Laborator 2 - Încapsularea

Tema 2.1

Să se analizeze programul `EX2.C`

Indicații 2.1

- ⇨ A nu se uita de fișierul `EX2.H`

Tema 2.2

Să se modifice funcţiile referitoare la cerc astfel încât parametrul `CERC` să fie transmis prin referinţă.

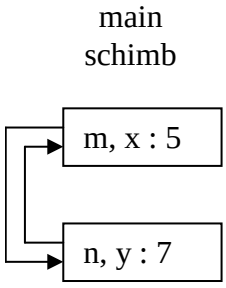
Consideraţii teoretice 2.2

La consideraţii teoretice 1.1 am prezentat faptul că funcţiile pot primi argumente în două feluri: apel prin valoare şi apel prin pointer. Acest lucru este valabil pentru compilatorul de C. Compilatorul C++ vine cu încă o metodă de transmitere a argumentelor: apelul prin referinţă.

Primele două moduri de transmitere a argumentelor le-am introdus prin intermediul unui exemplu, o funcţie care interschimbă două valori. Vom exemplifica acest nou mod de transmitere a argumentelor prin intermediul aceluiaşi exemplu.

```
#include <iostream.h>

void schimb(int &x, int &y)
{
 int r;

 r = x;
 x = y;
 y = r;
}

void main()
{
 int m=5,n=7;

 cout<<m<<"  "<<n<<"  ";
 schimb(m,n);
 cout<<m<<"  "<<n;
}
```

În acest caz variabilele locale *x, y* vor avea alocată memoria peste variabilele *m, n*. De fapt variabila *x* va ocupa în memorie exact aceeaşi zonă cu variabila *m*, la fel se întâmplă cu variabilele *y* şi *n*.



Perechile m, x şi n, y ocupând aceleaşi locaţii de memorie în momentul în care se face schimbarea de valori între x şi y automat acestă schimbare se reflectă şi pentru m şi n

Indicaţii 2.2

- ⇨ Se va forţa utilizarea compilatorului de C++ (de exemplu prin modificarea extensiei fişierului sursa în *.CPP)

Tema 2.3

Să se definească tipul `CERC` în context C++ (fără operatorul typedef)

Consideraţii teoretice 2.3

La consideraţiile teoretice 1.5 am prezentat modul de definire a unei structuri. Precizăm că acel mod de definire este specific compilatorului C (dar este totodată recunoscut şi de compilatorul C++**).** În C++**,** o dată ce a fost declarată o structură, se pot declara variabile de acel tip folosind doar numele structurii, fără să mai trebuiască să fie precedată de cuvântul cheie *struct*. Motivul acestei diferenţe între C şi C++ este acela că în C un nume generic de structură nu este un nume de tip complet, în timp ce în C++ este.

În concluzie în C++ prin intermediul lui *struct* se vor defini noi tipuri de date. Forma generală va fi tot cea prezentată la punctul 1.5.

```
struct nume_structura
{
 tip1 nume_membru1;
 tip2 nume_membru2;
 ...
} variabile_structura;
```

Memoria ocupată de o structură este egală cu suma memoriei ocupate de fiecare membru al structurii:

```
sizeof(struct)= Σ(sizeof(membrii))
```

Ne vom folosi în continuare de următorul exemplu: o structură `RATIONAL`, pentru reprezentarea numerelor raţionale (de forma m/n) şi doua funcţii: una de afişare şi una care face adunarea unui număr raţional cu un întreg.

```
#include <iostream.h>

struct RATIONAL
 {
  int numarator;
  int numitor;
 };

void Afisez(RATIONAL x)
{
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void Adun_int(RATIONAL &x, int y)
{
 x.numarator += y*x.numitor;
 Afisez(x);
}

void main()
{
 RATIONAL x;
```

```
 x.numarator = 4;
 x.numitor = 3;
 Adun_int(x,2);
}
```

Indicaţii 2.3

- ⇨ Atenţie la sintaxa de definire (la numele tipului şi la " ; ").

Tema 2.4

Să se mute funcţia `Muta` în interiorul structurii `CERC`

Consideraţii teoretice 2.4

Într-o structură se pot defini nu numai câmpuri de date ci şi funcţii. Forma generală a definirii structurii va deveni:

```
struct nume_structura
{
 tip1 nume_membru1;
 tip2 nume_membru2;
 ...
 tip_returnat1 nume_functie1(lista_parametrii1);
 ...
};
```

Declararea funcţiilor definite într-o structură se modifică astfel:

```
tip_retunat1 nume_structura::nume_functie1(lista_parametrii)
{
 corp functie;
};
```

Apelarea funcţiilor definite într-o structură se va face astfel:

```
nume_strctura variabila;
variabila.nume_functie1(parametrii);
```

Funcţiile definite într-o structură mai poartă numele de metode.

Dimensiunea memoriei ocupată de structură nu se modifică datorită adăugării de metode în structură:

```
sizeof(struct)= Σ(sizeof(membrii))
```

Revenim la exemplul prezentat la punctul anterior. Dorim să introducem în structura `RATIONAL` funcţia `Adun_int`. Exemplul va deveni:

```
#include <iostream.h>

struct RATIONAL
 {
  int numarator;
  int numitor;
  void Adun_int(RATIONAL &x, int y);
 };

void Afisez(RATIONAL x)
{
```

```
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void RATIONAL::Adun_int(RATIONAL &x, int y)
{
 x.numarator += y*x.numitor;
 Afisez(x);
}

void main()
{
 RATIONAL x;
 x.numarator = 4;
 x.numitor = 3;
 x.Adun_int(x,2);
}
```

Indicaţii 2.4

- ⇨ Apelul metodei Muta trebuie făcut folosind operatorul "." legat de un cerc.

Tema 2.5

Să se renunţe la parametrul de tip `CERC` din cadrul metodei `Muta`.

Consideraţii teoretice 2.5

În exemplul cu clasa `RATIONAL` apelul funcţiei `Adun_int` era sub forma:

```
x.Adun_int(x,2);
```

Se observă că variabila `x` apare de două ori: la început fiind variabila care apelează funcţia `Adun_int` şi apoi ca parametru al funcţiei `Adun_int`. Apărând în două locuri ne punem întrbarea dacă nu este posibilă renunţarea la `x` ca şi parametru? Pentru a putea face acest lucru vom fi nevoiţi să ne folosim de pointerul `this`.

Atunci cand este apelată o funcţie membru, i se pasează automat ca argument implicit pointerul `this`, care este un pointer către variabila care a generat apelarea (variabila care a invocat funcţia).

Exemplul nostru va fi modificat astfel:

```
#include <iostream.h>

struct RATIONAL
 {
  int numarator;
  int numitor;
  void Adun_int(int y);
 };

void Afisez(RATIONAL x)
{
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void RATIONAL::Adun_int(int y)
{
 this->numarator += y*this->numitor;
 Afisez(*this);
}
```

```
void main()
{
 RATIONAL x;
 x.numarator = 4;
 x.numitor = 3;
 x.Adun_int(2);
}
```

Indicaţii 2.5

- ⇨ Se va elimina parametrul de tip `CERC` şi se va înlocui cu cercul indicat de pointerul `this`.
- ⇨ Atenţie, `this` este pointer şi uneori avem nevoie de cerc nu de adresa sa.

Tema 2.6

Să se introducă în structura `CERC` şi celelalte funcţii care ţin de cerc în mod asemănător cu funcţia `Muta`.

Indicaţii 2.6

- ⇨ Atenţie la funcţia `OurInitGraph()`.
- ⇨ Se vor repeta temele 2.4 si 2.5 pentru fiecare din metodele respective

Tema 2.7

Să se renunţe la scrierea lui `this` acolo unde este posibil.

Consideraţii teoretice 2.7

La membrii unei structuri se poate obține acces direct din cadrul unei funcţii membru, fără nici un specificator de obiect sau structură.

`this->x` este echivalent cu `x`, unde `x` este un membru al structurii (atât câmp de date cât şi metodă).

```
#include <iostream.h>

struct RATIONAL
 {
  int numarator;
  int numitor;
  void Adun_int(int y);
 };

void Afisez(RATIONAL x)
{
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void RATIONAL::Adun_int(int y)
{
 numarator += y*numitor;
 Afisez(*this);
}

void main()
{
 RATIONAL x;
 x.numarator = 4;
 x.numitor = 3;
```

```
 x.Adun_int(2);
}
```

Indicatii 2.7

- ⇨ Deşi se poate renunţa la `this` nu uitaţi că de fapt **compilatorul îl foloseşte**.

Tema 2.8

Să se găsească un membru (dată sau metodă) al structurii `CERC` care poate fi făcut privat şi programul să meargă fără alte modificări.

Consideraţii teoretice 2.8

Într-o structură membrii pot fi de două tipuri: publici sau privaţi. Membrii publici sunt membrii care pot fi accesaţi atât de către metodele definite în structură cât şi de funcţiile din afara structurii. Membrii privaţi sunt membrii care pot fi accesaţi doar de către metodele definite în structură. Forma generală pentru definirea structurii va deveni:

```
struct nume_structura
{
 public:
    lista_membrii_publici;
 private:
    lista_membrii_privati;
 public:
    lista_membrii_publici;
 ...
};
```

Implicit membrii unei structuri sunt publici.

Indicaţii 2.8

- ⇨ Membrul respectiv nu trebuie să fie folosit din exteriorul structurii.

Tema 2.9

Să se introducă o nouă metodă care să permită definirea tuturor datelor din structura `CERC` private.

Consideraţii teoretice 2.9

Pentru a declara datele din structură private este necesară o funcţie publică care să facă iniţializarea datelor respective. Devenind private datele nu vor mai putea fi modificate decât cu ajutorul metodelor definite în structură.

Exemplul cu structura RATIONAL ar putea fi rescris astfel:

```
#include <iostream.h>

struct RATIONAL
 {
  private:
    int numarator;
    int numitor;
  public:
    void Adun_int(int y);
    void Initializez(int x, int y);
 };
```

```
void Afisez(RATIONAL x)
{
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void RATIONAL::Initializez(int x, int y)
{
 numarator = x;
 numitor = y;
};

void RATIONAL::Adun_int(int y)
{
 numarator += y*numitor;
 Afisez(*this);
}

void main()
{
 RATIONAL x;
 x.Initializez(4,3);
 x.Adun_int(2);
}
```

O justificare a ascunderii datelor în cazul exemplului nostru este aceea că altfel am putea risca să uităm să iniţializăm `numitor`-ul.

Indicaţii 2.9

- ⇨ Se va folosi o metodă publică (cu patru parametrii) care va initializa toate datele respective.

Tema 2.10

Să se transforme structura CERC în clasa CERC.

Consideraţii teoretice 2.10

În C++ o structură este echivalentă cu o clasă, singura diferenţă fiind aceea că într-o structură implicit membrii sunt publici iar într-o clasă membrii sunt implicit privaţi. Forma generală a definirii unei clase este:

```
class nume_clasa
{
 public:
    lista_membrii_publici;
 private:
    lista_membrii_privati;
 public:
    lista_membrii_publici;
 ...
};
```

O variabilă definită de tipul unei clase este un obiect, iar în loc de definire vom spune că instanţiem un obiect de tipul clasei.

```
#include <iostream.h>

class RATIONAL
 {
```

```
  private:
    int numarator;
    int numitor;
  public:
    void Adun_int(int y);
    void Initializez(int x, int y);
 };

void Afisez(RATIONAL x)
{
 cout<<x.numarator<<"/"<<x.numitor<<endl;
};

void RATIONAL::Initializez(int x, int y)
{
 numarator = x;
 numitor = y;
};

void RATIONAL::Adun_int(int y)
{
 numarator += y*numitor;
 Afisez(*this);
}

void main()
{
 RATIONAL x;           //instantierea obiectului x
 x.Initializez(4,3);
 x.Adun_int(2);
}
```

Indicaţii 2.10

- ⇨ Atenţie la tipul implicit al membrilor.

Tema 2.11

Să se separe codul sursă aferent clasei `CERC` în fişiere separate (cerc.cpp şi cerc.h) care să fie parte a unui proiect.

Consideraţii teoretice 2.11

Prin directiva `#include` compilatorul ştie că trebuie să citească nu numai fişierul sursă (care conţine directiva) ci şi fişierul care este specificat în directivă. Prin această directivă se introduc pentru citire şi compilare fişiere antet pentru sistemul de fişiere al funcţiilor de bibliotecă.

Utilizatorii pot avea sursele scrise în mai multe fişiere, care vor trebui organizate în proiecte pentru ca compilatorul să ştie de unde poate citi funcţiile. Proiectele sunt colecţii de fişiere sursă (.cpp). Principala condiţie într-un proiecte este aceea că în fişierele sursă nu are voie să existe decât o singură funcţie main.

Indicaţii 2.11

- ⇨
- ⇨ Fiecare fişier cpp este compilat separat, link-editarea este comună.

Anexa 2

ex2.c

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

#include "ex2.h"

// variabile Globale

CERC c[16];    // declararea unui vector de 16 cercuri

void main()
//*********
{
// variabile locale

int CercCurent=0,gata=0,k;

// cod

   OurInitGraph(); // initializarea modulului grafic

// initializarea valorilor pentru cele 16 cercuri si afisarea lor pe ecran

   for(k=0;k<16;k++)
      {
       c[k].x = 30*k+100;
       c[k].y =      200;
       c[k].r =  5*k+ 25;
       c[k].c =    k    ;
       Afiseaza(&c[k]);
      }

   while(!gata)
     switch(getch())
       {
        case ESC:
          gata=1;
          break;
        case TAB:
          CercCurent++;
          CercCurent%=16;
          break;
        case UNU:    Creste( &c[CercCurent], +10     ); break;
        case DOI:    Creste( &c[CercCurent], -10     ); break;

        case 0:
          switch(getch())
            {
             case LEFT:  Muta(   &c[CercCurent], -10,  0 ); break;
             case RIGHT: Muta(   &c[CercCurent],  10,  0 ); break;
             case UP:    Muta(   &c[CercCurent],   0,-10 ); break;
             case DOWN:  Muta(   &c[CercCurent],   0, 10 ); break;
             default:    break;
            }
```

```
      }
   closegraph();
}

void Sterge(CERC *c)
//******************
// functie care sterge un cerc (prin scrierea unui cerc negru peste cel vechi)
{
   setcolor(BLACK);
   circle(c->x,c->y,c->r);     // stergere
}

void Afiseaza(CERC *c)
//********************
// functie care afiseaza un cerc
{
   setcolor(c->c);
   circle(c->x,c->y,c->r);     //afisare
}

void Muta(CERC *c,int dx,int dy)
//******************************
// functie care muta un cerc (sterge cercul de pe vechea pozitie, modifica
// coordonatele centrului si apoi afiseaza cercul pe noua pozitie)
{
   Sterge(c);           //stergere cerc

   c->x += dx;          //mutare coordonate centru
   c->y += dy;

   Afiseaza(c);         //afisare in noua pozitie
}

void Creste(CERC *c,int dr)
//*************************
// functie care modifica dimensiunea unui cerc (sterge cercul cu dimensiunea
// veche, modifica raza cercului si apoi afiseaza cercul cu noua dimensiune)
{
   Sterge(c);           //stergere

   c->r += dr;          //redimensionare

   Afiseaza(c);         //afisare cu noua dimensiune
}

void OurInitGraph()
//*****************
{
int gdriver = DETECT, gmode, errorcode;

   initgraph(&gdriver,&gmode,""); /* initialize graphics and local variables */

   errorcode = graphresult();     /* read result of initialization */
   if (errorcode != grOk)         /* an error occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);                    /* terminate with an error code */
   }
}
```

ex2.h

```
// definirea ca si constante a codurilor ASCII pentru tastele folosite

#define TAB    9
#define ESC   27

#define LEFT  75
#define RIGHT 77
#define UP    72
#define DOWN  80

#define UNU   '1'  // tasta 1 folosita pentru cresterea cercului
#define DOI   '2'  // tasta 2 folosita pentru micsorarea cercului

// definirea structurii CERC

typedef struct
  {
    int x;
    int y;
    int r;
    int c;
  }CERC;

// prototipuri de functii folosite

void OurInitGraph(void);
void Muta(CERC *c,int dx,int dy);
void Sterge(CERC *c);
void Afiseaza(CERC *c);
void Creste(CERC *c,int dr);
```