# LUCRAREA 6

***Scopul lucrării*** *constă în studiul mai amănunţit al instrucţiunilor de ciclare, (instrucţiunile* `while()` *şi* `do-while()`*) precum şi în iniţierea lucrului cu vectorii.*

## I. OBSERVAŢII TEORETICE

### 1.1. Instrucţiuni de ciclare cu test iniţial (*continuare*)

#### 1.1.1. Instrucţiunea `while( )`

În laboratorul anterior am discutat despre `for()`, clasificată drept instrucţiune de ciclare cu test iniţial. Testul se referă la momentul în care compilatorul evaluează expresia asociată `condiției`. Dacă aceasta este adevărată, atunci este permisă intrarea în corpul ciclului. Dacă nu, ciclul este ignorat, iar execuţia continuă cu instrucţiunea imediat următoare ciclului.

O instrucţiune echivalentă lui `for()` este `while()`. Aceasta face parte tot din clasa instrucţiunilor cu test iniţial. Traducerea lui `while` în română este de **cât timp**. De aici şi modul de funcţionare, care este complet identic cu cel de la `for()`. Corpul ciclului se execută **cât timp** condiţia este adevărată.
Se admit şi aici expresii pentru instrucţiunile de `start` şi `condiție`.

Trebuie însă făcută o **observaţie**:
Nu există aici, explicit, o instrucţiune pentru `pas`, (denumită astfel în discuţia de la `for()`). De aceea, pentru a evita situaţia unui ciclu infinit - aţi văzut comportamentul unei asemenea variante - este obligatoriu ca în corpul de instrucţiuni al ciclului `while()` să apară cel puţin una (o instrucţiune) prin care programatorul să se asigure că valoarea de adevăr a condiţiei nu rămâne mereu adevărată - odată ce se intră în ciclu, este clar că valoarea de adevăr a condiţiei a fost `True`.

**Exemplu**:
```
int i = 20;
while(i>=1){
   printf("\n Am ajuns la %i", i);
   --i;  // Asigur că valoarea de adevăr a condiției devine falsă
         // Ciclu descrescător.
}
```

Dacă este nevoie de două sau mai multe condiţii care controlează un ciclu `while()`, este util - ca sfat - să interpretaţi situaţiile pe condiţia de ieşire, adică

în varianta de `până când`, şi nu de `cât timp`. Urmează apoi traducerea în variantă `cât timp` prin folosirea operatorului `NOT (!)` asupra variantei `până când`.

Regula logică folosită este **relaţia lui deMorgan**.

```
not(A sau B) = not(A) si not(B)
```

Prin negaţie, operatorul conjuncţie (`ŞI`) devine disjuncţie (`SAU`), şi reciproc. Această relaţie se poate generaliza relativ uşor.

**Exemplu**:

```
// Ciclul rulează până când: (i este mai mare ca j) SAU (restul
// împărţirii lor întregi este 0)
#define SAU ||
#define SI &&
typedef unsigned int BOOLEAN;   // un nou tip de dată

int i = 3, j = 5;
BOOLEAN A = (i<=j);                    // Prima condiţie: adevărată
BOOLEAN B = (i%j == 0);         // A doua condiţie: falsă.

printf("\n În exteriorul ciclului: deci condiţie adevărată");
printf("\n Valoarea logica a lui A: %i", A);
printf("\n Valoarea logica a lui B: %i", B);

// pentru a respecta cerinţele impuse, condiţia (A SAU B) trebuie negată
while( !(A SAU B) ) {
A = i>j    // impun re-evaluarea condiţiei A (cu noile valori i şi j)
B = (i%j == 0); // impun re-evaluarea condiţiei B (cu noile valori i, j)

printf("\n Sunt în buclă: deci condiţie adevărată");
printf("\n Valoarea logica a lui A: %i", A);
printf("\n Valoarea logica a lui B: %i", B);

printf("\n Valoarea lui i: %i", i);
printf("\n Valoarea lui j: %i", j);
--j;              // mă asigur ca la un moment dat condiţia este falsă
getch();  // aştept un caracter
}
```

După cum se vede, dacă ni se impun anumite cerinţe de execuţie, sau dacă interpretăm drept normale condiţii de tip `disjuncţii` (care nu pot avea loc simultan, adică folosirea operatorului `SAU`), ori `conjuncţii` (care au loc simultan, adică folosirea operatorului logic `ŞI`), atunci modalitatea de a le interpreta drept `până când` utilizează operatorul negaţie logică - `NOT (!)`.

Se pot folosi operatorii logici, practic în orice variantă imaginabilă. Totul este să gândim corect - sintactic şi semantic -, preferabil anterior, condiţiile ciclurilor.

### 1.1.2. Echivalenţa `for()-while()`

Aşa cum am spus, funcţionarea lui `for()` este complet echivalentă cu cea a unui `while()`. Aceasta rezultă şi din echivalenţa *C* a celor două cicluri.

Etapele conversiei sunt:

a) Expresia de `start` din `for()` va apărea înaintea instrucţiunii `while()`;
b) Expresia pentru `condiție` devine condiţia lui `while()`;
c) Corpul de instrucţiuni al lui for() devine corp de instrucţiuni al lui `while()`;
d) Expresia pentru `pas` devine parte a corpului de instrucţiuni al lui `while()`.

**Exemplu**:
Ciclul for următor:

```
for(i = valInceput; par(i) || i<=20; i++)
   printf("\n Număr par, sau cel mult 20");
```

devine, în formă `while()`:

```
i = valInceput;
while(par(i) || i<=20){
   printf("\n Număr par, sau cel mult 20");
   i++;
}
```

## 1.2. Instrucţiune de ciclare cu test final

Există şi situaţii în care se preferă un mod de funcţionare un pic schimbat: se începe ciclul cu execuţia primului pas, după care urmează evaluarea condiţiei de test. În funcţie de valoarea sa de adevăr, ciclul se continuă sau nu.

### 1.2.1. Instrucţiunea `do-while()`

Astfel de comportare putem obţine prin folosirea lui `do-while`. Între cuvintele-cheie `do` şi `while` apare corpul de instrucţiuni, între acolade. Expresia pentru condiţia de test apare între parantezele rotunde ale lui `while`, din final. Apariţia din final a condiţiei îi impune funcţionarea amintită.

După întreg ciclul apare `punctul şi virgula`, spre deosebire de `for()` sau `while()`.

**Exemplu**:

```
…
char ch;        // pentru opțiunile ce formează meniul.
 do{
   printf("\n Optiunea 1: calcul ('c' sau 'C')");
```

```
    printf("\n Opțiunea 2: afisare ('a' sau 'A')");
    printf("\n Opțiunea 1: iesire program ('q' sau 'Q').");
    ch = getch();            // sau cu scanf: scanf("%c", ch);
} while(ch!='c' && ch!='C' && ch!='a' && ch!= 'A' && ch!='q' && ch!= 'Q');

switch(ch)
{
 case 'c':
 case 'C':      puts("\n\n Ati ales calcul");
          calcul();
          break;
 case 'a':
 case 'A':      puts("\n\n Ati ales afisare");
          afisare();
          break;
 case 'q':
 case 'Q':      puts("\n\n Iesiti din program");
          goto gata;   // salt la eticheta
 default: puts("\n Asa ceva nu exista!!");
}
gata:
 printf("\n\n am incheiat!");
...
```

Ca situaţii utile în care putem folosi un `do-while` se pot enumera:

- verificarea unei intrări din partea utilizatorului, cu revenire dacă nu se respectă cerinţele de intrare:
- impunerea funcţionării unui `meniu`, din care utilizatorul poate alege doar câteva opţiuni limitate; orice altă opţiune are ca efect reluarea afişării variantelor corecte, până în momentul în care alegerea este una dintre cele impuse;
- necesitatea execuţiei unui pas, înaintea testării condiţiei - de exemplu folosirea unei valori iniţiale (de început), în cazul unor aproximări de calcul numeric (de exemplu metode iterative de rezolvare a ecuaţiilor sau sistemelor liniare pătratice)

### 1.2.2. Echivalenţa `for()` - `do-while()`

Se poate echivala un `for()` şi cu un `do-while()`. Tot ce trebuie avut în vedere este expresia pentru `condiție`. Aici trebuie ţinut seamă de faptul că deja s-a executat un pas, deci corpul de instrucţiuni a fost deja rulat o dată.

**Exemplu**:

```
for(i=20; i >= 1; i--) printf\n for: Am ajuns la %i", i);
int i = 20;
do{
   printf\n do-while: Am ajuns la %i", i);
   --i;  // asigur că valoarea de adevăr a condiției devine falsă
```

```
   // ciclu descrescător.
} while(i>=1);
```

## 1.3. Vectori (introducere)

Vectorii fac parte din categoria mulţimilor de date **omogene**. Aceasta înseamnă că elementele conţinute sunt de aceeaşi natură. Pe lângă vectori, din aceeaşi clasă fac parte şi enumerările.

Prin contrast, există mulţimile de date **ne-omogene**, cum sunt structurile şi uniunile, în care elementele sunt de tipuri diferite, deci de natură diferită.

Un vector este complet de finit de tripletul
{*nume, tip de bază, număr elemente*}

În memorie, un vector apare ca o succesiune de locaţii. Dimensiunea fiecărei locaţii este dată de lăţimea în octeţi (`sizeof`) a tipului de bază al vectorului. Aceasta succesiune începe de la o adresă, adică de la un `marker` (numit **`pointer`**, în *C* şi nu numai) pentru o locaţie de memorie. Marker-ul acesta punctează prima locaţie dintre cele impuse de lungimea vectorului, care arată câte locaţii de acelaşi fel apar una după alta. Lungimea vectorului este **obligatoriu constantă**. Numele vectorului joacă rolul marker-ului amintit.

Dacă se foloseşte cumva o variabilă în locul constantei, se primeşte la compilare un mesaj de genul **`'Constant expression required'`**:

***34 C:\BC\MyApp\Laborator\Lab_6\lab6_6.cpp - variable-sized object `vect' may not***

**Concluzie**:
Din momentul declarării unei variabile de tip vector, lungimea acesteia **NU** se mai poate modifica pe parcursul programului. Acest fel de alocare (rezervare) de spaţiu de memorie poartă denumirea de **alocare statică**, adică `în momentul compilării`.

### 1.3.1. Declararea/definirea vectorilor

Declararea/definirea vectorilor se face printr-o `instrucțiune de declarație`, similară celor din cazul variabilelor obişnuite. Dimensiunea acestui trebuie să fie constantă.

De exemplu, pentru declararea unui vector numit `vect`, cu `8` elemente de tip `real simplă precizie`, se scrie:

```
float vect[8];  // dimensiunea 8 nu se poate modifica (!).
```

Apare o notaţie specială, care face ca această scriere să se refere la un vector şi nu la o simplă variabilă numită `vect`, anume **`perechea de paranteze drepte`**, între care apare constanta cu rol de dimensiune, ce specifică numărul

de elemente ale vectorului. Fără acea notaţie, instrucţiunea anterioară arată aşa:

```
float vect;
```

adică se declară o simplă variabilă reală, numită `vect`.

Pentru scrierea într-un element al vectorului se foloseşte **`operatorul de indexare`**, adică perechea de paranteze drepte între care apare un **`index`**. Indexul este o variabilă întreagă fără semn, cu rolul de specificare a poziţiei din cadrul vectorului. De exemplu pentru poziţia a doua se scrie

```
vect[1]
```

iar pentru poziţia a şasea se va scrie:

```
vect[5]
```

De ce mereu cu câte unu mai puţin? Pentru că implicit, compilatorul de *C* alocă (rezervă) şi poziţia cu index `0`, astfel că numărătoarea componentelor începe cu `0`.

O altă **variantă de declarare** foloseşte constanta pentru dimensiune definită cu ajutorul `#define`:

```
#define DIM 8
…
float vect[DIM];
…
```

Se observă că se declară mai întâi constanta `DIM` (pentru dimensiunea vectorului), după care se foloseşte acea constantă pentru definiţia vectorului.

Se mai poate scrie şi în felul următor:

```
const int DIM = 8;          // o constantă inițializată cu 8
float vect[DIM];
```

### 1.3.2. Iniţializarea componentelor unui vector

Pentru iniţializarea componentelor unui vector este consacrată folosirea unui ciclu `for()`. La fiecare pas al ciclului se stabileşte o valoare pentru una şi numai una dintre componentele vectorului, folosindu-se un `index variabil` (de obicei `i`). Valoarea poate fi stabilită, funcţie de context, fie de către utilizator, fie aleator.

**Exemple**:

***a)***

```
// componentele sunt date de către utilizator
int i;
float vect[8];
```

```
for(i=0; i<8; i++)
{
   printf("\n Dați valoarea componentei %i", i);
   scanf("%f", &vect[i]);
}
```

În loc de condiţia `i<8` se mai putea scrie `i<=7`, de această dată cu '`mai mic sau egal`', din cauza componentei de index `0`, de la care se porneşte numărătoarea componentelor (de la `0` la `7` sunt `8` numere).

***b)***

```
// componentele sunt date aleator
int i;
float vect[8];

for(i=0; i<8; i++) vect[i] = random(8);
```

Funcţia `random()`, cu prototip în `stdlib.h` generează o valoare aleatoare, aici în intervalul `[0, 7]`.

***c)***

```
// componentele sunt inițializate în momentul declarației/
// definiției.
// Se specifică dimensiunea vectorului.
int i;
float vect[8] = {0, 1, 2, 3, 4, 5, 6, 7};
for(i=0; i<=7; i++) printf("\n Valoarea inițială %i: %f", i, vect[i]);
```

***d)***

```
// Componentele sunt date aleatoriu. Nu se specifică explicit
// dimensiunea vectorului (!)
int i;
float vect[ ] = {0, 1, 2, 3, 4, 5, 6};
for(i=0; i<=7; i++) printf("\n Valoarea inițială %i: %f", i, vect[i]);
```

În situaţia exemplelor **`c)`** şi **`d)`** se observă o sintaxă specială (iniţializare rapidă) prin care componentele vectorului sunt iniţializate chiar în momentul declaraţiei vectorului. În situaţia **`c)`** se specifică şi dimensiunea vectorului, pe când în exemplul **`d)`** acest lucru lipseşte. Programatorul trebuie să aibă grijă să verifice mereu dimensiunea vectorului şi să nu depăşească limitele acestuia (spaţiul de memorie rezervat).

În situaţia **`d)`** compilatorul numără valorile de iniţializare, îşi stabileşte dimensiunea vectorului, dar mai departe nu mai face verificări de limită a vectorului. Aceasta cade în sarcina programatorului.

**Observaţii**:

- Cu ajutorul unui index variabil `i`, putem parcurge vectorul. fără însă a trece de (a depăşi) indexul cu valoare maximă, adică `DIM-1`.

- Pentru situaţiile **c)** şi **d)** anterioare, dacă nu sunt specificate atâtea valori de iniţializare cât să fie atinsă dimensiunea vectorului, în funcţie de compilator, restul componentelor rămase fără valoare sunt fie aleatoare (deci lăsate ne-iniţializate), fie iniţializate cu `0` sau cu altă valoare.

### 1.3.3. Parcurgerea unui vector

Parcurgerea vectorului înseamnă vizitarea tuturor componentelor acestuia, **una câte una**, fie plecând din stânga, fie din dreapta. Deja aţi văzut o parcurgere, în momentul iniţializării vectorului în varianta cu ciclu `for()`.

O **parcurgere clasică** este următoarea (dreapta-stânga):

```
const int N = 8;
int lungime;
for(lungime = N, i=lungime-1; i>=0; i--)
   printf("\n valoarea %i: %f", i, vect[i]);
```

Variabila `lungime` este iniţializată cu numărul total al componentelor vectorului.

Se poate face şi o parcurgere mai interesantă, `sărindu-se` (ignorându-se) anumite componente:

```
const int N = 8;
int lungime;
for(lungime = N, i=lungime-1; i>=0; i -= 2)
   printf("\n valoarea %i: %f", i, vect[i]);
```

Aici sunt parcurse componentele din `2` în `2`. Observaţi că valoarea lui `i` la fiecare pas scade cu `2`. Sunt afişate componentele cu indice: `7, 5, 3, 1`.

**Observaţii**:

- Dacă se întâmplă **să depăşim** vectorul în extrema stângă sau în cea dreaptă, programul are şansa să se blocheze, cu efecte imprevizibile asupra sistemului de operare. Acesta pentru că **se trece dincolo** de spaţiul rezervat vectorului, deci se ajunge în memorie în zone care de drept nu ne aparţin.
- O afişare, sau un calcul în contextul unei depăşiri de limită a vectorului poate genera **efecte secundare**, depăşiri de gamă de reprezentare, sau poate conduce la cazuri exceptate (gen `infinit/infinit`, `"Divide by 0"`, sau altele similare).

### 1.4. Detalii de programare

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

### 1.5. 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. Instrucţiunea cu test iniţial `while()` se studiază în programul *lab6_1.cpp*. Este folosit aici operatorul `typedef` pentru construcţia unui tip de dată `BOOL`, util pentru declararea variabilelor logice ce apar în condiţia singulară a ciclului. Condiţia de rămânere în corpul ciclului este una singură (o singură variabilă logică). Condiţia devine la un moment dat falsă cu ajutorul instrucţiunii din corpul ciclului ce modifică variabila $i$.

2.2. Un ciclu `while()` a cărei condiţie de rămânere în buclă este controlată `de două condiţii logice` este studiat în programul `lab6_2.cpp`. Este folosită relaţia lui **`deMorgan`** pentru transformările logice impuse de situaţiile practice. Condiţia logică este compusă din două sub-condiţii, legate prin operator logic `ŞI`. Revedeţi **`Lucrarea 3`** pentru tabelul de adevăr al operatorului logic `ŞI`.

2.3. Echivalenţa între cele două cicluri cu test iniţial, `for()` şi `while()` se analizează în programul `lab6_3.cpp`. Afişările de la ecran justifică această echivalenţă. Primul ciclu este `for()`, al doilea este ciclul

`while( )` echivalent. Revedeţi şi **`Lucrarea 5`** pentru sintaxa ciclului `for()`.

Justificaţi afişarea la ecran - în finalul ciclurilor `for()` şi `while()` - a valorii `21`. Ce trebuie modificat pentru ca această valoare să fie `20`, iar cea de start să fie `0`, astfel încât să rămână tot `21` de treceri prin ciclu.

2.4. Comportamentul instrucţiunii de ciclare cu test final, `do-while()` este studiată în programul `lab6_4.cpp`. Condiţia de rămânere în ciclu este controlată de o singură condiţie. Revedeţi **`Lucrarea 5`** pentru sintaxa instrucţiunii de decizie `switch-case`.

2.5. Echivalenţa `for()` - `do-while()` este tratată în programul `lab6_5.cpp`. Condiţia de rămânere este controlată de un test singular. Echivalenţa celor două construcţii de ciclare este asigurată prin afişările de la ecran, prin care se observă numărul de treceri identic în urma rulării celor două bucle.

2.6. Lucrul introductiv cu vectorii este studiat în programul `lab6_6.cpp`. Aici apar instrucţiunile de declarare a unui vector, de iniţializare a componentelor acestuia şi de parcurgere secvenţială a vectorului. Componentele vectorului sunt citite de la tastatură, deci sunt introduse de către utilizator.

Dacă renunţaţi la comentariile de pe liniile `21…31` puteţi observa ce se întâmplă în situaţia în care componentele unui vector sunt generate `aleatoriu`. Revenind la comentariul ignorat, dar renunţând acum la comentariul de pe liniile `33…42` compilatorul va genera eroare la linia `36`. Notaţi-vă această eroare şi comparaţi-o cu cea din expunerea teoretică, din **`paragraful 1.3`**. Se observă şi situaţiile în care vectorul este doar declarat/definit dar neiniţializat (liniile `45…53`) ca şi situaţia foarte utilă de iniţializare a componentelor unui vector în momentul declarării acestuia, prin utilizarea listei de valori de iniţializare (liniile `58…67` şi `73…79`).

În ultima parcurgere a vectorului apare o depăşire la dreapta, afişându-se incorect a opta componentă, inexistentă în vector. Corectaţi afişarea.

De fiecare dată parcurgerea s-a făcut cu ciclu `for()`. Modificaţi acest ciclu înlocuindu-l cu un ciclu `while()` echivalent, respectiv cu `do-while()` echivalent.

2.7. Un alt mod de obţinere a numerelor aleatoare diferit de cel din *random()* este prezentat în programul *lab6_7.cpp*.

2.8. Parcurgerea unui vector se poate observa în detaliu în programul `lab6_8.cpp`. Aici sunt tratate situaţiile excepţionale în care fie parcurgerea se face eronat, depăşindu-se numărul alocat de componente (liniile `19…21`) unde depăşirea apare din cauza generării

în condiţie a unor indecşi **`negativi`**, fie una dintre componentele vectorului este zero şi apare o împărţire la `0` (`Divide by 0`) în liniile `27…35`, fie parcurgerea nu se mai face secvenţial (componentă cu componentă) ci în salt, din doi în doi (liniile `40…46`).

Modificaţi ciclurile de parcurgere înlocuindu-le cu cele echivalente, conform discuţiei din prezentarea teoretică a acestui laborator. Generaţi o depăşire în zona componentelor `cu index mare`, adică spre sfârşitul vectorului. Reveniţi apoi la forma corectă a programului.

# III. ÎNTREBĂRI ŞI EXERCIŢII

3.1. Declaraţi un vector de elemente reale dublă precizie şi iniţializaţi-i componentele în momentul declarării cu următoarele valori: `7.0, -15.5, 21.2, -3.6` şi `0`. Declaraţi şi un al doilea tablou de dimensiune `MAXSIZE` (constantă), pentru care componentele vor fi date de utilizator.

3.2. Scrieţi un ciclu `do-while` (deci cu test final), care preia la fiecare pas un număr real dublă precizie. Fiecare valoare reală citită se va scrie într-o componentă a unui vector de elemente reale dublă precizie. Ciclul se opreşte atunci când se tastează `caracterul două puncte` (`':'`).

3.3. Scrieţi un ciclu `for()` care să copieze într-un vector destinaţie valorile dintr-un vector sursă, componentă cu componentă. Tipul de date conţinut de cei doi vectori va fi identic, şi este la alegere. Iniţializarea elementelor vectorului sursă se va face în două moduri.

3.4. Pentru vectorul următor

```
int vector[] = {-3, -2, -1, 0, 1, 2, 3};
```

afişaţi doar componentele de indice (rang) impar.

3.5. Pentru vectorul:

```
int vector[] = {-3, -2, -1, 0, 1, 2, 3};
```

afişaţi doar componentele de indici *4* şi *6*.

# IV. ANEXA – sursele complete ale programelor

**// lab6_1.cpp**
**// Exemplu de început pentru while() - condiție singulară.**

```
#include<stdio.h>

typedef unsigned int BOOL;             // tipul logic

int main()
{
 int i = 20;
 BOOL logic = (i>=1);

 printf("\n Valoarea logica a conditiei (inainte de intrarea in \
             ciclu): %i\n", logic);
 while(logic){
  logic = (i>=1); // testul conditiei.
  printf("\n Am ajuns la %i. Conditia are valoarea logica %i", i, logic);
    --i;   // asigur ca valoarea de adevar a conditiei devine falsa
           // ciclu descrescator.
 }

// final program
 printf("\n Gata...");
 scanf("%i",&i);
}
```

**// lab6_2.cpp**
**// Ciclu while() cu condiție dublă (prin generalizare: condiție multiplă)**

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

#define SAU ||
#define SI &&
#define NOT(x) !(x)             // functie macro (pseudo-functie)

typedef unsigned int BOOLEAN;   // nou tip de data

int main(void)
{
 // ciclul ruleaza pâna când : (i este mai mare ca j) SAU (restul
 // împartirii lor întregi este 0)

 int i = 3, j = 7;
 BOOLEAN A = (i>=j);            // Prima conditie: adevarata
 BOOLEAN B = (i%j == 0); // A doua conditie: falsa.

// pentru a respecta cerintele impuse, conditia (A SAU B) trebuie negata
 printf("\n In afara ciclului: deci conditie adevarata");
 printf("\n Valoarea logica a lui A: %i", A);
 printf("\n Valoarea logica a lui B: %i", B);

/ ciclu while cu conditie dubla/
  while(NOT(A) SI NOT(B)) {
```

```
   A = i>j; // impun re-evaluarea conditiei A (cu noile valori i si j)
   B = (i%j == 0);
       // impun re-evaluarea conditiei B (cu noile valori i si j)
   printf("\n Sunt in ciclu: deci conditie adevarata");
   printf("\n Valoarea logica a lui A: %i", A);
   printf("\n Valoarea logica a lui B: %i", B);

   printf("\n Valoarea lui i: %i", i);
   printf("\n Valoarea lui j: %i", j);
   --j; // ma asigur ca la un moment dat conditia ciclului devine falsa
   getch();              // astept un caracter
   }

// final program
 printf("\n Gata...");  scanf("%i", &i); return 0;
}
```

**// lab6_3.cpp**
**// Echivalenta for-while.**

```
#include<stdio.h>

int par(int);            // declaratia (prototipul) functiei

int main(void){
 int i, valInceput = 1;
 int r;    // pentru a evita dublul apel al functiei in corpul for() sau
           // while()

// Ciclul for() urmator …:
for(i = valInceput; (r=par(i)) || i<=20; i++){
    printf("\n par(%i): %i", i, r);
    printf("for: conditia de oprire: 'Numar par, sau cel mult 20': %i", i);
    }

// Pauza dintre cicluri
 printf("\n Dati un intreg..."); scanf("%i", &i);

// … devine, în forma while():
 i = valInceput;
 while((r=par(i)) || i<=20){
    printf("\n par(%i): %i", i, r);
   printf(" while: conditia de oprire: 'Numar par, sau cel mult 20': %i", i);
    i++;
    }

 // Exercițiu: justificați apariția valorii 21 la afișare (!)

 // final program
  printf("\n Gata..."); scanf("%i", &i); return 0;
}

int par(int nrInt){
 // varianta folosind operatia logica pe bit
 int mask = 0x0001;             // doar bitul LSb este interesant
 if(nrInt & mask) return 1;     // numar impar
```

```
 else return 0;
    // Numar par. else nu este neaparat necesar aici, in contextul
    // particular in care if-ul contine intructiunea 'return'.
    // Comentariile urmatoare dau o alta varianta de calcul a paritatii
    // unui numar
/*
 if(!(nrInt % 2)) return 0; // facem conventia ca functia 'par()' sa returneze
           // chiar valoarea restului impartirii intregi a lui 'i' la 2.
 return 1; // merge si cu else, dar pe aceasta ramura se ajunge doar daca
           // conditia din if este falsa.
*/
}
```

**// lab6_4.cpp**
**// do-while().**

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

void calcul(void);       // o functie
void afisare(void);      // alta functie

int main(void) {
char ch;   // pentru optiunile de la utilizator.

 do{
    printf("\n Optiunea 1: calcul ('c' sau 'C')");
    printf("\n Optiunea 2: afisare ('a' sau 'A')");
    printf("\n Optiunea 3: iesire program ('q' sau 'Q').");
    ch = getch();         // sau cu scanf: scanf("%c", ch);
 }
while(ch!='c' && ch!='C' && ch!='a' && ch!= 'A' && ch!='q' && ch!= 'Q');
switch(ch){
 case 'c':
 case 'C':        puts("\n\n Ati ales calcul");
           calcul();
           break;
 case 'a':
 case 'A':        puts("\n\n Ati ales afisare");
           afisare();
           break;
 case 'q':
 case 'Q':        puts("\n\n Iesiti din program");
           goto final;   // salt la eticheta
 default:  puts("\n Asa ceva nu exista!!");
}
final:
 printf("\n\n am incheiat!");
 scanf("%c", &ch);
 exit(0);  // iesire din intreg programul, cu transmiterea valori 0 catre
           // sistemul de operare.
}

void calcul(void) {
 printf("\n Ati apelat functia de calcul...");
```

```
 printf("\n Urmeaza un calcul oarecare...");
 int i = 10, r=1;
 do{
  r = 2*r;
  - - i;
 } while(i>=1);

 printf("\n 2 la puterea 10: %i", r);
}

void afisare(void){
 printf("\n\t\t -= un mesaj pentru afisare =-");
}
```

**// lab6_5.cpp**
**// Echivalenta for() - do-while().**

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

int main(void){
 int i = 20, pas=0;

 for(; i >= 1; i--) {
    printf("\n for: Am ajuns la %i", i);
    pas++;
    }
 printf("\n numarul de treceri prin for(): %i", pas);
 getch();

// ciclul echivalent do-while()
 i=20;
 pas ^= pas;      // reset.
 do{
    printf("\n do-while: Am ajuns la %i", i);
    --i;   // asigur ca valoarea de adevar a conditiei devine falsa
           // ciclu descrescator.
    pas++; // numar trecerile
    } while(i>=1);
 printf("\n numarul de treceri prin do-while(): %i", pas);

// final program
  printf("\n Gata..."); scanf("%i", &i); return 0;
}
```

**// lab6_6.cpp**
**// Declarația/definiția unui vector – inițializare.**

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

int main(void)
{
// componentele sunt date de catre utilizator
 int i;
```

```
{
 int i;
 float vect[8];   // vector cu 8 elemente

 for(i=0; i<8; i++) {
   printf("\n for1: Dati valoarea componentei %i: ", i);
   scanf("%f", &vect[i]);
   printf("Ati dat valoarea: %f", vect[i]);
   }
}

// Componentele sunt date aleator
/*
{
 int i;
 float vect[8];

 for(i=0; i<8; i++) {
   vect[i] = random(8);
   printf(" \n for2: valoarea aleatoare %i: %f", i, vect[i]);
   }
}
*/
// Renunțați la comentariile următoare și marcați-vă eroarea apărută.
// Notați-vă mesajul (mesajele) de eroare
/*
 // dimensiunea nu este constanta. Eroare !
 {
  int n = 8;
  float vect[n];

  for(i=0; i<8; i++)
    printf("\n for: Valoarea aleatoare %i: %f", i, vect[i]);
 }
 */

// vector pentru care nu sunt stabilite valori pentru componente.
{
 #define DIM 8    // se poate folosi si local
 int i;
 float vect[DIM]; // nu stabilesc valori pentru componente.
          // Ele pot fi initializate automat cu 0, de catre compilator
 for(i=0; i<8; i++)
   printf("\n for3: Valoarea aleatoare %i: %f", i, vect[i]);
}
 scanf("%i", &i);

// vector ale carui componente sunt initializate in momentul
// declaratiei/definitiei
{
 #undef DIM
 const int DIM = 8;             // se poate si local
 int i;
 float vect[DIM] = {0,1,2,3,4,5,6,7};
          // stabilesc valori pentru componente.
```

```
          // Ele pot fi initializate automat cu 0, de catre compilator
 for(i=0; i<8; i++)
   printf("\n for4: Valoarea initiala %i: %f", i, vect[i]);
}

 scanf("%i", &i);

 // vector ale carui componente sunt initializate in momentul
 // declaratiei/definitiei
 // NU se specifica dimensiunea vectorului.
{
 int i;
 float vect[ ] = {0, -11, -12, -13, 14, 15, -16};

 for(i=0; i<=7; i++)
   printf("\n for5: Valoarea initiala %i: %f", i, vect[i]);
}

// final program
  printf("\n Gata..."); scanf("%i", &i); return 0;
}
```

**// lab6_7.cpp**
**// Generare de numere (pseudo)aleatoare.**

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

#define GAMA 5

int main(void)
{
 int i;
 float start = 0.56;
 float vector[GAMA];


 for(i=0; i<GAMA; i++) {
  start = start * 1011;   // numarul 1011 este scris in baza 10.
  printf("\n Valoarea start: %f, si valoarea parte intreaga: %i",
                             start, (int)ceil(start));
  start = fabs(ceil(start) - start);
  vector[i] = start;
  printf("\n Valoarea aleatoare %i: %f", i, vector[i]);
  }

// final program
  printf("\n Gata..."); scanf("%i", &i); return 0;
}
```

```
// lab6_8.cpp
// Vectori: parcurgeri / depășiri / cazuri exceptate.
#include<stdio.h>

int main(void)
{
 const int N = 8;
 int i, lungime;
 float vect[N] = {0, -1.1, 2.2, -3.3, 4.4, -5.5, 6.6, -7.7};

// parcurgere dreapta-stanga. NU sunt depasiri
 for(lungime = N, i=lungime-1; i>=0; i--)
    printf("\n for1: valoarea %i: \t %11.10f", i, vect[i]);

// parcurgere dreapta-stanga. Apare depasirea spre stanga
// (se ajunge in memorie inaintea vectorului (!!)). Observati valorile aleatoare
// afisate.
// Atentie!: Programul are sanse sa se blocheze. Asa ceva trebuie evitat (!!)

 puts("");
 for(lungime = N, i=lungime-1; i>=-3; i--)
    printf("\n for2: valoarea %i: \t %11.10f", i, vect[i]);

// caz exceptat: 'Divide by 0'  = infinit (se imparte la o componenta nula)
// Asa ceva trebuie evitat (!!)
 scanf("%i", &i);
 puts("");
 {
  const int N = 8;
  int i, lungime;

  float vect[N] = {0, -1.1, 2.2, -3.3, 4.4, -5.5, 6.6, -7.7};
  for(lungime = N, i=lungime-1; i>=0; i--)
    printf("\n for3: valoarea %i: \t %11.10f", i, 1/vect[i]);
 }

// parcurgere cu salt din 2 in 2
 scanf("%i", &i);
 puts("");
 {
 const int N = 8;
 int i, lungime;

 for(lungime = N, i=lungime-1; i>=0; i-=2)
    printf("\n for4: valoarea %i: \t %11.10f", i, vect[i]);
 }

// final program
  printf("\n Gata..."); scanf("%i", &i); return 0;
}
```