //p este incarcata cu adresa lui i

    //utilizam r-valoarea expresiei *p
    j=*p;
    if(j==i) cout<<"valoarea tinteii lui p este "<<i<<endl;

    //utilizam l-valoarea expresiei *p
    *p=100;
    if(i==100) cout<<"tinta lui p este i"<<endl;

    return 0;
}
/* Rezultat:
valoarea lui i este 108
adresa lui i este 0045FE10
valoarea tinteii lui p este 108
tinta lui p este i
Press any key to continue . . .*/

```

B4. *Operatorul de păstrare a semnului* (operatorul *plus unar*) nu modifică cu nimic valoarea operandului asupra căruia se aplică. În cazul tipurilor **char** și **short** au loc totuși promovările implicite la tipul **int**.

```

#include<iostream>
using namespace std;
int main()
{
    int i=100;
    cout<<i<<endl;           //100
    cout<<+ i<<endl;          //100
    cout<<+ +i<<endl;          //101
    //cout<<+++i<<endl;        //error C2105: '++' needs l-value
    //cout<<++(i)<<endl;        //error C2105: '++' needs l-value
    cout<<+(++i)<<endl;          //102
    cout<<++++i<<endl;          //104

    char ch='a';
    cout<<sizeof(ch)<<endl; //1
    cout<<sizeof(+ch)<<endl; //4

    return 0;
}

```

B5. *Operatorul de schimbare a semnului* (operatorul *minus unar*) se aplică asupra unui operand aritmetic (întreg sau flotant) și are ca rezultat operandul cu semn schimbat. Compilatorul avertizează schimbările de semn în cadrul tipului unsigned, dar nu le interzice. În acest caz rezultatul operației depinde de compilator.

```

#include<iostream>
using namespace std;
int main()
{
    int i=10,j,k,h,n;
    j=-i;
    k=-(-i);
    h=--i;
    n=--+-i;
    cout<<"i="<<i<<endl;
    cout<<"j="<<j<<endl;
    cout<<"k="<<k<<endl;
    cout<<"h="<<h<<endl;
    cout<<"n="<<n<<endl;
    return 0;
}

/*REZULTAT
i=9
j=-10
k=10
h=9
n=9
Press any key to continue . . .
*/

```

B6. *Operatorul de negare pe biți* (operatorul *tilda*) se aplică unui operand întreg și are ca rezultat întregul obținut prin comutarea din 0 în 1 și din 1 în 0 a biților operandului. Spunem că operandul este negat

bit cu bit. Tipul rezultatului este cel obținut după promovările implicite din cadrul tipurilor întregi. Cadrul natural de lucru pe biți este tipul **unsigned int**.

```
#include<iostream>
using namespace std;
int main(void) {
    unsigned n,m;
    n=0xA000FFF5;
    m=~n;
    cout<<hex<<"n="<<n<<"\nm="<<m<<dec<<endl;
    return 0;
}
/*REZULTAT:
n=a000fff5
m=5fff000a
Press any key to continue...*/
```

B7. *Operatorul de negare logică* (operatorul *not*) se aplică unui operand de tip aritmetic sau pointer și are ca rezultat 0 (fals) dacă operandul este nenul (adică adevărat) și 1 dacă operandul este nul. Tipul rezultatului este **bool**.

```
#include<iostream>
using namespace std;
int main(void) {
    int a=2,b=3,tab[2]={1000,2000};
    cout<<!a<<!!a<<!!!a<<endl;

    if(!(a==b)) cout<<"NU sunt egale"<<endl;
    else cout<<"DA, sunt egale"<<endl;

    cout<<tab[1-!(a<b)]<<endl;

    return 0;
}
/*REZULTAT:
010
NU sunt egale
2000
Press any key to continue . . .*/
```

Prezentăm în programul următor un exemplu tipic de utilizare a operatorului de negare logică. Se cere să se genereze în mod *randomizat* (adică aleator) numere naturale mai mici decât 100 până la prima apariție a lui 10:

```
#include<iostream>
#include <time.h>
using namespace std;
int main(void) {
    srand ( time(NULL) );//initiere generator de numere aleatoare
    int nrExtras;
    for(bool amGasit = false; !amGasit; amGasit=(nrExtras==10)){
        nrExtras =rand() % 100;//rand() returneaza un numar aleator
        cout<<nrExtras<<endl;
    }
    return 0;
}
```

B8. Operatorul **sizeof** returnează mărimea în octeți a spațiului necesar stocării în memorie a unei date de tipul operandului. Operandul poate fi o denumire de tip sau o expresie.

În primul caz numele tipului trebuie scris obligatoriu între paranteze rotunde.

În al doilea caz parantezele pot lipsi dacă nu sunt neapărat necesare. Tipul expresiei este determinat fără evaluarea expresiei.

```
#include<iostream>
using namespace std;
int main(void){
    int a=2,tab[100];
    cout<<sizeof(double)<<endl;    //8
    cout<<sizeof(tab)<<endl;        //400
    cout<<sizeof (a+1.1)<<endl;    //8
    cout<<sizeof ++a+1.1<<endl;    //5.1
    cout<<sizeof 1.1+a<<endl;      //10
    cout<<sizeof("Clar!")<<endl;  //6
    return 0;
}
```

Operatorul **sizeof** este indispensabil în alocarea memoriei de către programator.

În exemplul următor încercăm să aflăm mărimea unei date de tip „funcție care primește un **int** și returnează un **int**”. Este evident că funcțiile de acest tip nu au o lungime prescrisă a codului și, prin urmare, acestui tip de dată nu i se poate asocia o anumită mărime a spațiului de stocare:

```
#include<iostream>
using namespace std;
int dubleaza(int a){
    return 2*a;
}
int main(void){
    cout<<sizeof(dubleaza(100))<<endl;
    //4 (marimea tipului returnat de apelul functiei)
    cout<<sizeof(dubleaza)<<endl;
    //error C2070: illegal sizeof operand
    return 0;
}
```

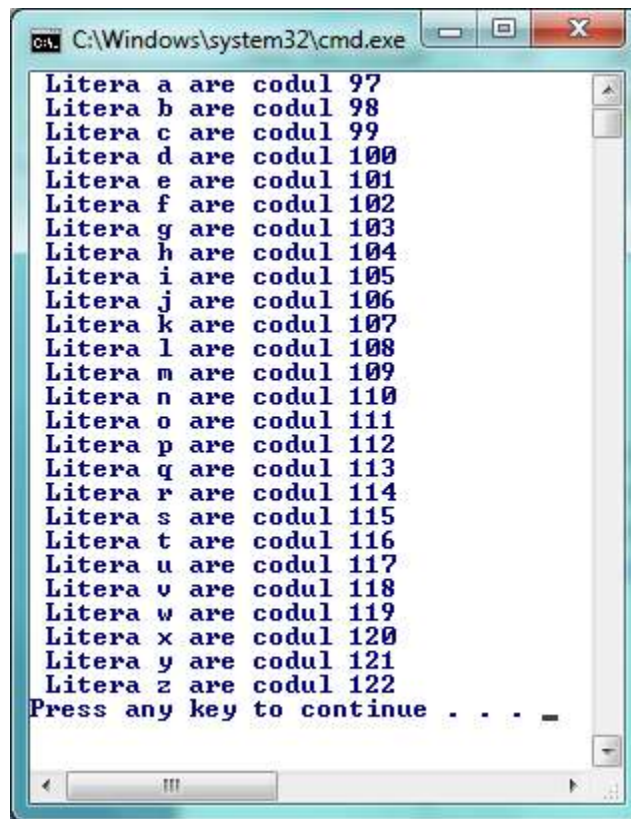
B9. Operatorul de conversie explicită, numit și operatorul de *casting* (cu sensul de distribuire de roluri), schimbă temporar tipul operandului, dar numai dacă tipul cerut este compatibil cu tipul operandului.

```
#include<iostream>
using namespace std;
int main(){
    int a=7,b=3;
    cout<<a/b<<endl;                //impartire cu rest
    cout<<(double)a/b<<endl;        //impartire cu virgula
    cout<<(double) (a/b)<<endl;      //impartire cu rest
    cout<<(int)cout<<endl;
    //error C2440: 'type cast' : cannot convert
    //from 'std::ostream' to 'int'
    return 0;
}/*
2
2.33333
2
Press any key to continue . . .*/
```

În exemplul următor a fost necesar castingul variabilei **ch** de la **char** la **int** deoarece operatorul de scriere în **cout** se comportă diferit când are de scris un **char** sau un **int**, chiar dacă cele două date au aceeași valoare numerică:

```
#include<iostream>
using namespace std;
int main(void){
    for(char ch='a'; ch<='z'; ch++)
        cout<<" Litera " << ch <<" are codul " <<(int)ch<<endl;
    return 0;
}
```

Rezultat:



```
C:\Windows\system32\cmd.exe
Litera a are codul 97
Litera b are codul 98
Litera c are codul 99
Litera d are codul 100
Litera e are codul 101
Litera f are codul 102
Litera g are codul 103
Litera h are codul 104
Litera i are codul 105
Litera j are codul 106
Litera k are codul 107
Litera l are codul 108
Litera m are codul 109
Litera n are codul 110
Litera o are codul 111
Litera p are codul 112
Litera q are codul 113
Litera r are codul 114
Litera s are codul 115
Litera t are codul 116
Litera u are codul 117
Litera v are codul 118
Litera w are codul 119
Litera x are codul 120
Litera y are codul 121
Litera z are codul 122
Press any key to continue . . .
```

Observație: compilatorul nostru suportă și „stilul C++ de conversie explicită”, de forma

```
ch=char(i+1);
```

dar acest stil poate fi utilizat numai pentru conversii către tipuri desemnate printr-un identificador. Din acest motiv nu recomandăm utilizarea lui acum.

## 5 C. Operatori binari.

Operatorii binari aritmetici au tipul rezultatului stabilit în funcție de tipul operanzilor, care pot fi de tip aritmetic sau de tip pointer. În cazul prezenței pointerilor se aplică regulile de calcul cu pointeri, iar în cazul ambilor operanzi de tip aritmetic se aplică următoarea convenție: dacă operanzii au același tip rezultatul are tipul comun, în caz contrar operandul cu tipul mai mic

este convertit la tipul celuilalt care va fi și tipul rezultatului. Ordinea tipurilor aritmetice este următoarea: **int**, **unsigned int**, **long int**, **unsigned long**, **long long int**, **unsigned long long int**, **float**, **double**, **long double**.

Orice operand de tip **char** sau **short** este promovat din start la **int** și apoi, dacă mai este cazul, la tipul rezultatului.

Operatorii binari logici au rezultatul de tip **bool** iar operanzii trebuie să fie de tip aritmetic sau logic. La evaluare, operanzii de tip aritmetic sunt mai întâi convertiți implicit la tipul **bool** și apoi se aplică operațiile logice corespunzătoare. Conversia implicită la **bool** se efectuează după convenția: zero trece în **false**, orice valoare nenulă trece în **true**.

C1. *Operatorii multiplicativi* sunt următorii: *operatorul de înmulțire* \*, *operatorul de împărțire* / și *operatorul modulo* %. Operanzii trebuie să fie aritmetici. În cazul împărțirii, dacă ambii operanzi sunt întregi rezultatul este câtul împărțirii cu rest, altfel are loc împărțirea cu virgulă. La împărțirea cu rest câtul se determină prin trunchiere:

```
cout<<8.0/-3.0<<endl;           //-2.66667
cout<<8/-3<<endl;               //-2
```

Toți acești trei operatori au același nivel de prioritate iar asocierea lor este de la stânga la dreapta:

```
cout<<2*100/2*100<<endl;         //10000
```

Operatorul modulo se aplică numai operanzilor întregi și are ca rezultat restul împărțirii determinat astfel încât să fie respectată „proba împărțirii”:

$a = (a / b) * b + a \% b$

Deoarece câtul  $a/b$  se determină prin trunchiere, în cazul existenței unui operand negativ restul  $a \% b$  rezultă negativ. Din acest motiv, pentru a decide de exemplu dacă un număr întreg  $n$  este congruent cu 2 modulo 3 (adică este de forma  $3k+2$ ), nu este suficient să testăm dacă  $n \% 3 == 2$ , deoarece și acele numere  $n$  pentru care  $n \% 3 == -1$  sunt congruente cu 2 modulo 3. În astfel de situații este indicat să testăm numai congruența cu zero:

```
#include<iostream>
using namespace std;
int main(void) {
    int tab[10]={12,14,23,-4,54,56,82,72,58,34};
    cout<<"In tabel sunt urmatoarele numere de forma 3k+2:"<<endl;
    for(int i=0;i<10;i++)
        //if(tab[i]%3==2 || tab[i]%3==-1) cout<<tab[i]<<" ";
        if( (tab[i]-2)%3==0) cout<<tab[i]<<" ";
    cout<<endl;
    return 0;
}

/*REZULTAT:
In tabel sunt urmatoarele numere de forma 3k+2:
14; 23; -4; 56;
Press any key to continue . . . */
```

C2. *Operatorii binari aditivi* sunt următorii: *operatorul de adunare* + și *operatorul de scădere* – . Acești operatori se aplică la operanzi de tip aritmetic sau de tip pointeri. Dacă ambii operanzi sunt aritmetici se aplică conversiile aritmetice uzuale, rezultatul fiind suma sau, respectiv, diferența operanzilor.

Dacă unul dintre operanzi este de tip pointer se aplică regulile de calcul cu pointeri, care vor fi studiate mai târziu. Un exemplu:

```
#include<iostream>
using namespace std;
int main(void){
    int i;
    int* p=&i;
    cout<<p<<endl;    //0023F814
    cout<<p+1<<endl;  //0023F818
    return 0;
}
```

C3. *Operatorii de deplasare pe biți* sunt utilizați pentru calcule logice rapide în binar. Operanzii trebuie să fie de tip întreg, cadrul natural de lucru fiind **unsigned int** (în cazul operanzilor negativi rezultatul depinde de compilator).

În baza 2, înmulțirea lui  $n$  cu  $2^m$  se execută prin mutarea cifrelor lui  $n$  în stânga cu  $m$  poziții și completarea cu zerouri în dreapta, operație efectuată de *operatorul de deplasare la stanga* << prin expresia  $n<<m$ . Exemplu:

```
unsigned n=40;
unsigned n_ori_8=n<<3;
cout<<n_ori_8;    //320
```

Analog, împărțirea lui  $n$  cu  $2^m$  se execută în baza 2 prin mutarea cifrelor lui  $n$  în dreapta cu  $m$  poziții și completarea cu zerouri în stânga, operație efectuată de *operatorul de deplasare la dreapta* >> prin expresia  $n>>m$ . Exemplu:

```
unsigned n=40;
unsigned n_supra_8=n>>3;
cout<<n_supra_8;    //5
```

Următorul program afișează „înainte și înapoi” primele 32 de puteri ale lui 2:

```
#include<iostream>
using namespace std;

int main(){
    unsigned a,b;
    for(int i=0; i<32; i++){
        a=1u<<i;
        cout<<a<<endl;
    }
    for(int i=0; i<32; i++){
        b=a>>i;
        cout<<b<<endl;
    }
    return 0;
}
```



Observație: în expresia `cout<<a` operatorul de deplasare la stânga are ca operanzi obiectul **cout** și întregul **a**, și nu doi întregi așa cum cere limbajul C. Acest lucru este posibil deoarece în C++ operatorii predefiniți pot fi supraîncărcați astfel încât să poată fi aplicați și obiectelor de tip clasă. Expresia dată este tradusă automat de orice compilator de C++ în apelul unei funcții membru cu numele „operator<<”, astfel:

```
cout.operator<<(a);
```

iar acest apel este acceptat deoarece în clasa **ostream** a obiectului **cout** a fost definit modul de acțiune al acestei funcții. În clasele de *stream*-uri operatorii << și >> au fost deci supraîncărcați, primind noi semnificații. Atenție, supraîncărcarea operatorilor extinde numai domeniul lor de aplicabilitate dar nu le poate schimba nici nivelul de prioritate și nici ordinea de asociere.

C4. Operatorii de comparație sunt următorii 4, cu sensuri evidente: *strict mai mic* <, *strict mai mare* >, *mai mic sau egal* <=, *mai mare sau egal* >=. Operanzii lor trebuie să fie ambii de tip aritmetic sau ambii de tip pointer. Rezultatul are tipul **bool**, cu valoarea **true** (1) dacă relația este respectată și **false** (0) în caz contrar.

Iată un exemplu corect de utilizare:

```
#include<iostream>
using namespace std;
int main() {
    int a,b;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    if(a<b) cout<<"a<b"<<endl;
    else cout<<"a>=b"<<endl;
    return 0;
}
```

unul ciudat:

```
#include<iostream>
using namespace std;
int main() {
    char mesaje[2][100]={"a<b", "a>=b"};
    int a,b;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    cout<<mesaje[a>=b]<<endl;
    return 0;
}
```

și unul greșit:

```
#include<iostream>
using namespace std;
int main() {
    int a=10, b=5, c=3;
    if( a > b > c )
        cout<<"DA"<<endl;
    else
        cout<<"NU"<<endl;
    return 0;
} //pe monitor: NU
```

C5. Operatorii de egalitate sunt operatorul *egal-egal* `==` și operatorul *not-egal* `!=`. La fel ca la operatorii de comparație ambii operanzi trebuie să fie aritmetici sau ambii de tip pointer, iar rezultatul este de tip **bool**.

Expresia `a!=b` este echivalentă cu `!(a==b)`, dar prima este de preferat deoarece are un singur operand iar ultima are doi. Expresia `a!=b` nu este echivalentă cu `!a==b` deoarece operatorul de negare leagă mai tare decât comparația. Atenție și la confuzia des întâlnită între comparație și atribuire:

```
#include<iostream>
using namespace std;
int main(){
    int a=1, b=2;

    cout<<"\nCORECT:"<<endl;

    cout<<"a="<<a<<" b="<<b<<endl;
    if(a==b)
        cout<<"DA a==b"<<endl;
    else
        cout<<"NU a==b"<<endl;
    cout<<"a="<<a<<" b="<<b<<endl;

    cout<<"\nGRESIT:"<<endl;

    cout<<"a="<<a<<" b="<<b<<endl;
    if(a=b)
        cout<<"DA a=b"<<endl;
    else
        cout<<"NU a=b"<<endl;
    cout<<"a="<<a<<" b="<<b<<endl;
    a=0;
    b=0;

    cout<<"\nGRESIT:"<<endl;

    cout<<"a="<<a<<" b="<<b<<endl;
    if(a=b)
        cout<<"DA a=b"<<endl;
    else
        cout<<"NU a=b"<<endl;
    cout<<"a="<<a<<" b="<<b<<endl;
    return 0;
}
/*REZULTAT:

CORECT:
a=1 b=2
NU a==b
a=1 b=2

GRESIT:
a=1 b=2
DA a=b
a=2 b=2
```

```

GRESIT:
a=0 b=0
NU a=b
a=0 b=0
Press any key to continue*/

```

C6. Operatorul „și” pe biți & se aplică la doi întregi după ce s-au efectuat conversiile aritmetice uzuale. Cei doi operanzi sunt parcurși în același timp și se aplică pe rând operatorul boolean „și” biților de același rang.

C7. Operatorul „sau exclusiv” pe biți ^ este analog cu operatorul „și” pe biți, operația iterată fiind „sau exclusiv”. Rezultatul lui „sau exclusiv” aplicat unei perchi de biți este 0 dacă biții sunt egali și 1 dacă sunt diferiți.

C8. Operatorul „sau” pe biți | este analog cu operatori pe biți de mai sus, operația iterată fiind acum „sau”-ul boolean.

Cadrul natural de lucru pentru operatorii logici pe biți este tipul **unsigned**. Exemplu de aplicare:

```

#include<iostream>
using namespace std;
int main(void) {
    unsigned p,q,x,y,z;
    p=0xff00aa00;      //1111 1111 0000 0000 1010 1010 0000 0000
    q=0xaabbccdd;      //1010 1010 1011 1011 1100 1100 1101 1101

    x=p&q;              //1010 1010 0000 0000 1000 1000 0000 0000
    y=p^q;              //0101 0101 1011 1011 0110 0110 1101 1101
    z=p|q;              //1111 1111 1011 1011 1110 1110 1101 1101

    cout<<hex;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
    cout<<"z="<<z<<endl;
    cout<<dec;
    return 0;
}

/* REZULTAT:

x=aa008800
y=55bb66dd
z=ffbbeedd
Press any key to continue . . .*/

```

Operatorii logici pe biți pot fi folosiți, de exemplu, pentru implementarea calculului cu mulțimi. Dacă  $U = \{x_0, x_1, x_2, \dots, x_{31}\}$  este o mulțime finită cu 32 de elemente, atunci orice submulțime  $A$  a sa este unic determinată de un **unsigned a** având biții poziționați după regula: pentru fiecare  $k$  de la 0 la 31, bitul de ordin  $k$  este 1 dacă  $x_k$  aparține lui  $A$ , altfel este 0.

Dacă  $a$  și  $b$  determină submulțimile  $A$  și  $B$ , atunci  $a \& b$ ,  $a^b$ ,  $a|b$  și  $\sim a$  determină intersecția  $A \cap B$ , diferența simetrică  $A \Delta B$ , reuniunea  $A \cup B$  și, respectiv, complementara lui  $A$  față de mulțimea universală  $U$ .

Poziționarea bitului de ordin  $k$  poate fi efectuată cu ajutorul funcțiilor următoare:

```

unsigned set_bit(unsigned a, int k){           //a_k=1
    if(k<0||k>31) return a;
    return a|(1u<<k);
}

unsigned clear_bit(unsigned a, int k)         //a_k=0
    if(k<0||k>31) return a;
    return a&~(1u<<k);
}

unsigned toggle_bit(unsigned a, int k){      //a_k=~a_k
    if(k<0||k>31) return a;
    return a^(1u<<k);
}

```

Aici utilizăm „masca de biți”  $1u \ll k$  în care numai bitul de ordin  $k$  este 1, restul fiind 0. Masca protejează biții celui alt operand de acțiunea operatorului. Deoarece pentru orice bit  $x$  expresia „ $x$  sau 0” are valoarea  $x$ , se spune că 0 ascunde (maschează) acțiunea lui „sau”. Biții 0 ai unei măști ascund acțiunea lui „sau” și a lui „sau exclusiv” iar biții 1 ascund acțiunea lui „și”.

**C9. Operatorul „și” logic &&** se aplică la doi operanzi logici și are ca rezultat un **bool** cu valoarea **true** dacă ambii operanzi sunt adevărați, altfel rezultatul este **false**. În cazul operanzilor numerici (aritmetici sau pointeri) are loc conversia implicită la tipul **bool** înainte de evaluare.

Exemplu:

```
if(a==b && b==c) cout<<"a,b si c sunt egale"<<endl;
```

În cazul operatorului && este precizată ordinea de evaluare de la stânga la dreapta a operanzilor, mai mult, dacă evaluarea primului operand decide rezultatul final (adică dacă primul operand e fals), atunci al doilea operand nu mai este evaluat, și în consecință eventualele efecte secundare ale acestuia nu se mai produc. Se spune că operatorul && este „leneș” sau că este evaluat în scurt-circuit.

Iată un exemplu de utilizare a acestei proprietăți:

```

#include<iostream>
using namespace std;
int main (void){
    int tab[10]={1,2,0,3,4,0,5,6,0,7};
    int i,contor;
    contor=0;
    cout<<"In sirul"<<endl;
    for(i=0;i<10;i++){
        cout<<tab[i]<<" ";
        tab[i] && contor++;
    }
    cout<<"\nsunt "<<contor<<" elemente nenule"<<endl;
    return 0;
}
/*REZULTAT:
In sirul
1 2 0 3 4 0 5 6 0 7
sunt 7 elemente nenule
Press any key to continue . . .*/

```

Utilizarea evaluării în scurt-circuit a și-ului logic este esențială în multe situații. În programul următor, de exemplu, parcurgerea tabloului **tab** se termină cu o eroare la rulare deoarece la evaluarea expresiei

```
0!=tab[i]++ && i<dimMax
```

noi modificăm elementul de indice **i** înainte de a testa dacă avem voie. Expresia corectă este

```
i<dimMax && 0!=tab[i]++
```

deoarece acum incrementarea lui **tab[i]** nu mai are loc dacă am ieșit din tablou.

```
include<iostream>
using namespace std;

const int dimMax=8;

void mareste(int tab[dimMax]) {
    //incrementeaza elementele lui tab pana la primul zero
    //for(int i=0; i<dimMax && 0!=tab[i]++; i++)//corect
    for(int i=0; 0!=tab[i]++ && i<dimMax; i++)
        cout<<tab[i]<<endl;
    return;
}

int main() {
    int a[dimMax]={10,20,30, 40,50,60,70,80};
    mareste(a);
    return 0;
}
```



C10. Operatorul „sau” logic || acționează într-un mod similar „și”-ului logic, având prescrisă și el ordinea de evaluare a operanzilor, tot de la stânga la dreapta în scurt-circuit.

Exemplu de utilizare:

```
#include<iostream>
using namespace std;

//Urmatoarea functie decide daca
//i si j apartin multimii {0,1}

bool suntUndeTrebuie(int i, int j){
    return (i == 0 || i == 1) && (j == 0 || j == 1);
}
```

```

int main(){
    int i = 1, j = 1;
    if (suntUndeTrebuie(i, j))
        cout << "DA" << endl;           //    DA
    else
        cout << "NU" << endl;
    return 0;
}

```

Programul următor numără din câte încercări reușește utilizatorul să introducă de la tastatură un număr dintr-un interval specificat:

```

#include<iostream>
using namespace std;
int main()
{
    double x, xmin=0, xmax=100;
    cout<<"Dati un numar strict intre "<<xmin<<" si "<<xmax<<endl;
    int i, imax=5;
    bool aReusit = false;
    for (i = 0; i <= imax && !aReusit; i++){
        cout << "x="; cin >> x;
        aReusit = (xmin < x && x < xmax);
    }
    if(i==1) cout<<"Ok, ati reusit de la prima incercare"<<endl;
    else if(i<=imax) cout<<"Ati reusit din "<<i<<" incercari"<<endl;
    else cout<<"Data viitoare poate veti fi mai atent!"<<endl;
    return 0;
}

```

/\*REZULTAT:

```

Dati un numar strict intre 0 si 100
x=100000
x=10000
x=1000
x=100
x=10
Ati reusit din 5 incercari
Press any key to continue . . .*/

```