Laborator 5 – Polimorfismul

Tema 5.1

Analizaţi programul din fişierele `Lab5.H, Lab5.CPP, CLASE.H, CLASE.CPP` din anexa 5.

Tema 5.2

Să se modifice numele clasei `CERC` în `FIGURA` în idea ca în această clasă să rămână mai târziu doar elemente comune diferitelor figuri pe care la va gestiona programul.

Tema 5.3

Să se definească o noua clasă `CERC` (derivata din clasa `FIGURA`) în care vom muta elementele specifice cercului. Nu va avea date specifice suplimentare (doar cele moştenite din `FIGURA`) dar va avea metode proprii şi constructori / destructori proprii (identici ca prototip cu cei ai clasei `FIGURA`). Obiectele definite în program vor fi de tipul `CERC` iar tabloul `pc[]` va rămâne însă de tipul `FIGURA`.

Indicaţii 5.3

- Metodele `Muta` şi `Creste` rămân în clasa `FIGURA` (fiind generale, vor fi la fel pentru orice figură posibilă).
- Metodele `Afiseaza` şi `Sterge` vor fi vide în clasa `FIGURA` şi vor fi rescrise pentru clasa `CERC` (aşa cum erau ele iniţial scrise în clasa figura) fiind o implementare specifică unui cerc de fapt.
- La rulare nu se va vedea nimic pe ecran. Explicaţi (undeva avem de-a face cu o legare statică).
- Constructorii clasei `CERC` doar pasează parametrii prin apel explicit spre constructorii clasei `FIGURA` fără a face nimic suplimentar.

Tema 5.4

Să se definească metodele `Afiseaza` şi `Sterge` virtuale în clasa `FIGURA`.

Consideraţii teoretice 5.4

Să ne reamintim cum se calculează dimensiunea unei clase:

```
Dimensiune clasa = Σ dimensiune variabile membru.
```

Considerăm implementarea metodei Muta din cadrul clasei FIGURA:

```
void FIGURA::Muta(int dx,int dy)
//*****************************
{
   Sterge();              //stergere
   POZITIE::Muta(dx,dy);  //mutare
   Afiseaza();            //afisare in noua pozitie
}
```

În cadrul acestei metode se apelează metoda `Sterge` din cadrul clasei `FIGURA` (legare statică), urmează apelul metodei `Muta` din cadrul clasei `POZITIE` (apel explicit al unei metode din clasa de bază, pentru a nu produce recursivitate) urmate de apelul metodei `Afiseaza` din cadrul clasei `FIGURA`. În cazul tuturor celor trei apeluri compilatorul şi editorul de legături fac corespondenţa

între numele metodei şi adresa acesteia din segmentul de cod – legare statică. Odată fixată această legătură (în momentul compilării – link-editării) aceasta este definitivă în tot timpul rulării programului. La nivel instrucţiune de asamblare această legare corespunde apelului direct al unei proceduri (de exemplu o instrucţiune `CALL` direct). Acesta reprezintă modul clasic de legare folosit de toate limbajele structurate şi de limbajele obiectuale pentru metodele ne-virtuale (metodele definite în program până la acest moment).

În acest moment faptul că metoda `Muta` apelează metoda `Sterge` şi `Afiseaza` din clasa `FIGURA` (care oricum nu fac nimic) nu ne mulţumeşte. Dorim să apelăm metodele `Sterge` şi `Afiseaza` din clasa `CERC`. Pentru aceasta de exemplu ar trebui sa mutam – în contextul actual - şi metoda `Muta` în clasa `CERC` ceea ce nu ne convine. Am dori ca metoda `Muta` să reuşească să apeleze metode din alte clase care vor fi dezvoltate ulterior. Astfel am dori ca clasa `FIGURA` (care conţine metoda `Muta`) să devină „polimorfă" în funcţie de metodele din alte clase pe care va trebui să le apeleze.

Polimorfismul este al treilea principiu al programării orientate pe obiecte şi reprezintă proprietatea unei aplicaţii de a se comporta diferit în funcţie de condiţiile din momentul rulării. O definire a polimorfismului ar putea fi: „transmiterea de mesaje (în cazul nostru apelul de metode) către obiecte necunoscute (în cazul nostru clase care vor fi ulterior definite) dar care recunosc mesajul (în cazul nostru clasele respective au implementate metodele respective).

Pentru a implementa mecanismul de polimorfism limbajul pune la dispoziţie o posibilitate de a înlocui legarea statică (stabilirea adresei metodei în momentul link-editării) prezentată mai sus, cu o legare dinamică în care adresa metodei care se apelează se stabileşte în momentul în care se apelează acea metodă. Pentru aceasta limbajul pune la dispoziţie aşa numitele metode virtuale care permit legarea dinamică prin adăugarea unui nivel suplimentar de indirectare. O metodă virtuală se defineşte prin prefixarea definirii metodei cu cuvântul cheie „virtual" astfel:

```
specific_clasa nume_clasa_derivata [:modificator_acces nume_clasa_baza]
      {
      .........
      virtual tip_intors nume_metoda([lista_parametri]);
      .........
      };
```

Consecinţa definirii unei metode virtuale este că dacă într-o clasă derivată oferim un înlocuitor pentru această metodă compilatorul o va folosi pe aceasta într-un apel dintr-o metodă din clasa de baza.

În momentul în care într-o clasă se defineşte cel puţin o metodă virtuală dimensiunea clasei creşte cu dimensiunea unui pointer astfel:

```
Dimensiune clasa = Σ dimensiune variabile membru + dimensiune pointer
```

În acel pointer compilatorul va memora adresa unei tabele care conţine adresele tuturor metodelor definite virtual în clasa respectiva (chiar şi cele moştenite). Această tabelă poartă numele de tabelă VMT (Virtual Method Table). Fiecare clasă (nu obiect al clasei respective) va avea o tabelă VMT proprie în care se trec prima dată metodele declarate virtual în clasa de bază în aceeaşi ordine pe urmă sunt trecute metodele noi definite virtual în clasa derivată. Trebuie să atragem atenţia că mecanismul de polimorfism nu poate exista fără moştenire.

În momentul în care se defineşte o metodă virtuală compilatorul o scrie în VMT scriind acolo adresa unde se găseşte în memorie acea metodă. În timpul rulării programului când se apelează o metodă virtuală adresa acelei metode se preia din tabela VMT prin adăugarea unui nivel suplimentar de indirectare, astfel se preia adresa tabelei VMT din obiect (aceasta este memorată în

obiect) şi pe urmă de la acea adresă se preia adresa unde se găseşte în memorie metoda care se doreşte apelată.

Să considerăm următorul exemplu:

```
#include <stdio.h>
class VIETUITOARE
{
 public:
  virtual char* nume()      {return NULL;};
  virtual char* hranire()   {return NULL;};
          void  definitie() {printf("\n%s se hraneste cu %s",nume(),hranire());}
};
class VRABIA : public VIETUITOARE
{
 public:
          char* hranire()   {return „insecte”;};
          char* nume()      {return „VRABIA”;};
};
class CAL : public VIETUITOARE
{
 public:
          char* hranire()   {return „iarba”;};
          char* nume()      {return „CALUL”;};
};

void main()
{
  VIETUITOARE *vietuitoare1, *vietuitoare2;
  vietuitoare1 = new VRABIA();
  vietuitoare2 = new CAL();
  vietuitoare1->definitie();
  vietuitoare2->definitie();
}
```

În acest exemplu funcţiile `hranire` şi `nume` trebuiesc definită în clasa `VIETUITOARE` deoarece avem nevoie de ele pentru a defini o vieţuitoare, chiar dacă nu ştim cum se hrăneşte o vieţuitoare şi nici la ce vieţuitoare ne referim. Prin definirea acestora virtuale avem posibilitatea ca să le redefinim în dezvoltările ulterioare urmând ca în momentul în care e nevoie să se decidă care variantă a acestor metode să se apeleze. Programul va afişa:

```
VRABIA se hraneste cu insecte
CALUL se hraneste cu iarba
```

Indicaţii 5.4

- Aplicaţi ideile din exemplul anterior pentru `FIGURA` şi `CERC`.
- Este esenţial existenta caracterului virtual al celor 2 metode în clasa `FIGURA`. Atribuirea caracterului virtual doar în clasa `CERC` nu este satisfăcătoare. Verificaţi acest lucru.

Tema 5.5

Să modifice unul din obiectele de tip `CERC` din program astfel încât sa fie de tip `FIGURA`. Observaţi ce se întâmplă la rulare.

Indicaţii 5.5

- Atenţie la numărul de cercuri de pe ecran

Tema 5.6

Să se modifice funcţiile `Afiseaza` şi `Sterge` din clasa `FIGURA` astfel încât să devină pure.

Consideraţii teoretice 5.6

Se pune uneori problema de a defini clase care au un caracter general, fără a ne referi la o particularitate anume (de exemplu clasa `VIETUITOARE`). De aceea multe funcţii din acea clasă nu ştim cum să le implementăm, acestea definindu-se doar pentru a putea fi apelate de alte metode din clasa respectivă, dar evitându-se apelul lor în timpul rulării. În exemplul de mai sus, se evită apelul metodelor `hranire` şi `nume` din clasa `VIETUITOARE` deoarece acestea ar genera o eroare în momentul execuţiei - s-ar afişa mesajul `(null) se hraneste cu (null)` ceea ce nu ar semnifica nimic.

Pentru a putea obliga compilatorul să ne atenţioneze ori de câte ori instanţiem un obiect de tipul unei astfel de clase putem specifica în acele clase funcţiile care nu ştim cum să le implementăm ca având corpul = 0, acestea devenind funcţii `pure`.

O metodă virtuală pură se declară astfel:

```
specific_clasa nume_clasa_derivata [:modificator_acces nume_clasa_baza]
      {
      .........
      virtual tip_intors nume_metoda([lista_parametri])=0;
      .........
      };
```

O metodă pura nu se implementează pentru acea clasă. Rolul ei este doar acela de a fi înlocuită într-o clasă derivată. Alte metode ale clasei însă o pot folosi ca şi cum ar exista deja implementată.

În momentul în care într-o clasă se defineşte cel puţin o funcţie pură clasa respectivă devine clasă abstractă iar compilatorul nu va mai permite instanţierea unei clase `abstracte`.

Rolul unei clase abstracte este de a fi moştenita într-o clasa derivata care să înlocuiască toate metodele pure.

Indicaţii 5.6

- ⇨ Atenţie la ce se întâmplă când instanţiem clase abstracte.
- ⇨ În final renunţaţi la instanţierea de obiecte de clasa `FIGURA`. Atenţie, pointerii nu sunt obiecte.

Tema 5.7

Să se definească o clasa `PATRAT` asemănător cu clasa `CERC` existentă, fără a introduce membrii suplimentari. O parte din obiectele gestionate de program se vor defini ca şi pătrate. Latura pătratului va fi egală cu dublul membrului `r` din clasa `FIGURA` iar membrii `x` şi `y` vor reprezenta coordonatele centrului pătratului.

Consideraţii teoretice 5.7

Pentru desenarea unui pătrat biblioteca grafică a mediului pune la dispoziţie următoarea funcţie:

```
void rectangle( int left,  int top,  int right,  int bootom);
```

Această funcţie desenează un dreptunghi în stilul curent de linie, cu forma curentă şi cu culoarea curentă. Parametrii <left, top> specifică colţul din stânga sus iar parametrii <right, bottom> specifică colţul din dreapta jos.

Indicaţii 5.7

- ⇨ Se poate implementa clasa PATRAT prin copierea clasei CERC (şi a implementărilor aferente) modificându-se numelui clasei şi conţinutul metodelor Afiseaza şi Sterge.

Tema 5.8

Să se execute pas cu pas partea de program responsabila cu redesenarea tuturor cercurilor.

Consideraţii teoretice 5.8

Una dintre cele mai importante aplicaţii ale polimorfismului o reprezintă tratarea uniformă a masivelor eterogene în sensul că obiecte de tipuri diferite sunt tratate la fel. Aceasta este posibil deoarece compilatorul permite ca într-un pointer de tipul unei clase bază să se poată reţine adresa unui pointer o clasă derivată din clasa de bază. În acest context polimorfismul permite o tratare uniformă a unor asemenea masive prin legarea dinamică pe care o presupune.

De exemplu pentru clasele prezentate mai sus programul principal ar putea fi de forma:

```
void main()
{
  VIETUITOARE *vietuitoare[2];
  vietuitoare[0] = new VRABIA();
  vietuitoare[1] = new CAL();
  for(int i = ; i < 2; i++)
     vietuitoare[0]->definitie();
}
```

În prezenţa polimorfismului se apelează metoda definitie corespunzătoare instanţei efective (VRABIE sau CAL). În acest fel s-a reuşit o tratare uniformă a masivului (din punct de vedere al sursei programului). La baza soluţiei stă legarea dinamică a metodelor virtuale (deci polimorfismul).

Tratarea uniformă se referă la apelul unor metode existente în clasa de bază (fie ele şi pure) eventual înlocuite în clasele derivate.

Indicaţii 5.8

- ⇨ Observaţi apelul metodei Afiseaza corespunzător tipului obiectului (care nu apare explicit la apel).

Tema 5.9

Să se modifice aplicaţia astfel încât, la apăsarea tastei INSERT, obiectul inserat sa fie in mod consecutiv CERC sau PATRAT.

Consideraţii teoretice 5.9

Tipul obiectului se stabileşte în momentul în care acesta se instanţiază. Chiar dacă tipul pointerului este al clasei de bază în momentul în care se instanţiază obiectul acesta poate fi de orice tip al claselor derivate.

Indicaţii 5.9

- ⇨ Pentru decizia `CERC` / `PATRAT` se poate tine cont de variabila `NrFiguri`.

Tema 5.10

Să se definească o clasa care reprezintă o figura la alegere (maşină, faţă, casă, etc) şi să se folosească câteva instanţe de acel tip.

Consideraţii teoretice 5.10

Se pot folosi şi alte funcţii din biblioteca grafica (de exemplu `setlinestyle`).

Indicaţii 5.10

- ⇨ Asemănător cu introducerea clasei `PATRAT` în tema 5.7.

Anexa 5

LAB5.h

```
#define TAB    9                          //definire constante cu coduri taste
#define ESC   27

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

#define UNU    '1'
#define DOI    '2'
#define TREI   '3'
#define PATRU  '4'


// prototipuri de functii
void Afiseaza(void);
void OurInitGraph(void);
```

LAB5.CPP

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

#include "lab5.h"
#include "clase.h"

#define MAX_FIGURI 20

//variabile Globale

CERC *pc[MAX_FIGURI];                  // tablou de pointeri la cercuri
int NrFiguri;

CERC tf[2];                            // tablou de cercuri
CERC fg1(200,200,25),fg2(100,100,25,BLUE,"blue");   //cercuri definite global

void main()
//*********
{
int FiguraCurenta=0,gata=0,k;         // variabile locale
CERC fl1(500,300,75,RED,"red"),fl2(300,300);   //cercuri definite local

  OurInitGraph();

   pc[0]=&tf[0];                      //initializare tablou pointeri la cerc
   pc[1]=&tf[1];
   pc[2]=&fg1;
   pc[3]=&fg2;
   pc[4]=&fl1;
   pc[5]=&fl2;
   pc[6]=new CERC(400,200,100,YELLOW,"yellow");   // cercuri "dinamice"
   pc[7]=new CERC;
   NrFiguri=8;
```

```
  for(k=0;k<NrFiguri;k++)            //afiseaza toate cercurile pe ecran
     pc[k]->Afiseaza();

  while(!gata)
    switch(getch())
    {
      case ESC:
        gata=1;
        break;
      case TAB:
        FiguraCurenta++;
        FiguraCurenta%=NrFiguri;
        break;
      case UNU:    pc[FiguraCurenta]->Creste(  +10  ); Afiseaza(); break;
      case DOI:    pc[FiguraCurenta]->Creste(  -10  ); Afiseaza(); break;
      case TREI:   if(NrFiguri==MAX_FIGURI) break;  //prea multe figuri
                         pc[NrFiguri++]=new CERC;      //o figura noua
                   FiguraCurenta=NrFiguri-1;         //devine cea curenta
                     Afiseaza();
                   break;
      case PATRU:   if(NrFiguri==6) break;           //primele 6 nu se distrug
                   delete pc[--NrFiguri];            //distrugem ultima figura
                   if(NrFiguri==FiguraCurenta)
                       FiguraCurenta=0;              //daca era cea curenta...
                   Afiseaza();
                   break;

      case 0:
        switch(getch())
        {
          case LEFT:  pc[FiguraCurenta]->Muta( -10,  0 ); break;
          case RIGHT: pc[FiguraCurenta]->Muta(  10,  0 ); break;
          case UP:    pc[FiguraCurenta]->Muta(   0,-10 ); break;
          case DOWN:  pc[FiguraCurenta]->Muta(   0, 10 ); break;
        }
       Afiseaza();
  }
  closegraph();
}

void Afiseaza()
//*****************
{
  for(int k=0;k<NrFiguri;k++)                      //afiseaza toate figurile
     pc[k]->Afiseaza();
}

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 */
   }
```

```
   settextjustify(CENTER_TEXT,CENTER_TEXT);
}
```

CLASE.H

```
#include <graphics.h>

class POZITIE                                   //clasa pozitie
{
  public:                                       //membrii publici
         POZITIE();                             //constructor fara parametrii
         POZITIE(int x0,int y0);                //constructor cu parametrii
    int  GetX   ()             { return(x); }; //metoda citire membru x
    int  GetY   ()             { return(y); }; //metoda citire membru y
    void Muta   (int dx,int dy);
  protected:                                    //membrii protected
    int  x;
    int  y;
};

class CERC : public POZITIE
{
    int  r;                                     //membrii implicit privati
    int  c;
    char *Nume;
  public:                                       //metode PUBLICE
         CERC();                                //constructor fara parametrii
         CERC(int x0,int y0,int r0=20,int c0=RED,char *n0="IMPLICIT");
         ~CERC();                               //destructor
    void Muta(int dx,int dy);                   //metode publice
    void Afiseaza();
    void Creste(int dr);
  private:
    void Sterge();                              //metoda PRIVATA
};
```

CLASE.CPP

```
#include <graphics.h>
#include <string.h>
#include <conio.h>
#include <alloc.h>

#include "clase.h"

//~~~~~~~~~~~~~~~~~~
//  clasa POZITIE
//~~~~~~~~~~~~~~~~~~

POZITIE::POZITIE()                    //constructor fara parametrii
//****************
{
    x =   320;
    y =   240;
}

POZITIE::POZITIE(int x0,int y0)       //constructor cu param
//*****************************
{
    x = x0;
```

```
    y = y0;
}

void POZITIE::Muta(int dx,int dy)
//*******************************
{
    x += dx;
    y += dy;
}

//~~~~~~~~~~~~~~~
//  clasa CERC
//~~~~~~~~~~~~~~~

CERC::CERC()                        //constructor fara parametrii
//**********
{
    r =    50;
    c = WHITE;
    Nume = (char*)malloc(strlen("CERC")+1); //aloca memoria necesara sir
    strcpy(Nume,"CERC");                    //copiaza un sir in Nume
}

CERC::CERC(int x0,int y0,int r0,int c0, char *n0):POZITIE(x0,y0)
//*************************************************************
{
      r = r0;
      c = c0;
      Nume = (char*)malloc(strlen(n0)+1);    //aloca memoria necesara
      strcpy(Nume,n0);
}

CERC::~CERC()                       //destructor
//***************
{
    Sterge();                           //sterge de pe ecran
    free(Nume);                         //elibereaza memoria alocata
}

void CERC::Sterge()                 //metoda sterge o figura de pe ecran
//*****************
{
    setcolor(BLACK);
    circle(x,y,r);                      //stergere prin desenare cu negru
    outtextxy(x,y,Nume);                //sterge text din mijlocul figurii
}

void CERC::Afiseaza()               //metoda afisare figura
//********************
{
    setcolor(c);
    circle(x,y,r);                      //desenare figura pe ecran
    outtextxy(x,y,Nume);                //scrie text in mijlocul figurii
}

void CERC::Muta(int dx,int dy)
//****************************
{
    Sterge();                           //stergere figura
    POZITIE::Muta(dx,dy);               //mutare
    Afiseaza();                         //afisare in noua pozitie
}
```

```
void CERC::Creste(int dr)
//************************
{
    Sterge();                           //stergere
    r += dr;                            //redimensionare
    Afiseaza();                         //afisare cu noua dimensiune
}
```