POO – C++ - Laborator 9

Cuprins

1. Pointeri la clasa de bază

O proprietate importantă a moştenirii este faptul că pointerii către o clasă de bază pot primi adresa unui obiect de tip clasă derivată.

Să ilustrăm această proprietate într-un exemplu. Avem clasa de bază `Carte` cu câmpul `titlu` şi funcţia `afisare()`. Din ea este derivată clasa `Culegere`, care este o carte ce conţine mai multe lucrări. Culegerea are 2 câmpuri în plus – `nrLucrari` şi `lucrari` – numărul şi titlurile lucrărilor. Şi mai are o altă funcţie `afisare()` – care afişează toate datele din culegere.

Carti.h

```
#ifndef _Carti_
#define _Carti_
#pragma warning(disable : 4996)

class Carte {
protected:
	char *titlu;
public:
	Carte(char *titlu);
	~Carte();
	void afisare();
};

class Culegere : public Carte {
protected:
	int nrLucrari;
	char **lucrari;
public:
	Culegere(char *titlu, int nrLucrari, char
**lucrari);
	~Culegere();
	void afisare();
};
```

```
#endif
```

Carti.cpp

```
#include<iostream>
#include<string.h>
#include"Carti.h"
using namespace std;

Carte::Carte(char *titlu){
	this->titlu = new char[strlen(titlu)+1];
	strcpy(this->titlu, titlu);
}

Carte::~Carte() {
	delete[] titlu;
}

void Carte::afisare() {
	cout << titlu << endl;
}

Culegere::Culegere(char *titlu, int nrLucrari, char
**lucrari): Carte(titlu) {
	this->nrLucrari = nrLucrari;
	this->lucrari = new char*[nrLucrari];
	for(int i=0; i<nrLucrari; i++) {
		char *lucrare =
		    new char[strlen(lucrari[i])+1];
		strcpy(lucrare, lucrari[i]);
		this->lucrari[i] = lucrare;
	}
}

Culegere::~Culegere() {
	for(int i=0; i<nrLucrari; i++) {
		delete[] lucrari[i];
	}
	delete[] lucrari;
}

void Culegere::afisare() {
	cout << titlu << ". Lucrari:" <<endl;
	for(int i=0; i<nrLucrari; i++) {
		cout << "    " << lucrari[i] << endl;
	}
}
```

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Carti.h"
```

```
using namespace std;

int main() {
      Carte *carte = new Carte("Moara cu Noroc");
     char *poeme[] = {"Luceafarul", "Memento Mori"};
      Culegere *culeg = new Culegere("Poeme Eminescu", 2,
poeme);
      Carte *cculeg = culeg;

      cout << "carte->afisare(): ";
      carte->afisare();
      cout << "cculeg->afisare(): ";
      cculeg->afisare();
      cout << "culeg->afisare(): ";
      culeg->afisare();

      delete carte;
      delete culeg;
      _getch();
      return 0;
}
```

Ieşire:

```
carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu
culeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
```

Observăm atribuirea din funcţia `main()`:

```
Carte *cculeg = culeg;
```

Atribuirea unui pointer de tip clasă derivată `Culegere*` către un pointer de tip clasă de bază `Carte*` este perfect validă.

Prin intermediul pointerului la clasa de bază `Carte` putem accesa orice membru din `Carte`, indiferent dacă obiectul referit este de tip `Carte` sau `Culegere`. De exemplu putem să accesăm funcţia `afisare()`.

Şi într-adevar, observăm că în rezultatul apelului

```
cculeg->afisare();
```

este executată funcţia `Carte::afisare()`. Ca să putem accesa funcţia `Culegere::afisare()`, în acest exemplu, suntem nevoiţi să utilizăm pointerul la clasa derivată:

```
culeg->afisare();
```

Funcţiile `Carte::afisare()` şi `Culegere::afisare()` nu au nici o legătură între ele în acest program.

2. Metode virtuale. Polimorfismul.

Metodă (funcţie) virtuală este o metodă definită în clasa de bază, şi care poate fi redefinită în clasele derivate. Se declară cu ajutorul cuvântului cheie `virtual`. Atunci cînd apelăm o metodă virtuală prin intermediul unui pointer la obiect, se va apela întotdeauna metoda din tipul obiectului, indiferent de tipul pointerului.

Se spune că metodele virtuale pot fi ***redefinite*** (overloaded) în clasele derivate. (A se face diferenţa dintre *redefinire* şi *supraîncărcare*, două facilităţi diferite.)

Să modificăm exemplul de mai sus, şi să declarăm funcţia `afisare()` din clasa `Carte` cu cuvântul cheie `virtual` în faţă:

Carti.h

```
…
class Carte {
protected:
      char *titlu;
public:
      Carte(char *titlu);
      ~Carte();
      virtual void afisare();
};
…
```

Restul programului rămâne neschimbat. Executăm programul. Rezultatul devine:

```
carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
culeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
```

De data aceasta funcţiile `Carte::afisare()` şi `Culegere::afisare()` sunt ambele virtuale, şi sunt legate. A 2-a funcţie o redefineşte pe prima.

Întrucât pointerii `culeg` şi `cculeg` referă acelaşi obiect, atât pentru expresia

```
cculeg->afisare();
```

cât şi pentru

```
culeg->afisare();
```

se va apela funcţia `Culegere::afisare()` – funcţia din tipul obiectului. Chiar dacă cei 2 pointeri sunt de tipuri diferite.

O funcţie definită virtuală în clasa de bază, va rămâne virtuală în tot arborele ierarhic a acelei clase, indiferent câte nivele are derivarea. Funcţia redefinită trebuie să aibă aceiaşi parametri şi acelaşi tip de return ca şi funcţia originală din clasa de bază.

Datorită funcţiei `afisare()` virtuale, putem rescrie funcţia `main()` astfel încât să folosim doar pointeri de tip `Carte*`:

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Carti.h"
using namespace std;

int main() {
	char *poeme[] = {"Luceafarul", "Memento Mori"};
	Carte *carte = new Carte("Moara cu Noroc");
	Carte *cculeg = new Culegere("Poeme Eminescu", 2,
poeme);

	cout << "carte->afisare(): ";
	carte->afisare();
	cout << "cculeg->afisare(): ";
	cculeg->afisare();

	delete carte;
	delete cculeg;
	_getch();
	return 0;
}
```

Ieşire:

```
carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
```

Putem generaliza programul şi mai mult şi să utilizăm un vector de pointeri de tip `Carte*`, al cărui elemente să fie pointeri la diferite tipuri de obiecte. Să modificăm funcţia `main()` în felul următor:

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Carti.h"
using namespace std;

int main() {
	char *poeme[] = {"Luceafarul", "Memento Mori"};
	Carte *carti[3];
	carti[0] = new Carte("Moara cu Noroc");
	carti[1] = new Culegere("Poeme Eminescu", 2,
poeme);
	carti[2] = new Carte("Amintiri din copilarie");

	for(int i=0; i<3; i++) {
		cout << i << ". ";
		carti[i]->afisare();
		delete carti[i];
	}
```

```
        _getch();
        return 0;
}
```

Ieşire:

```
0. Moara cu Noroc
1. Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
2. Amintiri din copilarie
```

Observăm că singurul loc din program care ştie tipul obiectelor sunt liniile de cod care realizează instanţierea:

```
carti[0] = new Carte("Moara cu Noroc");
carti[1] = new Culegere("Poeme Eminescu", 2, poeme);
carti[2] = new Carte("Amintiri din copilarie");
```

În restul programului, o singură instrucţiune ştie să afişeze fiecare carte în mod corect, corespunzător tipului său:

```
carti[i]->afisare();
```

Polimorfismul reprezintă capacitatea de a apela metode diferite prin intermediul unei singure expresii de genul

```
pb->f()
```

, unde `pb` este pointer la o clasă de bază, iar `f()` este o funcţie virtuală redefinită în clasele derivate. Funcţia apelată este decisă în timpul execuţiei programului, în funcţie de tipul obiectului referit de `pb`.

Un astfel de apel de funcţie, pentru care nu se cunoaşte funcţia concretă apelată în momentul compilării programului, se numeste ***apel polimorfic***.

Polomorfismul reprezintă cel de-al 3-lea principiu al POO, dupa ***Încapsulare (abstractizare)*** şi ***Moştenire***.

Principiul polimorfismului ridică POO la adevărata sa valoare, îndeosebi în proiectele mari. Putem să scriem o bibliotecă de clase în care să manipulăm obiectele prin intermediul poiterilor la clasa de bază, fără să cunoaştem tipul concret al obiectelor. Iar în alte proiecte, să folosim biblioteca în combinaţie cu mai multe clase derivate.

1. Destructor virtual

Ultimul program dat ca exemplu afişează rezultatul aşteptat, dar conţine un defect. Pe linia:

```
delete carti[i];
```

operatorul `delete` va apela destructorul. Dar pentru că pointerul este de tip `Carte*`, destructorul apelat va fi doar `Carte::~Carte()`. Iar câmpul - pointer `lucrari`, din obiectele de tip `Culegere` va rămâne dezalocat. Ca să corectăm acest defect, vom declara şi destructorul clasei `Carte` virtual:

```
…
class Carte {
protected:
      char *titlu;
public:
      Carte(char *titlu);
      virtual ~Carte();
      virtual void afisare();
};
…
```

De data aceasta, la apelarea operatorului `delete` pentru pointer la clasa de bază, se va apela tot lanţul de destructori de la tipul obiectului pâna la cea mai de bază clasă, indiferent de tipul pointerului.

Pentru a avea garanţia că obiectele sunt întotdeauna dezalocate corect, există următoarea regulă:

În orice ierarhie de clase C++, clasa cea mai de bază trebuie să conţină un destructor virtual. Chiar dacă în clasa de bază nu avem ce dezaloca, vom defini un destructor virtual vid (fără instrucţiuni).

2. Clase abstracte

Există cazuri când o funcţie virtuală nu are sens să fie implementată într-o clasă de bază, dar are sens în clasele derivate. De exemplu putem avea clasa de bază `Figura` cu funcţia virtuală `arie()` (convenim că orice figură în plan are o arie), şi clasele derivate `Dreptunghi` şi `Cerc`. Cunoaştem formula ariei pentru dreptunghi şi cerc, dar nu putem defini aria pentru o figură în general.

În acest caz, vom declara funcţia din clasa de bază ***virtuală pură***.

Funcţie virtuală pură este o funcţie virtuală care nu are corp. Se declară adăugând simbolurile "`=0;`" la sfârşitul declaraţiei funcţiei.

De exemplu, declaraţia funcţiei virtuale pure `arie()` va arăta în felul următor:

```
class Figura {
public:
      virtual float arie()=0;
      …
};
```

Clasă abstractă este o clasă care conţine cel puţin o funcţie virtuală pură. Clasele abstracte au restricţia că nu pot fi instanţiate, adică nu putem crea obiecte pe baza lor. Ele sunt create special pentru a fi derivate. Însă putem să declarăm pointeri pe clase abstracte, care vor referi obiecte de tip clase derivate concrete.

Clasele derivate în schimb trebuie să implementeze funcţiile virtuale pure moştenite, pentru a deveni clase concrete.

Codul de mai jos conţine declaraţia şi implementarea claselor `Figura`, `Dreptunghi` şi `Cerc`. Clasele derivate implementează metodele virtuale pure moştenite de la clasa de bază.

figuri.h

```
#ifndef _figuri_
#define _figuri_
#pragma warning(disable : 4996)

class Figura {
public:
	virtual ~Figura(){}
	virtual float arie()=0;
	virtual void afisare()=0;

};

class Dreptunghi : public Figura {
private:
	int x1,y1,x2,y2;
public:
	Dreptunghi(int x1,int y1,int x2,int y2);
	float arie();
	void afisare();
};

class Cerc : public Figura {
private:
	int x,y,r;
public:
	Cerc(int x, int y, int r);
	float arie();
	void afisare();
};

#endif
```

figuri.cpp

```
#include<iostream>
#include"Figuri.h"
using namespace std;

Dreptunghi::Dreptunghi(int x1, int y1, int x2, int y2) {
	this->x1 = x1;
	this->y1 = y1;
	this->x2 = x2;
	this->y2 = y2;
}

float Dreptunghi::arie() {
	return (float)(x2 - x1)*(y2 - y1);
}

void Dreptunghi::afisare() {
	cout << "Dreptunghi cu coordonatele ("
<<x1<<","<<y1
		<<")-("<<x2<<","<<y2<<"), si aria " <<
arie()<<endl;
}
```

```
Cerc::Cerc(int x, int y, int r) {
	this->x = x;
	this->y = y;
	this->r = r;
}

float Cerc::arie() {
	const float PI = 3.14F;
	return PI * r * r;
}

void Cerc::afisare(){
	cout << "Cerc cu coordonatele ("
	    <<x<<","<<y
	      <<"), raza " << r <<" si aria "
	    << arie()<<endl;
}
```

figuriMain.cpp

```
#include<conio.h>
#include"Figuri.h"

int main() {
	Figura *dr = new Dreptunghi(1,2,4,4);
	Figura *cerc = new Cerc(1,1,3);
	//urmatoarea linie ar genera eroare:
	//Figura *fig = new Figura();
	dr->afisare();
	cerc->afisare();
	delete dr;
	delete cerc;
	_getch();
	return 0;
}
```

Dacă am încerca să creăm un obiect de tip `Figura` în funcţia `main()`, am obţine o eroare de compilare, deoarece clasa `Figura` este abstractă şi nu poate fi instanţiată.

În continuare vom folosi aceste 3 clase într-un exemplu care demonstrează mai elocvent avantajele polimorfismului.

Vom adăuga la program o funcţie globală care va determina figura cu arie maximă dintr-un vector de pointeri la figuri:

figuri.h

```
…
Figura *figCuArieMax(Figura **figuri, int n);

#endif
```

figuri.cpp

```
//...
Figura *figCuArieMax(Figura **figuri, int n) {
```

```
	float max = 0;
	Figura *figMax = NULL;
	for(int i=0; i<n; i++) {
		float arie = figuri[i]->arie();
		if (arie > max) {
			max = arie;
			figMax = figuri[i];
		}
	}
	return figMax;
}
```

În funcţia `main()` sunt instanţiate câteva figuri. Apoi este determinată şi afişată figura cu arie maximă.

figuriMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Figuri.h"
using namespace std;

int main() {
	const int n = 3;
	Figura *figuri[n];
	figuri[0] = new Dreptunghi(0,0,2,5);
	figuri[1] = new Dreptunghi(0,0,2,2);
	figuri[2] = new Cerc(0,0,3);
	Figura *figMax = figCuArieMax(figuri, n);

	cout << "    Dintre figurile:"<<endl;
	for(int i=0; i<3; i++) {
	    cout << i << ". ";
		figuri[i]->afisare();
	}
	cout << endl
	    <<"    aria maxima o are:"<<endl;
	figMax->afisare();
	for(int i=0; i<n; i++) {
		delete figuri[i];
	}
	_getch();
	return 0;
}
```

Ieşire

```
    Dintre figurile:
0. Dreptunghi cu coordonatele (0,0)-(2,5), si aria 10
1. Dreptunghi cu coordonatele (0,0)-(2,2), si aria 4
2. Cerc cu coordonatele (0,0), raza 3 si aria 28.26

    aria maxima o are:
Cerc cu coordonatele (0,0), raza 3 si aria 28.26
```

Se observă că funcţia `figCuArieMax()` poate să determine figura cu arie maximă chiar şi dintr-o listă care conţine atât dreptunghiuri cât şi cercuri. Acest lucru a fost posibil datorită apelului polimorfic a funcţiei `arie()` în funcţia `figCuArieMax()`:

```
float arie = figuri[i]->arie();
```

Putem chiar să extindem programul şi să adăugam noi tipuri de figuri, iar funcţia `figCuArieMax()` va şti automat să determine aria maximă şi pentru ele.

3. Exerciţii

Continuaţi dezvoltarea ultimului exemplu din acest laborator. Clasa `Figura` şi derivatele sale. Efectuaţi următoarele înbunătăţiri:

1. Adăugaţi clasa `Triunghi` derivată din `Figura`.

   Triunghiul va fi definit de cele 3 vârfuri, fiecare avand coordonatele x şi y. (Câţi parametri va avea constructorul clasei `Triunghi`?)

   Aria triunghiului se poate calcula după formula lui Heron: s = sqrt(p*(p-a)*(p-b)*(p-c)), unde p este semiperimetrul: p = (a + b + c) / 2.

2. Adăugaţi la clasa `Figura` funcţia virtuală pură `perimetru()`.

   Afişaţi figura cu perimetrul maxim.

3. Scrieţi o funcţie care sortează un vector de pointeri la figuri crescător după arie. Afişaţi figurile sortate.