LUCRAREA 2

Scopul lucrării *este familiarizarea studenților cu alfabetul limbajului, cuvintele-cheie, convențiile de denumire a variabilelor, tipurile de date de bază precum și declarația și definiția unei variabile.*

I. OBSERVAŢII TEORETICE

1.1. Cuvinte-cheie. Alfabetul limbajului

Ca în orice limbaj de programare există o mulţime de combinaţii (`simboluri`) care sunt `rezervate`, adică este interzisă folosirea lor în alte situaţii (sau `contexte` de lucru) decât cele impuse de compilatorul acelui limbaj.

Aceste simboluri rezervate acoperă atât tipurile de date de bază (pe care le vom studia la punctul *1.3.*), a modificatorilor acestora (acelaşi paragraf), cât şi situaţiile în care se impun decizii sau în cazurile în care se definesc tipurile de date omogene sau neomogene (enumerări, structuri şi uniuni). Vom vedea pe parcursul laboratoarelor exact despre ce este vorba.

Pentru comunicarea în C există o serie de `cuvinte` (`mulțimi de simboluri`) pe care compilatorul le aduce, predefinite, şi pe care un programator trebuie să le reţină. Prin exerciţii, el va trebui să facă asocierile necesare pentru a le folosi aşa cum se impune.

`Cuvintele-cheie` sunt:

auto	enum	short	while
break	extern	sizeof	
case	float	static	
char	for	struct	
const	goto	switch	
continue	if	typedef	
default	int	union	
do	long	unsigned	
double	register	void	
else	return	volatile	

Prima *remarcă* este aceea că numărul acestor cuvinte-cheie este rezonabil, adică nu atinge un număr exagerat de mare. Acesta constituie, fără îndoială, un avantaj.

Toate aceste simboluri au asociată o serie de `reguli de sintaxă`, care le defineşte complet modul de folosire, şi care cuprinde şi excepţiile de la regula generală.

Un exemplu este regula de sintaxă pentru `const`: acesta este un `modificator de format`, în sensul că aduce reguli în plus la folosirea tipurilor de dată de bază. Apare de obicei înaintea, dar poate şi succede, unui cuvânt-cheie ce specifică un tip de dată de bază (cum sunt `char`, `int` sau

`float`). Efectul este acela al declarării unei `constante` de acel tip, ceea ce înseamnă că, fără iniţializarea variabilei respective în momentul declarării, ea nu are nici un sens.

Un exemplu corect:

```
const int a = 10; // constantă în baza 10, inițializată la declarare
```

şi unul incorect:

```
const int a;    // aici apare marcată o eroare de sintaxă
```

În cel de-al doilea caz, nespecificând o valoare de început constantei `a`, nu o vom putea folosi decât după specificarea unei valori, adică:

```
a = 10;         // și aici apare o eroare de sintaxă
printf("\n Valoarea lui a: %i", a);
                // aici este folosită, pentru afișare, valoarea lui a.
```

Apare `contradicția`: `orice constantă` nu are dreptul de a apărea într-o `atribuire` - instrucţiunea C prin care se stabileşte o valoare pentru entitatea din partea stângă a semnului de egalitate. Deci, pe linia:

```
a=10;
```

vom avea un mesaj de eroare din partea compilatorului, de genul:

5 - C:\BC\MyApp\Laborator\lab2_1.cpp - uninitialized const 'a'

6 - C:\BC\MyApp\Laborator\lab2_1.cpp - assignment of read-only variable 'a'

Cifrele de la începutul rândurilor care denumesc eroarea arată numărul liniei din program unde a apărut acea eroare.

Observaţie:

În sursele salvate cu extensie `.cpp` compilatorul (fiind unul propriu C++) sunt valabile şi unele dintre funcţiile considerate ne-standard, împreună cu fişierele header asociate (v. `getch()` alături de `conio.h`).

Modificatorul `short` este aplicabil doar tipului întreg (`int`).

După cum vedeţi, există, bine stabilite, anumite caracteristici şi comportamente ale compilatorului, în diferite contexte de lucru, aşa cum aminteam la început. Rolul exerciţiilor de pe parcursul laboratoarelor este acela de a evidenţia aceste particularităţi.

! Luaţi toate notiţele pe care le consideraţi necesare pentru a reţine cât mai multe dintre aspectele întâlnite. Rapiditatea şi modul de înţelegere depind şi de felul în care vă marcaţi prin comentarii aceste erori. !

1.2. Convenţiile de denumire ale variabilelor

Într-un program C vor exista, în 99% dintre cazuri (majoritatea absolută), variabile sau constante de lucru. Pentru a le putea folosi va trebuie să le dăm nume. Numele variabilelor fac parte din clasa identificatorilor. Printre ei mai apar şi numele funcţiilor, numele vectorilor şi numele tipurilor de date omogene/neomogene (enumerări/structuri, uniuni).

Regula de bază este: identificatorii încep cu literă sau caracter de subliniere (underscore), şi au o lungime de cel mult 32 de caractere. Din acestea, compilatorul consideră semnificative doar primele şase. Apare un compromis: un nume de variabilă este bine să spună despre ce este vorba, caz în care am dori să utilizăm atâtea caractere (în limita celor 32) câte ne sunt necesare pentru a-i explica rolul. Aici intervine un inconvenient: dacă acea variabilă este des folosită, a-i repeta numele devine dificil.

Compromisul este acela de a împăca modul de denumire cu acela al semnificaţiei şi numărului de apariţii în program al acelei variabile. Există deja consacrate anumite variante de denumire ale variabilelor. De exemplu literele i, j, sau k sunt de obicei dedicate contoarelor - adică variabilelor de tip întreg care controlează numărul de rulări ale ciclurilor de tip for() sau while().

Exemplu:

```
for(i=10; i>=1; i--) printf("\n pasul: %i", i);
```

sau:

```
i=10;
while(i>=1)
{
   printf("\n pasul: %i", i);
   i--;  // echivalent cu i = i-1;
}
```

Am folosit aici două cuvinte-cheie, for şi while. Ele nu pot apărea în alte situaţii decât cele pentru care au fost proiectate. Dar putem apela la artificii de notaţie, precum _for sau __while. Observaţi că identificatorii sunt corecţi (încep cu _) şi au şi o semnificaţie imediată, denumirea lor făcând imediat referire la cuvintele consacrate deja de către compilator.

Numele unei variabile poate fi ales (în mod expres) exagerat, dar corect. Afişările din program ale unei variabile de tip întreg se pot face în variantele:

- cu specificatorul %x – având efect un rezultat în baza 16.
- cu specificatorul %o – având efect afişarea în bază 8,
- cu specificatorul %c, prin care se afişează efectiv simbolul alfa-numeric asociat numărului zecimal (de exemplu codul ASCII 97 este atribuit caracterului 'a' minuscul).

Caracterele mici (*lowercase*) sunt situate în tabela ASCII ***după*** cele mari (majuscule = *uppercase*) (!).

Tabela ASCII este o colecţie de numere, prin care se realizează o codificare necesară lucrului pe calculator. Studiind-o veţi observa cum fiecare simbol cunoscut de noi din scrierea

obişnuită are asociat un cod numeric. Codificarea constă tocmai în această asociere între o reprezentare numerică şi una simbolică.

Ca un exemplu pentru convenţia de denumire a variabilelor, în programul *lab2_3.cpp*, sunt raportate următoarele erori:

```
6 lab2_3.cpp - stray '$' in program
6 lab2_3.cpp - redeclaration of `char
```

Pe linia 6 este raportată o eroare deoarece în numele variabilei apare caracterul $, interzis a face parte dintr-un identificator.

Înlăturaţi semnele de comentariu de pe linia 6 şi de pe liniile 13-17 şi recompilaţi. Veţi observa chiar erorile amintite mai sus.

1.3. Tipurile de date de bază

Ca în orice limbaj de programare, şi în C există un set (mulţime) de tipuri de date predefinite (fundamentale sau de bază).

De această mulţime de tipuri de date depinde şi flexibilitatea limbajului: sau există un număr important de tipuri de date, împreună cu o clasă de operatori asociaţi, proprii acelor tipuri - caz în care limbajul este de nivel înalt şi flexibil, putând fi folosit într-o gamă extinsă de aplicaţii, sau se restrânge clasa tipurilor de bază la cele fundamentale (caracter, întreg, real), situaţie în care limbajul este mai puţin flexibil, şi deci de nivel mai scăzut (cu exprimare relativ greoaie).

În C există, prin comparaţie cu C++, o clasă apropiată de tipuri de date, însă o mulţime restrânsă de operatori asupra tipurilor. De aici şi clasificarea sa drept de *nivel mediu*.

Tipurile de date de bază sunt:

Numele tipului	***Spaţiul de memorie (Bytes)***	***Gama de valori***
char (implicit signed)	1	-2^7, $2^7 - 1$
unsigned char	1	0, $2^8 - 1$
int (implicit short şi signed)	2	-2^{15}, $2^{15}-1$
unsigned int (implicit short)	2	0, $2^{16}-1$
long int (implicit signed)	4	-2^{31}, $2^{31}-1$
unsigned long int	4	0, $2^{32}-1$
float (semn obligatoriu)	4	Real în simplă precizie.
double (semn obligatoriu)	8	Real în dublă precizie.

long double	10	Real în dublă precizie extinsă.
void	-	Are semnificația de ***nimic*** sau ***orice,*** funcție de context.
tip *pTip (tipul pointer)	Spațiul asociat lui tip	-
tip tablou[DIM] (tipul vector 1-dimensional)	DIM*sizeof(tip)	-
Tip tablou[Dim1][Dim2]…[DimN] (tip tablou multidimensional)	Dim1*Dim2*Dim3*…*DimN*sizeof(tip)	-

struct nume
{
tip1 câmp1;
tip2 câmp2;
…
}
	sizeof(struct nume)
	(se ține cont de eventualele alinieri în memorie)
		-

enum nume{ const1[=init1][, const2[=init2],…]} (tipul enumerare)	Compilatorul tratează valorile enumerare drept întregi.	-

union nume
{
tip1 arg1[,
tip2 arg2,…]
}
(tipul uniune)
	La un moment dat se folosește doar una dintre variabilele pe care le cuprinde uniunea. Spațiul de memorie alocat este dat de:
	Max(sizeof(tip1), sizeof(tip2),…)
		-

Ceea ce trebuie reţinut de aici este:

- fiecare tip de dată are rezervat un număr de biţi (sau, echivalent, de octeţi);
- pentru anumite tipuri de date - cele aritmetice (`char` şi `int`)- există reprezentări cu şi fără semn;
- doar între tipurile de date matematice (`int`, `float`, `double`, `long double`) se pot face conversii, adică treceri dintr-un tip într-altul.
- pe lângă tipurile matematice există şi tipuri de date mai complexe, catalogate drept:
    i. omogene (vectori, matrice, enumerări) ce conţin elemente de acelaşi tip,
    ii. neomogene (structuri şi uniuni) ce pot conţine tipuri de date diferite.

Operatorul specific `sizeof` întoarce `numărul de octeți` ai tipului de dată asupra căruia se aplică.

Înainte de a ne referi la spaţiul de reprezentare, să vedem care sunt puterile lui 2 care ne oferă cele mai apropiate numere de puterile lui 10 (ordinele de mărime).

Astfel:

$1K = 1000|_{10} = 2^{10} = 1024$ (kilo)

// observaţi un offset de 24 (faţă de valoarea de 1000), care se
// propagă la înmulţirile ulterioare.

$1M = (1k)^2 = 1\ 000\ 000|_{10} = 2^{20} = 1\ 048\ 576$ (mega)
$1G = (1K)^3 = 1\ 000\ 000\ 000|_{10} = 2^{30} = 1\ 073\ 741\ 824$ (giga)
$1T = (1M)^2 = (1K)^4 = 1\ 000\ 000\ 000\ 000|_{10} = 2^{40} = 1\ 099\ 511\ 627\ 776$ (terra)

Să facem o discuţie asupra spaţiului de reprezentare al tipurilor din familia întregilor. După cum s-a văzut în Laboratorul 2 - "*Sistemul de numeraţie binar*", poziţia din extrema stângă într-o reprezentare în baza 2 poartă denumirea de `Most Significant bit` (`MSb`) şi are rolul de semn. `0 = plus`, şi `1 = minus`.

Cum fiecare bit adăugat într-o reprezentare binară are ca efect dublarea gamei de reprezentare de până în acel moment, operaţia inversă, de renunţare (rezervare) la o poziţie din reprezentare (şi acesta este `bitul de semn`) conduce la înjumătăţirea gamei de reprezentare. Aşadar, dacă considerăm bitul `MSb` drept bit de semn, pentru tipurile `char` şi `int` (cu modificatorul `signed`, implicit) rămâne un număr de biţi util cu unul mai puţin decât întregul spaţiu de biţi rezervat.

Concret:

- pt. `int` există rezervaţi `2B`, adică `16b`. Dacă calculăm numărul de combinaţii ce pot fi reprezentate cu toţi cei `16b` ajungem la:
  $2^{16} = 2^6 * 2^{10} = 64K$.
  Numărul `64K` reprezintă, de fapt, următoarea valoare zecimală:
  $64 * 2^{10} = 64 * 1024 = 65\ 536$.

Acum, dacă facem convenţia de semn, atunci `MSb` nu mai este poziţie utilă în reprezentări, astfel că numărul de combinaţii - exceptând, deci, poziţia semnului - poate fi:

$2^{15} = 32K = 32\ 768$.

Trebuie observat că există `32K` de combinaţii pozitive şi tot atâtea negative. Adică `32K` de reprezentări cu `MSb` pe `1`, şi `32K` de reprezentări cu `MSb` pe `0`.

Încă o **observaţie**: toate numerele negative au un corespondent (*complement*) calculat pe baza relaţiei:

$/N = 2^n - N = N|_{\text{complement de 1}} + 1$.

Prin notaţia `/N` am simbolizat `valoarea complementară` a lui `N`, adică valoarea `negată`.

Exemple:

Pentru programul *`Lab2_4.cpp`*, în varianta prezentată se raportează următoarele avertismente (`warnings`):

10 lab2_4.cpp - [Warning] initialization of negative value
10 lab2_4.cpp - [Warning] argument of negative value `-16'

De obicei, avertismentele sunt mai subtile decât erorile, pentru că în principiu ele apar în situaţiile în care se încalcă "uşor" (la limită) anumite reguli "de bun simţ". Este şi cazul liniei 10 din acest program, în care variabila a fost declarată ca fiind fără semn - specificator `unsigned` -, iar valoarea de iniţializare a avut, totuşi, semn. Acest lucru este posibil, cu condiţia de a nu depăşi capacitatea de reprezentare pentru tipul de dată respectiv.

Încercaţi câteva exerciţii pe hârtie pentru trecerea unui număr în complementul său.

Lăţimea implicită în octeţi a tipurilor întregi `depinde de compilatorul` pe care se lucrează. De aceea, la trecerea unui cod sursă de pe un calculator pe altul, în situaţia în care compilatorul este de aşteptat să fie altul, trebuie ca programatorul să-şi asigure portabilitatea folosind operatorul `sizeof`.

În ce priveşte numerele reale, aici situaţia este diferită: `orice` compilator respectă un standard de reprezentare a numerelor reale (cu denumirea `IEEE 754`) în care spaţiul de memorie rezervat fiecărui tip de dată este fixat şi nu poate fi modificat.

Există un efect similar dacă operatorul `sizeof` este aplicat asupra denumirii tipului de dată, sau asupra unei variabile de un anumit tip. În situaţia:

```
float real = -1.4; // declarația unei variabile reale simplă precizie.
printf("\n Număr octeți pt. float: %d, %d", sizeof(float), sizeof(real));
```

se va afişa de două ori `4` (numărul de octeţi ai lui `float`).

Studiaţi programul *`Lab2_4.cpp`*, modificându-l. Păstraţi însă o copie a originalului.

1.4. Declaraţia şi definiţia unei variabile

Prin `declarația` unei variabile se face de fapt o asociere între un nume (corect) şi un tip de dată. În acelaşi timp se stabileşte (se rezervă) şi o zonă de memorie, având lăţimea dată de numărul de octeţi asociaţi tipului de dată ce apare în instrucţiunea de declaraţie.

Dacă în momentul declarării se face şi o iniţializare a variabilei respective, atunci declaraţia se numeşte `definiție`, pentru că, `static - adică în momentul compilării` - se rezervă spaţiu de memorie pentru a putea stoca valoarea de iniţializare. Opus variantei statice de rezervare a memoriei este `varianta dinamică`, atunci când stabilirea lăţimii zonei de memorie se face `în momentul rulării` programului, folosind `funcții de bibliotecă` specifice acestei operaţii (numită alocare de memorie).

În toate programele rulate până în acest moment s-au făcut astfel de declaraţii de variabile. Pentru a putea afişa conţinutul acestor variabile, adică valoarea lor efectivă, funcţia `printf()` foloseşte numele variabilelor declarate. Adică trebuie să cunoască zona din memorie (spaţiul rezervat) unde există fiecare dintre acele variabile, şi unde sunt salvate (scrise) valori compatibile cu tipul lor. Această asociere între numele unei variabile şi o zonă de memorie este efectul instrucţiunii de declarare/definire a variabilei.

Concluzie: în urma declarării/definirii, prin folosirea numelui unei variabile, compilatorul realizează asocierea cu zona de memorie rezervată acelei variabile.

Observaţie:
În ANSI C, într-un `corp de instrucțiuni` (cel delimitat de acolade) toate declaraţiile/definiţiile `de variabile` apar primele, fiind plasate imediat după acolada deschisă. Toate celelalte instrucţiuni aparţinând programului (afişare, calcule ş.a.) succed declaraţiilor/definiţiilor de variabile.

1.5. Detalii de programare

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

1.6. 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. Definirea unei constante. Rulaţi programul `lab2_1.c`. Care sunt erorile din program referitoare la definirea constantei? Comparaţi-le cu cele din expunerea teoretică. Definiţi corect constanta.

2.2. Definiţi două variabile de tip contor. Rulaţi programul `lab2_2.cpp`. Observaţi efectele celor două instrucţiuni de ciclare. Comparaţi rezultatele.

2.3. Declararea/definirea corectă şi incorectă a variabilelor. Rulaţi programul `lab2_3.cpp`. Remarcaţi lungimea numelui variabilelor. Renunţaţi la comentariile de pe liniile 6 şi 14, 15, 16. Notaţi erorile de sintaxă. Readuceţi apoi la codul sursă la forma corectă. Modificaţi acum numele acelei variabile în:

```
$denumireExageratDeLunga_DarCorecta
```

sau în

```
denumireExageratDeLunga_Incorecta$
```

Observaţi şi notaţi erorile compilatorului.

2.4. Încercuiţi identificatorii ***corecţi*** din următoarea listă de nume:

`new_num` `2ndPlace` `Fourth` `4th` `Void`

2.5. Încercuiţi identificatorii ***incorecţi*** din lista de nume următoare:

`old-num` `place2` `Fifth` `5th` `void`

2.6. Definirea/declararea variabilelor de diferite tipuri de dată fundamentale, conform tabelului prezentat în secţiunea "*I. OBSERVAŢII TEORETICE*". Rulaţi programul `lab2_4.cpp`. Comparaţi rezultatele cu modalităţile specifice fiecărui tip prezentate în tabel. Notaţi avertismentele şi comparaţi-le cu cele din expunerea teoretică.

2.7. Ordinea în interiorul unui corp de instrucţiuni – declararea/definirea variabilelor apar întotdeauna primele. Rulaţi programul `lab2_5.cpp`. Apar două instrucţiuni de afişare: una `înaintea` declarării unei variabile întregi, şi alta `ce urmează` acelei definiţii. Compilatorul fiind însă unul de C++ (`Developer C++ - DevC++`), în care restricţia aceasta nu mai este valabilă, nu marchează nici măcar cu avertisment această încălcare a standardului `ANSI C`. Dacă însă se schimbă extensia programului, în `.c`, şi se compilează cu un compilator pur `C`, atunci această restricţie devine, evident, valabilă.

III. ÎNTREBĂRI ŞI EXERCIŢII

3.1. Daţi patru exemple de cuvinte rezervate în C.

3.2. Ce rezultat au la ieşire următoarele instrucţiuni de afişare:

```
printf( "-\nHello There\n" );
printf( "The number is %d", 6 );
printf( "\tGoodbye\n" );
```

3.3. Scrieţi instrucţiunea (singulară) prin care se face citirea unui întreg (denumit `Num`) şi a unui caracter (denumit `chIn`) de la un utilizator ce lucrează la dispozitivul standard de intrare (codificat în C prin `stdin` şi reprezentând tastatura).

3.4. Scrieţi operatorii C pentru operaţiile logice `NOT (NU = negație)`, `AND (ȘI)`, `OR (SAU)`.

3.5. Declaraţi patru variabile diferite, de tipuri distincte, şi daţi un exemplu pentru fiecare (o simplă instrucţiune de folosire a lor). Aceste variabile trebuie să poată conţine, respectiv: literele alfabetului, un număr ce conţine parte fracţionară, un număr a cărui

valoare este întotdeauna pozitivă şi mai mică decât 65.000, şi o valoare întreagă cu valori posibile între -1.000.000 şi +1.000.000.

Notă:
Compuneţi câte un scurt program pentru fiecare întrebare. Compilaţi-l şi rulaţi-l, marcându-vă eventualele erori care apar.

IV. ANEXĂ - sursele complete ale programelor

```
// lab2_1.cpp
   #include<stdio.h>
   #include<conio.h>

   int main(void){
   const int a = 10; // constanta 'a' ia valoarea invariabilă 10;

    printf("\n Valoarea lui a: %i", a);
    getch();    //așteaptă caractere de la intrarea standard (stdin=tastatura)
   }

// lab2_2.cpp
   #include<stdio.h>
   #include<conio.h>

   int main(void){
    int i; // nu este necesara neaparat initializarea acestei
         // variabile (NU este o constantă).

    for(i=10; i>=1; i--)
     printf("\n for: Pasul: %i", i);

   // echivalentul while al ciclului for() anterior
   i=10; // O noua intializare. Ce valoare are i dupa executia primului ciclu?
    while(i>=1)
   {
       printf("\n while: Pasul: %i", i);
       i--;
   }
    printf("\n i are acum valoarea: %i", i);
    getch();
   }

// lab2_3.cpp
   #include<stdio.h>
   #include<conio.h>
```

```
int main(void){
 char denumireExageratDeLunga_DarCorecta;

// La următorul comentariu se va renunța, în etapa de studiu
// a programului
/* char denumireExageratDeLunga_Incorecta$ = 'b';*/

 denumireExageratDeLunga_DarCorecta = 'a'; //cod ASCII hexa: 0x61 (97 in zecimal)

 printf("\n Valoarea hexa a caracterului declarat: %c este %x",
                  denumireExageratDeLunga_DarCorecta,
                  denumireExageratDeLunga_DarCorecta);

// Urmează un comentariu pe mai multe linii, la care se renunță
// în studiul programului (v. secțiunea II - Desfășurarea
// lucrării), pentru a-i observa efectele
 /*
  printf("\n Valoarea hexa a caracterului declarat: %c este %x",
      denumireExageratDeLunga_Incorecta$,
      denumireExageratDeLunga_Incorecta$);
  */
 printf("\n Afisare octal: %o", denumireExageratDeLunga_DarCorecta);
// exersati si pe foaie aceasta conversie de baza de numeratie.
getch();
}
```

// lab2_4.cpp

```
#include<stdio.h>
#include<conio.h>

int main(void){

// familia intregilor
 char ch = 'A';    // cod ASCII in hexa: 41|16 = 65|10
 int b = -16;      // reprezentare cu semn. La afisare este
                   // chiar -16.

 unsigned char ch1 = 'a'; // fara semn.
 unsigned int b1 = -16;  // desi numarul este negativ, deci in
      // reprezentarea interna exista MSb=1, afisarea ignora
      // acest bit, si ceea ce se obtine este complementul de
      // 2 al lui -16, adica: 2^16-16 = 65536-16 = 65 520.
 short int b2 = 1;
 b2 = -(b2 << 14);  // spatiu pe 16 biti, 15 utili (cu semn).
 unsigned short int b3 = 1;
 b3 <<= 15;   // spatiu pe 16 biti, toti utili (fara semn).

 long int b4 = 1;
```

```
    b4 <<= 31;   // spatiu pe 32b, din care 31b utili (cu semn).
    unsigned long int b5 = 1;
    b5 <<= 31; // spatiu pe 32b, din care toti utili (fara semn).

   // familia numerelor reale
    float r1 = -16.16; // in cazul numerelor reale semnul este MEREU prezent.
    double r2 = -16.16;
    long double r3 = -16.16;

   // afisari ale numerelor din familia intregilor
    printf("\n char: %c, int: %i", ch, b);
    printf("\n unsigned char: %c, unsigned int (hexa): %x", ch, b1);
    printf("\n short int (cu semn): %i, unsigned short int: %u", b2, b3);
    b2 ^= b2;
    b2 = -16;
    ch = (char)b2;     // conversie implicita, deci fara erori.
    printf("\n Dupa conversie: unsigned char: %x, unsigned int: %x", ch, b2);

    printf("\n long int (cu semn): %i, unsigned long int: %u", b4, b5);

   // afisari ale numerelor din familia numerelor reale
    printf("\n\n float: %f, double: %lf, long double: %e ", r1, r2, r3);

   // numarul de octeti pentru numere întregi
    printf("\n Valori implicite pentru clasa intregilor:char: %d,\
    int: %d, short int: %d, long int: %d",
    sizeof(char), sizeof(int), sizeof(short int), sizeof(long int));

   // numarul de octeti pentru numere reale
    printf("\n Valori implicite pentru clasa numerelor reale:\
       float: %d, double: %d, long double: %d", sizeof(r1), sizeof(r2), sizeof(r3));
   }
```

// lab2_5.cpp

```
   #include<stdio.h>
   #include<conio.h>

   int main(void){
    int varInt; // declararea unei variabile intregi: varInt.

    printf("\n Instructiune de afisare inaintea declararii/definirii de variabile");
    printf("\n Variabila 'varInt' are valoarea %i", varInt);
    varInt = 20; // definirea variabilei intregi varInt.
    printf("\n Instructiune de afisare ce urmeaza declararii/definirii de variabile");
    printf("\n Variabila 'varInt' are valoarea %i", varInt);
    getch();    // aștept caracter de la intrarea standard (stdin=tastatura)
   }
```