LUCRAREA 3

Scopul lucrării *este studiul conceptului tipic de corp de instrucțiuni, lucrul cu operatorii, construcția expresiilor care stau la baza oricărui program, combinarea acestora în instrucțiuni și parcurgerea unui exemplu complet de program.*

I. OBSERVAŢII TEORETICE

1.1. Corp (sau bloc) de instrucţiuni

Prin corp de instrucţiuni se înţelege o mulţime de instrucţiuni, grupate între acolade: deschisă la început, şi închisă la sfârşit, în această ordine.

Această construcţie poate apărea:

a. singular;
b. asociată unei instrucţiuni de decizie;
c. asociată unei instrucţiuni de buclare (ciclare);
d. legată de numele unei funcţii - adică urmând acestuia.

Pentru fiecare dintre situaţii, denumirea corpului de instrucţiuni se poate schimba:

a) corp de instrucţiuni;
b) corp de decizie;
c) corp de ciclu;
d) corp de funcţie.

După cum am văzut la punctul ***1.4.*** al laboratorului trecut, declaraţia/definiţia unei variabile se face `la începutul` unui bloc de instrucţiuni.

Observaţie:

Există o particularitate a limbajului `ANSI C`, pe care compilatorul `Dev_C++` nu o respectă. Aceasta se întâmplă pentru că în C++, o variabilă se poate declara/defini la momentul folosirii sale, dând o notă de flexibilitate acestui limbaj, faţă de C.

Un corp de instrucţiune poate fi `imbricat` - cuprins - într-un altul. Acesta este un aspect interesant, dacă îl privim d.p.d.v. al variabilelor. Şi anume: o variabilă `este valabilă` pe durata fiecărui bloc de instrucţiuni în care este declarată/definită. Odată cu încheierea acelui corp de instrucţiuni, dispare şi posibilitatea de a mai folosi variabilele ce au fost declarate/definite în cadrul său. Rămân valabile doar variabilele declarate/definite în cadrul corpului de instrucţiuni cel mai cuprinzător. Toate aceste aspecte sunt legate de `durata de viaţă` a variabilelor.

Mai există o situaţie, şi anume `vizibilitatea` variabilelor. Prin vizibilitate se înţelege porţiunea de program în care o variabilă poate fi accesată şi folosită. Regula este relativ simplă (şi intuitivă): dacă o variabilă, declarată în program la un moment `t+1` are un nume identic cu al altei variabile, ce exista în program de la momentul `t`, o 'maschează' pe cea de la momentul `t`. Adică, din momentul declarării celei de-a doua, prima nu mai este vizibilă din acest moment înainte. Dacă se doreşte ca ambele variabile să poată fi folosite, există, ascunsă, o soluţie pe baza corpului de instrucţiuni: dacă fiecare din cele două variabile cu nume identic face parte dintr-un corp de instrucţiuni diferit, atunci, funcţie de momentul în care se încheie unul sau altul dintre corpuri, una dintre variabile are o durată de viaţă mai mare, devenind vizibilă întregului corp de instrucţiuni în care a fost declarată.

Există în cadrul funcţiei `functie(void)` din programul `lab3_2.cpp` un comentariu. Dacă renunţaţi la el, recompilând avem următoarele erori/avertismente:

C:\BC\MyApp\Laborator\Lab_3\lab3_2.cpp - [Warning] In function `void functie()':

37 C:\BC\MyApp\Laborator\Lab_3\lab3_2.cpp -'ch1' undeclared (first use this

37 C:\BC\MyApp\Laborator\Lab_3\lab3_2.cpp - 'ch2' undeclared (first use this

1.2. Operatori

Odată declarate/definite variabilele, un programator merge mai departe, şi în programul său începe să folosească acele variabile, în diferite contexte particulare de lucru. În aceste contexte apar şi simbolurile destinate operatorilor. Prin operator se înţelege elementul (entitatea sau simbolul) prin care se poate obţine un efect oarecare în urma folosirii sale.

Un operator, pentru a avea sens, deci pentru a genera un efect are nevoie de entităţile numite operanzi, adică numele variabilelor pe care le cer, prin definiţie, pentru a exista, şi asupra cărora chiar acţionează, în anumite situaţii.

De exemplu, `operatorul de adunare (+)`, pentru a avea sens cere doi operanzi, să spunem a şi b, a.î. să putem scrie corect:

```
a + b;
```

Fără a sau fără b, operaţia `adunare` nu are sens, iar compilatorul marchează acest lucru printr-un mesaj de eroare de genul '`Missing operand or identifier`'.

8 C:\BC\MyApp\Laborator\Lab_3\lab3_3.cpp - parse error before ';' token

1.2.1. Clasificarea operatorilor

Operatorii se pot `clasifica` din următoarele puncte de vedere:

- **`clasă de operatori`**: aritmetici, logici, logici pe bit, comparaţie, atribuire, conversie explicită (cast), incrementare/decrementare (pre- şi post-), referenţiere/dereferenţiere, indexare, operatorul virgulă.
- **`mod de asociere a operanzilor`**:
  - stânga-dreapta (uzual)
  - dreapta-stânga (special).
- **`numărul de operanzi`** impus:
  - *unar*: operatorii logici pe bit: `NOT (!), OR (|), AND (&), XOR (^), Complement de 1 (~)`;
  - *binar*: operatorii aritmetici, logici, sau de comparaţie;
  - *ternar*: operatorul condiţional: `? :` (singurul cu această particularitate).
- **`prioritate`**:
  - *implicită* - dată de tabelul de prioritate;
  - *explicită*, modificând-o pe cea implicită, cu ajutorul parantezelor rotunde (ca operatori de prioritate maximă).

Precedență	**Asociativitate**
() a() a[b] a->b a[b..c] ({ }) ([ ]) (< >)	left to right
a++ a--	left to right
!a ~a (type)a ++a --a	right to left (!)
a*b a/b a%b	left to right
a+b a-b	left to right
a>>b a<<b	left to right
a>b a>=b a<b a<=b	left to right
a==b a!=b	left to right
a&b	left to right

a^b	left to right
a|b	left to right
&&	left to right
||	left to right
a?b:c	right to left (!)
= += -= *= /= %= <<= >>= &= |= ^=	right to left (!)
,	left to right

Tabelul priorității operatorilor (comportare implicită, predefinită* = `default`*)

1.2.2. Operatorul condiţional (operator ternar)

S-a folosit aici şi operatorul condiţional, (`?:`), care reprezintă singurul operator ternar al limbajului C. `Ternar` semnifică faptul că, pentru a avea sens, necesită trei operanzi. Aceştia constituie trei expresii, având semnificaţia următoare:

- prima expresie apare în faţa semnului de întrebare, şi are rol de condiţie
- a doua apare între semnul de întrebare şi cele două puncte, semnificând ceea ce trebuie evaluat în caz că prima expresie are valoarea logică Adevărat (`1`)
- a treia expresie apare după cele două puncte, şi are semnificaţia de expresie ce se evaluează în cazul în care prima expresie a avut valoarea logică Fals (`0`).

1.2.3. Operatorii de incrementare/decrementare (operatori unari)

O atenţie specială trebuie acordată operatorilor de post şi pre-incrementare/decrementare. Aceşti operatori sunt unari, adică necesită un singur operand pentru a avea sens (se aplică asupra unui singur operand).

Prin incrementare se înţelege `creşterea cu o unitate` a valorii asupra cărei se aplică operatorul. Iar prin decrementare se înţelege `descreşterea cu o unitate`.

Incrementarea se notează cu ++, iar decrementarea cu - -.

Există următoarele două situaţii: operatorul de incrementare/decrementare `precede` sau `succede` operandul asupra căruia se aplică. Funcţie de apariţia sa efectele sunt diferite.

Reţineţi că dacă operatorii de **`post-`**incrementare/decrementare apar într-o `expresie`, evaluarea lor se face **`ultima`**. Dacă însă nu apar în expresii, ci în instrucţiuni de genul:

```
a++;
```

sau

```
a--;
```

efectul lor este de a incrementa/decrementa valoarea variabilei asupra căreia se aplică, deci nu se pune problema de `ordine` de evaluare.

La operatorii de **`pre-`**incrementare/decrementare, dacă ei apar într-o expresie, evaluarea se comportă conform regulilor din tabelul de prioritate, bineînţeles dacă nu apar parantezele rotunde care să schimbe aceste reguli. Dacă apar singuri, ca în:

```
--a;
```

sau

```
++a;
```

efectul este de incrementare/decrementare, similar cu al operatorilor de post-incrementare.

Orice instrucţiune de genul celor prezentate reprezintă o expresie. De obicei, un program C conţine expresii în aceeaşi măsură în care conţine şi alte tipuri de instrucţiuni (de atribuire, de afişare, de alocare/dealocare de memorie ş.a.).

1.2.4. Operatori logici - tabel de adevăr

Există situaţia în care se doreşte o combinaţie între două valori cu caracter special: adevărat şi fals. Aceste valori sunt speciale pentru că ele au o semnificaţie deosebită î operaţiile logice. Pentru astfel de operaţii C aduce o gamă de operatori ce pot fi folosiţi atât pe bit (adică la nivel intern, în reprezentare binară), cât şi în expresii ce ţin loc de condiţii pentru **instrucțiunea de decizie** if-else, sau pentru instrucţiunile de ciclare (for(), while(), do-while()).

Tabelele de adevăr sunt următoarele:

Operand 1	Operand 2	OR (SAU)	AND (ŞI)	XOR (SAU exclusiv)	NOT (Operand1)
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Tabelul valorilor de adevăr ale operațiilor logice fundamentale

1.3. Expresia = {operanzi, operatori}

Compilatorul are predefinit un comportament (mod de lucru *default* - implicit), în ce priveşte operatorii, în situaţia în care întâlneşte o expresie. La baza acestui comportament stă tabela de prioritate a operatorilor.

Prin expresie se înţelege mulţimea de operanzi şi operatori ce constituie la un moment dat. o acţiune dorită din partea programatorului.

De exemplu:

```
d = (a<<3) + b/((c==3)?c+1 : c-1); // c devine 4, dacă inițial a fost 3,
                                   // sau devine 2, dacă inițial a fost diferit de 3.
```

reprezintă o expresie. Aici apar în acelaşi context şi operatori şi operanzi.

Problema importantă este d.p.d.v. al rezultatului. Observaţi că expresia conţine şi paranteze rotunde. Din tabelul de prioritate se vede că aceştia sunt operatorii de clasă de prioritate 1, adică, în orice expresie, sunt primii evaluaţi, şi în funcţie de poziţia lor se face evaluarea ulterioară a eventualelor expresii conţinute de acestea, şi în cele din urmă, a întregii expresii.

1.4. Instrucţiunea. Definiţie

O instrucţiune (denumită în limbajul compilatorului şi **statement**, ceea ce în română înseamnă afirmație) este acea combinaţie de simboluri şi cuvinte-cheie care se încheie cu punct şi virgulă (;)

Un program C este format NUMAI din instrucţiuni. Fie că ele sunt de afișare (apelul funcţiei printf()), de atribuire (de forma stânga=dreapta;), de decizie (if-else) sau de tip expresie.

Instrucţiunea de 'chemare în execuţie' a unei funcţii este denumită pe scurt apel de funcție.

Orice program de până acum poate constitui un exemplu pentru instrucţiuni. Rulaţi şi programul final al platformei, pentru un exemplu de program complet,de calcul al derivatei numerice a unei funcţii reale de variabilă reală.

1.5. Exemplu complet de program: calculul derivatei unei funcţii prin două puncte

În final, să urmărim efectul programului `lab3_6.cpp`. Aici se calculează derivata unei funcţii, după o definiţie uşor modificată faţă de cea clasică a lui `Lagrange`. Algoritmul în sine poartă denumirea de **`derivată prin două puncte`**.

Cele două puncte de calcul sunt vecinătăţi ale punctului central x_0, în care trebuie calculată derivata. Ca şi în definiţia clasică a derivatei există şi aici un raport, între o diferenţă de ordonate şi una de abscise.

Relaţia de calul, ce reprezintă o instrucţiune de tip expresie este:

$f'(x_0) = (f(x_0+h/2) - f(x_0-h/2)) / h;$

Cantitatea `h` din formulă este subunitară şi reprezintă chiar vecinătatea despre care am amintit mai sus. O valoare tipică este de 10^{-3}. Cele două puncte sunt: $x_0-h/2$, respectiv $x_0+h/2$. Se observă că ele sunt `vecinătăţi` ale punctului x_0. Aşadar, este şi aici valabilă noţiunea de `vecinătate`, ca în definiţia consacrată a derivatei.

Pentru un studiu asupra programului alegeţi mai multe funcţii, pentru a verifica generalitatea acestei formule. Ca orice algoritm, şi acesta, pentru a funcţiona normal (a oferi rezultate corecte) admite la intrare un domeniu limitat de valori, care poartă denumirea de `univers al problemei` de rezolvat. De aceea, în alegerea funcţiilor, trebuie avut în vedere domeniul de definiţie al acestora.

Observaţie:

Un programator care se respectă va face în cadrul programului toate testele - de bun simţ - necesare. Dar există cazuri în care nu poate avea în vedere toate situaţiile de eroare ce pot apărea. De exemplu, o valoare de intrare de tip caracter, într-o situaţie în care se doreşte citirea unui număr real. De aceea, un studiu prealabil din partea noastră asupra problemei în cauză ne poate scuti de eventualele erori neprevăzute (care nu au fost luate în calcul la scrierea acelui program).

1.6. Detalii de programare

Pentru înţelegerea programelor utilizate în lucrare studiaţi `ANEXA` lucrării.

1.7. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinaţia de taste `CTRL+S`), sau, echivalent, din meniul `File`, cu comanda `Save as...`.

Compilarea se va face cu comanda `CTRL+F9`, iar rularea cu comanda `CTRL+F10`. Compilarea şi rularea se pot executa, succesiv, printr-o singură tastă, anume `F9`. Acţiunile echivalente din meniu: meniul `Execute` comanda `Compile`, respectiv `Compile + Run`.

Observaţii:

1. Orice nouă modificare a codului-sursă trebuie urmată obligatoriu de salvarea acestuia (`CTRL+S`). Abia după aceea pot urma celelalte acţiuni dorite.
2. Dacă nu se realizează aducerea la zi a programului, compilarea şi rularea ce urmează se fac tot pe varianta veche, adică ce de dinaintea ultimelor modificări.

II. DESFĂŞURAREA LUCRĂRII

2.1. Se va analiza corpul de instrucţiuni în programul *lab3_1.cpp* ce realizează o funcţie $y=a^b$. Pe rând, în acest program sunt întâlnite cele patru situaţii de corp de instrucţiune. În acest program se calculează (cu ajutorul lui `functie(a, b)`) orice putere întreagă a bazei, care se poate reprezenta pe un întreg scurt, fără semn (`unsigned int`).

Pentru studiu, încercaţi să vedeţi care sunt puterile întregi ale lui 2. Atât `baza` cât şi `exponentul` sunt întregi fără semn. Rezultatul funcţiei este de asemenea întreg fără semn.

Ce se întâmplă dacă la un moment dat spaţiul de reprezentare necesar înmulţirilor este depăşit - o putere a lui 2 generează o valoare ce depăşeşte maxima unui întreg scurt fără semn, adică `65536` ($= 2^{16}$) ?

2.2. Se va studia folosirea aceluiaşi nume de variabilă în corpuri de instrucţiuni imbricate. Pentru a observa acest comportament interesant al compilatorului, compilaţi şi rulaţi programul `lab3_2.cpp`.

Modificaţi numele şi locul în care apar variabilele, şi notaţi-vă efectele asupra programului. Observaţi şi stilul funcţional de scriere al programului (stil funcţional = bazat pe `funcţii`). Apare aici şi o variabilă întreagă globală, pentru a marca zona în care se poate folosi aceasta: întreg fişierul în care aceasta apare.

Renunţând la comentariul de pe linia 37 a codului sursă, notaţi erorile şi avertismentele care apar. Comparaţi-le cu cele din paragraful 1.1.

2.3. Se studiază lipsa unui operand dintr-o expresie. Pentru aceasta se rulează programul `lab3_3.cpp`. Un mesaj de eroare apare dacă renunţaţi la comentariul de pe linia 8 a programului. Corectaţi eroarea, recompilaţi şi vedeţi rezultatul.

2.4. Se studiază operatorii de incrementare şi decrementare cu variantele `pre` şi `post`. Se rulează programul `lab3_4.cpp`. Aici, pe liniile 8 - 21 afişările sunt edificatoare.

2.5. Realizarea expresiilor şi prioritatea operatorilor se pot studia în programul `lab3_5.cpp`. Pentru expresiile

```
d = (a<<3) + b/((c==3)?c+1 : c-1);
```

şi

```
d = (a<<3) + b/((c==3)?c+1 : c-1);
```

şi valorile:

```
int a = 8;
float b = 12.2, c=3;
```

faceţi calculele individual pe hârtie, folosind tabelul de prioritate. Comparaţi rezultatele cu cele obţinute în program. Modificaţi ordinea de evaluare a expresiei folosind - în mod corect, pentru a nu genera erori - parantezele rotunde.

Se observă comportamentul compilatorului în situaţia a două `expresii`: una în care **nu** apar parantezele rotunde (operatorii cu prioritatea ce mai înaltă) şi o alta în care, datorită folosirii (voite) a parantezelor rotunde rezultatul este diferit, în funcţie ce a dorit programatorul. În primul caz compilatorul foloseşte regulile implicite de asociere a operatorilor, împreună cu ordinea lor de prioritate (din tabelul operatorilor) şi generează un rezultat în urma `evaluării expresiei`. În cel de-al doilea caz, apariţia parantezelor rotunde impune un alt mod de comportare decât cel implicit (schimbă comportamentul predefinit), evident cu un alt rezultat.

2.6. Ca o aplicaţie a noţiunilor prezentate se va calcula derivata prin două puncte a unei funcţii, prin rularea programului `lab3_6.cpp`. Identificaţi în program toate noţiunile prezentate în acest laborator. Notaţi-vă caracteristicile care vi se par esenţiale în fiecare caz.

III. ÎNTREBĂRI

3.1. Dacă variabilele au valorile specificate, folosind tabelul de prioritate (regulile standard de prioritate) determinaţi rezultatul următoarelor expresii:

```
float a = 2, b = 3,  c = 5;
a) a*b+c - 1/c
b) a+b*c - 9/c
c) (a+b) % c
d) 4*b*7/( (3*a) - c)
```

3.2. Determinaţi valorile finale ale variabilelor a, b, şi c. Presupunem că în fiecare caz, a porneşte cu valoarea 3, b cu valoarea 5, iar c cu valoarea 1.

```
a) b -= a++
b) c = ++a * b- -
```

3.3. Listaţi regulile operatorilor C: NOT, AND, OR, XOR , pentru *a = TRUE* şi *b=FALSE*.

3.4. Daţi exemplu de patru operatori logici pe bit şi explicaţi-le semnificaţia.

```
int a=4, b=15;
a) a&b
b) a^b
c) b <<(a|b)
d) !a & !(b>>2)
```

IV. ANEXĂ - sursele complete ale programelor

```
// lab3_1.cpp
// Studiu asupra variantelor de corpuri de instrucțiuni.
#include<stdio.h>
#include<conio.h>

unsigned int functie(unsigned int, unsigned int);  // calcul de putere intreaga:"a la b", ne-recursiv

int main(void)
{       // corp de functie = definitia functiei main()
  {     // corp de instructiuni
      unsigned int baza = 2, exponent = 31; // variabilele sunt aici initializate.
    // Ele pot fi citite si de la utiliztor, prin perechi 'printf()-scanf()'.
      printf("\n In corp: Valorile variabilelor intregi: a= %i, b= %i", baza, exponent);
      printf("\n Rezultatul lui %i la puterea %i: %u", baza, exponent, functie(baza, exponent));
  }     // end_corp

// comentariul următor se va ignora în etapa de studiu a programului
  /*
  printf("\n In afara corpului: Valorile variabilelor intregi: a= %i, b= %i", baza, exponent);
  */

  unsigned int baza = 2;   // redeclarare a variabilei 'baza'.
                           // Cea anterioara nu mai exista in acest punct.
  if(baza == 2)
  {     // corp de decizie.
   unsigned int rez;    // variabila declarata a inceputul unui corp de instructiuni. Asadar, corect.
   printf("\n Baza de calcul aici este : %i", baza);
   rez = functie(baza, 20);
   printf("\n Rezultatul calculului: %u, %x", rez, rez);
  }     // end_if

  getch();
  return 0;
}  // end_main

unsigned int functie(           unsigned int baza,
                                unsigned int exponent)
{       // corp de functie = definitia functiei
 unsigned int rez = 1;
 int i;         // am nevoie de un contor pentru un ciclu 'for()' (sau similar)
```

```
 if(exponent == 0) return 1;    // daca exponentul este nul, rezultatul este 1.
 for(i=exponent; i>=1; )
 {      // corp de ciclu
        rez = rez*baza;
        i = i-1;        // echivalenta cu 'i--;'
 }      // end_for
 // ganditi-va la un ciclu ce foloseste instructiunea i**, si NU i- -.

 return rez;
}       // end_functie


// lab3_2.cpp
// corpuri de instrucțiuni cuprinse unul în altul (imbricare).
#include<stdio.h>
#include<conio.h>

int global = 256;       // variabila globala, ce este vizibila in tot programul, deci in
                        // cadrul oricarui corp de instructiuni prezent.
void functie(void);     // nu se preia nici un argument (void), si nu se
                        // intoarce nici o valoare (void).

int main(void)
{
 {      // primul corp
        char ch1 = '9', ch2 = 'Y';      // locale corpului 1.
        printf("\n Caracterele din corp_1: ch1: %c si ch2: %c", ch1, ch2);

        { // al doilea corp
         char ch1 = '8';
        // local corpului 2. Mascheaza pe 'ch1' din corpul 1.
            printf("\n Caracterele din corp_2: ch1: %c si ch2: %c", ch1, ch2);
            printf("\n Variabila intreaga, din corp_2: global: %i", global);
        }// end_corp2

  // aici caracterul 'ch1' este cel declarat in corpul 1.
  printf("\n Caracterul din corp_1: ch1: %c", ch1);
  printf("\n Variabila intreaga, din corp_1: global: %i", global);
  }// end_corp1

  // apelul functiei
  functie();

  getch();
  return 0;
}// end_main

void functie(void)
{
 printf("\n In functie este vizibila DOAR variabila intreaga 'global' (!!)");
 printf("\n cu valoarea: %i", global);

 printf("\n Din functie: ch1: %c, ch2: %c, global: i", ch1, ch2, global);
}


// lab3_3.cpp
// Lipsa din expresie a unui operand.
#include<stdio.h>
#include<conio.h>

int main(void)
{
 int a=10, b=20, rez;

// renunțați la comentariul ce urmează și observați erorile de compilare
/* rez = a+;    // incorect: lipseste b, pentru ca adunarea sa aiba sens.*/

 rez = a + b;   // corect: ambii operanzi sunt prezenti,
 printf("\n Valoarea rezultatului: %i", rez);

 getch();
 return 0;
}
```

```
// lab3_4.cpp
// Operatori diverşi.
#include<stdio.h>
#include<conio.h>               // pentru funcția getch()

int main(void)
{
 int a = 16, b = 24, c; // valori hexa 10 respectiv 18.

 a = a++;       // valoarea lui a este 17. a++ este identic cu a=a+1
 printf("\n a: %d", a);

 c = a++;       // atentie la ordinea de evaluare (!). Mai intai c=a, apoi a=a+1
// Operatorul de post-incrementare apare intr-o expresie, deci se evalueaza ultimul.
 printf("\n a: %d, c: %i", a, c);       // la afisare c=17. a=18

 // incrementarea într-o instrucțiune
 c = ++a;       // valoarea lui c este 19. Valoarea lui a este tot 19.
// Operatorul de incrementare apare intr-o instructiune expresie
// (instructiunea de atribuire). Este primul evaluat, deci:
// mai intai a=a+1, apoi c = a
 printf("\n a: %d, c: %u", a, c);       // se afiseaza a: 19

// operatorul de incrementare, fără context expresie
 ++a;           // a = a+1. Nu exista aici ordine de evaluare (prioritate)
 printf("\n a: %d", a); // se afiseaza a: 19

// operatorul de decrementare, fără context expresie
 a- -;          // a=a-1. Nu exista aici ordine de evaluare (prioritate)
 printf("\n a: %d", a); // se afiseaza a: 19

 // operație logică pe biți (ŞI)
 c = 0; // stabilim in variabila 'c' valoarea neutra zero.
        // Se poate scrie si: c = c^c (operatia XOR=sau exclusiv)
 c = a&b;       // operatia SI pe bit
 printf("\n c: %u", c);

// operație logică pe biți (SAU)
 c = 0;         // reset
 c = a|b;       // SAU pe bit
 printf("\n c: %u", c);

// operație logică pe biți (XOR)
 c ^= c;        // reset in varianta XOR
 c = a^a;       // XOR
 printf("\n c: %u", c);

 // deplasare stânga
 c = b<<3;      // Deplasare stanga cu trei pozitii. Valoarea lui 'b' se inmmulteste
                // cu 2 la puterea 3. iar rezultatul se scrie in 'c'.
                // 'b' NU se modifica (!!!)
 printf("\n c: %u, b: %u", c, b);

// deplasare dreapta
 c = b>>3;      // Deplasare dreapta cu trei biti. Valoarea din b se imparte cu
                // 2 la puterea 3. 'b' NU se modifica (!!!)
 printf("\n c: %u, b: %u", c, b);

 getch();
 return 0;
}



// lab3_5.cpp
// Expresii în evaluare implicită şi/sau explicită.
#include<stdio.h>
#include<conio.h>

int main(void)
{
 int a = 8;
 float b = 12.2, c=3, d;        // variabila 'd' nu trebuie initializata neaparat,
// pentru ca in ea se va scrie, oricum, un rezultat.

// instrucțiune de calcul evaluată în prioritate implicită
// (lipsesc, cu excepția unei situații, parantezele rotunde)
```

```
 d = (a<<3) + b/c==3?c+1 : c-1; // o deplasare se poate aplica doar unei variabile intregi.
                                // Aici, lui a.
 printf("\n Expresia fara paranteze rotunde (evaluare implicita): %f", d);

// instrucțiune de calcul, cu prioritățile schimbate (apar paranteze rotunde)
 d = (a<<3) + b/((c==3)?c+1 : c-1);
 printf("\n Expresia cu paranteze rotunde (evaluare explicita): %f", d);

 getch();
 return 0;
}

// lab3_6.cpp
// Derivata unei funcții prin două puncte.
#include<stdio.h>
#include<conio.h>
#include<math.h>        // pentru functia 'exponentiala de x'.

float f(float); // functia de derivat. Preia ca argument punctul de derivare.
float der2p(float, float); // argumentele: punctul de calcul si vecinatatea 'h'.

int main(void)
{
 float punct, h;

 printf("\n Punctul de calcul: "); scanf("%f", &punct);
 printf("\n Vecinatatea punctului de calcul: "); scanf("%f", &h);

 printf("\n Valoarea derivatei in puncutl %f este %f", punct, der2p(punct, h));
 getch();
 return 0;
}

float f(float x)
{
 return log(x); // logaritmul natural ln(x). Pentru logaritm zecimal
// exista functia de biblioteca 'log10()'
// return exp(x);       // este permisă o singură instructiune return; folosind comentariile
                        // se poate schimba fie funcția, fie valoarea de retur.
 }

float der2p(    float x0,
                float h)
{
 return (f(x0+h/2) - f(x0-h/2) ) / h;   // aici nu sunt necesare variabile locale
}
```