LUCRAREA 4

Scopul lucrării *este studiul operatorilor de atribuire, de conversie explicită (cast) operatorul de secvenţiere, typedef şi instrucţiunea de decizie (if-else).*

I. OBSERVAŢII TEORETICE

1.1. Operatorul de atribuire (continuare)

În C există o serie de reguli în ce priveşte scrierea generală:
stânga = dreapta;

Este forma generică a instrucţiunii de atribuire. Atribuire înseamnă stabilirea unei valori operandului stânga în funcţie de partea dreaptă. În dreapta poate apărea un simplu operand, ca în:

```
a = b;
```

sau o expresie, ca în:

```
a = (a*(b-c) ) >> 2 & 15;
   // apar operatorii: înmulțire, deplasare dreapta, ŞI pe bit.
```

După cum se poate deduce, intuitiv, pentru a reuşi stabilirea unei valori operandului stânga, membrul drept trebuie mai întâi evaluat. Adică este necesară obţinerea valorii sale numerice. Aşadar, făcând legătura cu *laboratorul 3* (sub-punctul legat de clasificarea operatorilor), atribuirea are modul de asocierea dreapta-stânga (*right to left*). Iar clasa de prioritate este scăzută, aflându-se pe penultima poziţie, a *15*[-a]. Aceasta are ca efect luarea în considerare a acestui operator ultimul într-o expresie.

Pe lângă modul de asociere există şi o altă *regulă*. Legată de *tipurile de date* ale celor doi membri, drept şi stâng.

Să presupunem că în expresia următoare:

```
a = b/3 * (c>0 ? c : c--);
```

a şi *c* sunt de tip întreg, iar *b* este de tip real simplă precizie (`float`). Şi au valorile `a = 2, c = 4, b = 23.5`.

Care va fi tipul de dată şi valoarea membrului drept? În primul rând, trebuie ţinut seama de `prioritatea` operatorilor ce compun partea dreaptă. Parantezele rotunde schimbă ordinea de prioritate implicită, după cum am văzut în *Laboratorul* 3. Aşadar, ordinea de evaluare este următoarea, pas cu pas:

a. prima dată este evaluat operatorul condiţional dintre parantezele rotunde, expresia devenind:

`b/3 * d`, unde cu *d* am notat pe *(`c>0 ? c : c--`)*

b. expresia rezultată conţine doi operatori de aceeaşi prioritate (împărţirea şi înmulţirea). Dominant acum este modul de asociere al operanzilor la operator. Pentru ambii operatori acest mod este stânga-dreapta aşa că se porneşte în expresie de la stânga la dreapta. Expresia devine:

   `e * d`, unde cu *e* am notat pe `b/3`.

   Deoarece *b* este real, rezultatul împărţirii este tot real. Deci tipul de dată al expresiei din dreapta, până în acest moment este real simplă precizie.

c. în final, se atribuie lui *a* valoarea înmulţirii `e*d`.

Observaţi cum tipul de dată dreapta a devenit real.

Există o *regulă generală* pe care o adoptă compilatorul de ***C***. Pe scurt, aceasta spune că ***întotdeauna*** o expresie este 'promovată' (convertită) către tipul de dată ***cel mai cuprinzător***, dat de un operand, sau de rezultatul unei sub-expresii ce o compune. Sau, se stabileşte tipul unei expresii în funcţie de tipul ultimului operand acare apare în acea expresie.

Pentru exemplul dat, tipul de dată dreapta este real, din cauza lui *b*.

Cum se face, atunci, această atribuire? Tipul lui *a* este *întreg*, iar membrul drept este *real*.

Există şi aici un ***mecanism implicit*** de lucru, anume ***conversia implicită***. Compilatorul va face acum o conversie implicită a tipului real la tipul întreg. Implicită fiind, pot apărea erori de conversie, din cauza faptului că se încearcă trecerea către un tip de dată mai puţin cuprinzător. Este posibil un rezultat incorect, mai ales dacă apar şi semne în expresia din dreapta. În programul dat ca exemplu, se vede şi efectul unei expresii dreapta cu semn.

Ca şi în cazul operatorilor, putem influenţa şi aici comportarea predefinită a compilatorului, prin folosirea *operatorului de conversie explicită*, denumit *cast* (punctul de discuţie următor).

Există şi operatorii de atribuire aşa-zişi rapizi, de genul /=, += sau &=. Revedeţi tabelul priorităţii operatorilor din *lucrarea 3*.

De fapt scrierea explicită de genul:

```
a = a+b;
```

se ascunde, şi se foloseşte forma:

```
a += b;
```

La fel pentru toţi operatorii de atribuire de acest gen. Toţi aceştia au aceeaşi prioritate cu operatorul-mamă, `=`.

Observaţie importantă:

În partea stângă a unei atribuiri nu poate apărea decât o entitate adresabilă, adică o entitate pentru care s-a rezervat o zonă de memorie. Aşa cum este numele corect al unei variabile.

Este **interzisă** o scriere de genul:

```
7 = a;
```

şi este marcată cu mesaj de eroare din partea compilatorului, în care se spune ceva de genul: *Lvalue required*. Mesajul exact al lui *Dev_C++* este

30 C:\BC\MyApp\Laborator\Lab_4\lab4_1.cpp

Acelaşi lucru şi în situaţia în care în membrul stâng apare o expresie, ca în:

```
x+3 = 10;
```

Şi aici, după evaluarea membrului stâng obţinem o valoare, nimic altceva. Iar regula de sintaxă este încălcată.

Atenţie!

Unei *valori aritmetice* nu i se poate atribui o altă valoare, în primul rând pentru că nu este entitate adresabilă, şi în al doilea rând pentru că numele unei variabile nu începe cu număr (!!)

Reţineţi că entitatea din partea stângă a unei atribuiri are drept caracteristică ceea ce este denumit *Lvalue* (**valoare stângă**). Aceasta este de fapt adresa acelei variabile.

Iar membrul drept al atribuirii are o caracteristică proprie numită *Rvalue* (**valoare dreapta**). Valoarea dreapta este de fapt valoarea aritmetică (numerică) rezultată în urma unei evaluări.

Însăşi aceste caracteristici speciale spun ce poate să apară în partea stângă şi ce în partea dreaptă.

O variabilă simplă, deci care prin definiţie posedă *Lvalue* (adică adresa zonei sale de memorie) poate apărea - fără dubii - în partea stângă a atribuirii, dar şi în partea dreaptă, pentru că poate fi evaluată, rezultând o valoare. Este cazul atribuirii:

```
a = b;
```

unde compilatorul îl va înlocui pe *b*, în urma evaluării sale, cu valoarea efectivă pe care o are, în acel moment.

Încă o **observaţie**:

O constantă declarată cu ajutorul modificatorului *const*, **NU** poate apărea în partea stângă a unei atribuiri, pentru că nu are dreptul de a fi modificată, deci, într-un fel i se interzice - de către compilator - caracteristica de *Lvalue*.

Compilatorul marchează eroare pe linia atribuirii lui *b*:

```
const int b = 20; // valoarea lui b nu poate fi modificată,
   // echivalent cu a spune ca b nu poate apărea în
   // partea stângă a unei atribuiri.
b = 21;
```

Mesajul de eroare exact este:

39 C:\BC\MyApp\Laborator\Lab_4\lab4_1.cpp

1.2. Operatorul de conversie explicită

Acest operator ajută programatorul să realizeze, fără eroare, conversiile de la un tip de dată la altul, indiferent dacă se face către un tip de dată mai cuprinzător, sau către unul mai puţin cuprinzător.

În cazurile practice, programatorul cunoaşte dinainte expresia în care trebuie să folosească acest operator, şi o va face dacă există şansa ca evaluarea expresiei să genereze *efecte laterale*. Efectul lateral este acel comportament pe care nu-l putem prevedea, şi care poate apărea în timpul rulării programului, în funcţie de contextul particular de lucru.

Sintaxa acestui operator este:

(tip) variabilă;

sau

(tip) expresie;

unde prin *tip* se înţelege un tip de dată de bază, sau definit ulterior de către programator, eventual folosindu-se de tipurile C fundamentale. Este tipul ***către care*** se face conversia.

Exemplu:

```
int a;
float b = -23.5;
a = (int) b;
```

Dacă operatorul de conversie explicită n-ar fi apărut, *în funcţie şi de compilatorul pe care rulaţi liniile respective de program*, este posibil ca semnul să se piardă (!), sau să apară depăşiri de reprezentare, dacă cumva partea întreagă a variabilei reale nu poate fi reprezentată pe un întreg scurt.

1.3. Operatorul de secvenţiere (virgulă)

Este interesantă comportarea compilatorului în cazul acestui operator.
Să luăm ca exemplu expresia următoare:

```
a = (b -= 3, --c, a+4);
```

Modul său de execuţie este următorul, pas cu pas:

- prima dată se evaluează *b -= 3;* valoarea lui *b* scade, deci, cu *3*.
- a doua oară se evaluează *-- c*; valoarea lui *c* scade cu o unitate.
- în cele din urmă se evaluează *a+4* şi aceasta este valoarea care se atribuie lui *a*. Deci valoarea anterioară a lui *a* creşte cu *4*.

O expresie în care apare operatorul virgulă este posibilă şi în zona de început unui ciclu *for()*, ca în exemplul:

```
int pas = 0;
for(i=1, i-=3, j=i; i+j<10; i++)
{
 printf("\n la pasul %i avem: i= %d, j= %d", pas, i, j);
 pas++;   // reţin numărul pasului la care s-a ajuns.
}
```

După cum se vede, înainte de intrarea in ciclu, adică înainte de verificarea ***condiției*** `i+j<10`, se execută pe rând, una câte una, expresiile dintre virgule.

Adică:

- se stabileşte *i* la valoarea *1*;
- se scad trei unităţi din *i*, acesta ajungând la *-2*;
- se atribuie lui *j* valoarea curentă din i, adică *-2*.

Suma *i+j* dă, în acest moment *-4*. Condiţia de intrare în ciclu este adevărată, deci începe corpul de ciclare. La trecerea la pasul următor al buclei, întreaga expresie de început ***NU*** se mai execută (aşa funcţionează un ciclu *for()*), ci se testează doar condiţia de rămânere în ciclu. La fiecare nou pas, doar *i* creşte cu o unitate, *j* păstrându-şi valoarea de început, adică *-2*.

1.4. Operatorul dedicat definirii de tipuri - *typedef*

Pot apărea situaţii în care doar tipurile de dată fundamentale C nu sunt suficiente. De exemplu, o funcţie matematică de derivare numerică, ce se poate aplica unei ***clase*** întregi de funcţii reale de variabilă reală. Funcţia de derivare ar trebui să prezinte, pentru generalitate, un argument de tip ***pointer la funcție***. Acest tip de dată trebuie definit, înainte de a putea fi folosit.

Un alt exemplu este cel al tipurilor de date logice (boolean). C nu prezintă vreun tip logic predefinit. Dar dă posibilitatea ca, folosindu-ne de propriile tehnici de sintaxă, să ne putem compune (crea) propriile tipuri, dintre cele mai complexe.

Aşadar, pe lângă tipurile de bază, putem defini şi altele. Şi aici intervine acest operator special, *typedef* (denumirea sa provine de la ***type def****inition*).

Sintaxa este următoarea:

```
typedef tipBaza tipNou;
```

Observaţii:

1. Înainte de a veni cu noua denumire, trebuie să ne folosim de un tip de bază C.
2. În final apare 'punct şi virgulă', deci avem de-a face cu o instrucţiune de definire de tipuri

Exemple:

```
// pentru a lucra mai uşor cu tipul de dată int, îl redefinim
typedef int INTREG;
```

```
// tipul de dată logic (boolean)
typedef unsigned int BOOLEAN;
```

Ce trebuie reţinut este că:

- denumirea `tipNou` este un alias pentru tipul de dată pe care îl succede în instrucţiunea de definiţie anterioară.
- scrierea cu majuscule este o ***convenție***. Rostul ei este de a informa programatorul, în cazul unei lecturi de cod-sursă, asupra denumirilor noi, pe care cel care a scris programul le-a avut în vedere.

În exemplele date, `INTREG` este un alias pentru *int*, iar `BOOLEAN` este o altă denumire pentru `unsigned int`.

Şi în anumite fişiere header se aplică această convenţie. Este cazul numerelor întregi sau reale maxime, sau tipurilor de date speciale, cum este `FILE`. Deci şi creatorii compilatorului (limbajului) au recurs la aceasta tehnică de lucru.

Atenţie!

Este recomandabil să vă obişnuiţi într-un fel sau altul asupra scrierii, pentru a nu crea confuzie în propriile programe, la o lectură ulterioară.

1.5. Instrucțiunea de decizie - if (şi variantele if-else)

Începem în acest laborator discuţia supra unei instrucţiuni speciale: *if*. Discuţia va continua în laboratorul următor, cu varianta `switch-case`.

Regulile discutate acum vor fi valabile şi la instrucţiunile de ciclare, unde *condiția de intrare/ieşire din ciclu* este de fapt un *if*.

Există de nenumărate ori momente într-un program în care evoluţia sa ulterioară depinde de un moment prezent. Acel moment prezent poate însă avea o varietate de manifestări, şi este rolul nostru de a ghida execuţia programului pe calea corectă, în pe baza anumitor ***condiții***. aici apare rolul instrucţiunii de decizie *if*, care *ramifică* evoluţia programului în funcţie de *valoarea de adevăr* a condiţiei testate.

Noţiunea de *'valoare de adevăr'* am întâlnit-o deja în cazul operatorilor logici (generali sau pe bit). Semnificaţia sa este de ***`adevărat`*** sau ***`fals`***.

Atenţie!

Adevărat în C este interpretat drept orice evaluare cu rezultat ***`diferit de 0`***, iar fals este numai atunci când în urma evaluării unei expresii, valoarea sa devine `0`.

Sintaxa generală este:

```
if(conditieLogica) expresie;
```

sau

```
if (conditieLogica) expresie1;
else expresie2;
```

sau

```
if (conditie1) expresie1;
   else   if(conditie2) expresie2;
          else   if(conditie3) expresie3;
                 else …
```

Condiţia logică poate fi dată de o comparaţie (>,>=, <,<=) toţi operatori cu rezultat de tip *boolean*, de un operand ca atare, sau de o expresie.

În cadrul unui *if* pot apărea şi *operatorii logici generali*, combinaţi în funcţie de cerinţele problemei. Un exemplu corect de condiţie logică poate fi:

```
if( condA SAU (condB SI condC) )
```

unde valoarea de adevăr a condiţiei rezultă dintr-o *expresie*. Acea expresie este adevărată dacă *sau* `condA` *sau* sub-expresia (`condB SI condC`) este adevărată. Urmând tabelul de adevăr al operatorilor `ŞI / SAU` putem crea un tabel de adevăr al acestei expresii.

În situaţia în care un *if* are mai multe ramuri, atunci există o regulă de asociere a unui *else* la un *if*: anume: un *else* este asociat la instrucţiunea *if* cea mai apropiată. Instrucţiunea *if* cea mai apropiată poate avea unul din sensurile:

- dacă nu apar corpuri de instrucţiuni, atunci un *else* este legat la *if*-ul imediat anterior;
- dacă apar corpuri de instrucţiuni, atunci un *else* este asociat cu ultimul *if* fără pereche.

Exemple:

```
// o condiție poate fi formată dintr-un singur operand.
a = -4;
if(a)     // condiție adevărată
printf("\n valoarea;lui a: %i", a);

// condiția unui if poate fi o expresie
float a = 1.1, b = 3.3;
if(a-b) printf("\n Condiție adevărată");

// condiția este o comparație. Variantă de if cu mai multe ramuri
int a = 4, b = 1;
if(a>b) printf("\n a este mai mare decât b");
esle      if(a == b) printf("\n a este egal cu b");
else printf("\n a este mai mic decât b");
```

```
// corp de instrucțiuni. Modul de asociere al lui else la un if.
if(a>3) { // if_1
if(b<3) printf("\n A este: %i SI b este mai mic decât 3: %i", a, b);// if_2
// else printf("\n A este: %i SI b este: mai mare decât 3: %i", a, b);
}
else printf("\n a nu este mai mare decât 3");
```

Apariţia acoladelor în această ultimă variantă schimbă modul implicit de asociere al unui *else* la un *if*. Dacă acestea nu apăreau, ca în varianta:

```
if(a>3) // if_1
if(b<3) printf("\n A este mai mare decât 3:%i SI b este mai mic
decât 3:%i", a, b);       // if_2
    else printf("\n a nu este mai mare decât 3");
```

atunci regula de asociere este cea implicită, iar ultimul *else* s-ar asocia cu *if*-ul al doilea.

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 studiază clasa operatorilor de atribuire, prin rularea programului *lab4_1.cpp*. Ştergeţi pe rând comentariile de pe liniile 30 şi 39 şi

comparaţi erorile cu cele din ***paragraful 1.1***. Notaţi-vă şi reţineţi mesajele respective.

2.2. Conversia explicită (*cast*) se studiază prin rularea programului *lab4_2.cpp*. Prin folosirea acestui operator se poate modifica rezultatul unei expresii în funcţie de tipul de dată convertit al unei variabile. Notaţi valorile rezultatului pentru fiecare caz în parte.

2.3. Operatorul de secvenţiere (virgulă) este studiat în programul *lab4_3.cpp*. Comentaţi întâi valoarea lui *a* de pe linia 8 şi verificaţi apoi cu rezultatul execuţiei programului. Ştergeţi comentariile de pe linia 12 şi 26. Evaluaţi-l pe hârtie pe *i* din ciclul *for()* şi verificaţi-vă cu rezultatul programului.

2.4. Crearea de noi tipuri de date de bază cu ajutorul operatorului *typedef* se poate vedea în programul *lab4_4.cpp*. Care sunt tipurile de dată nou create? Ştergeţi comentariile de pe liniile 21 şi 24 şi notaţi-vă eroarea obţinută. Care este cauza erorii? Alegeţi valoarea 1 sau 0 pentru variabila *adevarat* şi notaţi-vă rezultatele programului.

2.5. Instrucţiunea de decizie *if-else* este studiată în programul *lab4_5.cpp*. Rulaţi programul şi notaţi rezultatele. Alegeţi apoi valoarea *0* pentru *a* şi notaţi-vă noile rezultate. Explicaţi rezultatele obţinute. Ce expresii apar drept condiţie pentru ramura *if( )*?

III. ÎNTREBĂRI

3.1. Scrieţi un test *if* prin care să determinaţi dacă sau nu variabila `numB` este `0`. În funcţie de rezultatul testului stabiliţi valoarea lui `Num` la `0`, respectiv la `numA` împărţit la `numB`.

3.2. Scrieţi un test *if* prin care să determinaţi dacă un caracter denumit `chIn` este un caracter hexazecimal valid. Caracterele hexazecimale valide sunt `0,…,9` şi `A,...,F` (revedeţi ***laboratorul 2***). Se cer doar condiţiile, fără instrucţiuni care să fie executate în caz de *Adevărat* sau *Fals*.

3.3. Scrieţi o funcţie denumită `numar_par()`, în care, folosind operatorul `modulo (%)`, să stabiliţi dacă un număr preluat ca argument este sau nu **par**. Funcţia va întoarce *0* dacă numărul este par, şi *1* în caz contrar.

3.4. Scrieţi un test *if* care determină, pentru început, dacă un număr este sau nu mai mare decât *0*. Apoi, dacă acesta este mai mare ca *0* determină dacă este *par* sau nu folosind funcţia creată la punctul *3.3*. Afişaţi unul din următoarele mesaje, pe baza testelor făcute:

a) Numărul este par şi mai mare ca 0
b) Numărul este impar şi mai mare ca 0
c) *Numărul nu este mai mare ca 0.*

3.5. Folosiţi conversia explicită astfel încât rezultatul următoarei expresii aritmetice

```
int a = 24, b=15;
int c = a/(a-b);
```

să fie real.

3.6. Răspundeţi la următoarele întrebări (legate de atribuiri):

a) `x` poate fi `Lvalue`, `Rvalue` sau *ambele*?
b) `y*3` poate fi `Lvalue`, `Rvalue` sau *ambele*?
c) `7` poate fi `Lvalue`, `Rvalue` sau ambele?
d) `z` poate fi `Lvalue`, `Rvalue` sau ambele?

IV. ANEXA - sursele complete ale programelor

```
// lab4_1.cpp
// Operatorul de atribuire. Cazuri posibile. Excepții.
#include<stdio.h>

int main(void){
int a = 2, b = 10, c = 4;

printf("\n Valorile inainte de prima expresie: a: %i, b: %i, c:%i", a, b, c);

// expresia in care apar parantezele rotunde.
 a = (a*(b-c) ) >> 2 & 15;
 // reluati expresia, pas cu pas, cf. prioritatii si asocierii operatorilor.
 printf("\n Valorile dupa prima expresie: a: %i, b: %i, c:%i", a, b, c);

// o alta expresie in care apar parantezele rotunde
 printf("\n\n Valorile inainte de a doua expresie: a: %i, b: %i, c:%i", a, b, c);
 a = b/3 * (c>0 ? c : c--);
 printf("\n Valorile dupa a doua expresie: a: %i, b: %i, c:%i", a, b, c);

// expresie cu operator de atribuire, in care lipsesc paranezele rotunde
 printf("\n\n Valorile inainte: a: %i, b: %i, c:%i", a, b, c);
 a = b/3 * c>0 ? c : c--;
 printf("\n Valorile dupa: a: %i, b: %i, c:%i", a, b, c);

// operatorii de atribuire 'rapizi'
 a /= 2;    // impartire lui 'a' la 2; 'a' primeste noua valoare.
 printf("\n Valoarea lui a: %i", a);
```

```
// în timpul studiului programului renunțați la comentariile următoare,
// recompilați și observați comportamentul compilatorului.
/* Erori. Exceptii
 7 = a;    // eroare de compilare. 7 nu are Lvalue (nu este nume de variabila)
 a+b = 10;
*/

// o constanta declarata folosind modificatorul const NU are Lvalue (!!)
/*
 {
  const int b = 20; // b nu poate f modificata, deci nu poate aparea in
                    // partea stanga a unei atribuiri.
  b = 21;
  }
*/
 scanf("%i", &a);   // pentru vizualizarea rezultatelor (tine loc lui getch())
}
```

// lab4_2.cpp
// Studiu asupra operatorului de conversie explicită.

```
#include<stdio.h>

int main(void){
 unsigned int a;
 float b = -23.5;

// conversia explicita.
 a = (int) b;
 printf("\n Valoarea lui a (conversie explicita): %i", a);

// conversia implicita
 b = -23.5;
 a = b;
 printf("\n Valoarea lui a (prima conversie implicita): %i", a);

// in functie de compilator, apare depasire de gama
 b = -65537.1;
// partea intreaga depaseste valoarea maxima posivbila de
// reprezentat in a (anume 65535 = (2 la puterea 16) - 1).
 a = b;
 printf("\n Valoarea lui a (a doua conversie implicita): %i", a);

// operatorul de conversie schimba tipul unei expresii, prin conversia unuia
// dintre operanzii acelei expresii
 {
    int a = 20;
    int b = -21;
    float rez;      // rezultatul impartirii lui 'a' la 'b'

    rez = a/b;
// fara operator de conversie. Rezultatul este 'catul'
// impartirii a doi intregi.
    printf("\n Valoarea lui rez, fara (cast): %i, %f ", (int)rez, rez);

// cu conversie explicita.
rez = (float)a/b;   // prima varianta
printf("\n Valoarea lui rez, cu (cast) - prima varianta: %f", rez);

rez = a/(float)b;   // a doua varianta
printf("\n Valoarea lui rez, cu (cast) - a doua varianta: %f", rez);
printf("\n Valoarea lui rez, cu (cast) - a treia varianta: %f",
```

```
(float)a/(float)b);
 }

// incheiere
 scanf("%i",&a);    // pentru vizualizarea rezultatelor la ecran.
}
```

// lab4_3.cpp
// Operatorul de secvențiere (opertorul virgulă).

```
#include<stdio.h>

int main(void){
 int a = 12, b = 5, c = 2;

 // operatorul virgula in expresii
 a = (b -= 3, --c, a+4);
 // b = 2; c=1; a=16.
 printf("\n Valorile variabilelor: a= %i, b= %i, c= %i", a, b, c);
// operatorul virgula in ciclu for(). Se folosesc doua corpuri de instructiuni
{
int pas = 0;        // de cate ori se trece prin ciclu?
int i, j;
// la intrarea in ciclu: i=-2; j=i=-2.
for(i=1, i -= 3, j = i; i+j<10; i++)
{
    printf("\n La pasul %i avem: i= %d, j= %d, iar (i+j)=%i",
pas, i, j, i+j);
    pas++;          // tin numarul pasului la care s-a ajuns.
    }
printf("\n Numarul de treceri prin corpul ciclului: %i", pas);
}
// incheiere
 scanf("%i", &a);   // vizualizarea rezultatului la iesire.
}
```

// lab4_4.cpp
// typedef

```
#include<stdio.h>

typedef int INTREG;         // redefinirea lui 'int' drept 'INTREG'
typedef unsigned int BOOLEAN;       // definirea noului tip de data 'BOOLEAN'

int main(void){
 INTREG a = 20, b = -a, d=a&0xff;   // a=20, b=-20, d=a
 BOOLEAN adevarat = 1, fals = !adevarat;    // adevarat=1, fals=1

 printf("\n Valorile variabilelor intregi: a= %i, b= %i", a, b) ;
 {
    typedef int INT;// redefinirea intr-un corp de ciclu a tipului 'int'
    int c = 0, d = !c;      // c=0, d=1

printf("\n Valorile variabilelor c= %i, d= %i", c, d) ;
 }

 // valabilă acum este denumirea INTREG. Renunțați la comentariul următor și
 // observați acest lucru.
/* INTREG c = 12; // eroare de compilare: " 'INT' undeclared (first use this "
    printf("\n Valoarea variabilei c= %i", c) ;
*/
 printf("\n Valoarea variabilei d= %i", d) ;
```

```
// variabilele adevarat si fals in cazul unor decizii de tip 'if'
 if(adevarat) printf("\n\n Conditie adevarata.");
 if(false) printf("\n Ramura aceasta nu se executa. ");
 else  printf("\n Ramura aceasta se executa.");

// sfarsitul programului
 scanf("%i", &a);
}
```

// lab4_5.cpp
// Instructiunea de decizie - if

```
#include<stdio.h>

int main(void){
 int a, b;
// o conditie poate fi formata dintr-un singur operand.
a = -4;
if(a)       // conditie adevarata
printf("\n Conditie adevarata: valoarea lui a: %i", a);
b = !a;     // b ia valoarea logica 0 (!!)
if(!b)      // prin faptul ca aceasta ramura se executa, se justifica valoarea
// logica 0 pentru 'b', deoarece not(0) = 1.
printf("\n Conditie falsa: valoarea lui b: %i", b);

// conditia unui if poate fi o expresie
{
    float a = 1.1, b = 3.3;
    if(a-b) printf("\n Conditie adevarata.");
    else printf("\n conditie falsa.");
}

// conditia este o comparatie. Varianta de 'if' cu mai multe ramuri. Excluderi.
a = 4, b = 1;
if(a>b) printf("\n a este mai mare decât b");
else        if(a == b) printf("\n a este egal cu b");
    else printf("\n a este mai mic decât b");

// corp de instructiuni. Modul de asociere al lui else la un 'if'. Apar acolade.
if(a>3) {// if_1
if(b<3) // if_2
    printf("\n a este: a=%i SI b este mai mic decât 3: b=%i", a, b);
// else printf("\n a este: a=%i SI b este mai mare decât 3: b=%i", a, b);
}
else printf("\n a nu este mai mare decât 3");

// asocierea unui else la un if - fara acolade
if(a>3) // if_1
 if(b<3) // if_2
    printf("\n a este mai mare decât 3: a=%i SI b este mai mic decât 3: \
                                          b=%i", a, b);
else printf("\n a nu este mai mare decât 3");

// sfârşitul programului
 scanf("%i", &a);
}
```