Lab 10 – P2 – Polimorfism (2)

Cuprins

1. Clase abstracte

Există cazuri când o funcție 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 funcția virtuală `arie()` (convenim că orice figură în plan are o arie), și clasele derivate `Dreptunghi` și `Cerc`. Cunoaștem formula ariei pentru dreptunghi și cerc, dar nu putem defini aria pentru o figură în general.

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

O funcție virtuală pură este o funcție virtuală care de obicei nu are definiție. Se declară adăugând simbolurile "`=0;`" la sfârșitul declarației funcției.

De exemplu, declarația funcției virtuale pure `arie()` va arăta în felul următor:

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

O clasă abstractă este o clasă ce conține cel puțin o funcție virtuală pură. Clasele abstracte au restricția de a nu putea fi instanțiate, adică nu putem crea obiecte de tipul lor. Aceste clase sunt create special pentru a fi derivate. Putem însă să declarăm pointeri pe clase abstracte, care vor referi obiecte de tip clase derivate concrete.

Clasele derivate trebuie să implementeze funcțiile virtuale pure moștenite, pentru a deveni clase concrete. Orice clasă derivată dintr-o clasă abstractă ce nu redefinește funcțiile virtuale pure din clasa de bază devine și ea abstractă! O funcție virtuală pură ce nu a fost definită în clasa derivată rămâne o funcție virtuală pură.

Codul de mai jos conține declarația și implementarea claselor `Figura`, `Dreptunghi` și `Cerc`. Clasele derivate implementează metodele virtuale pure moștenite de la clasa de bază.

figuri.h

```
#pragma once

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();
};
```

figuri.cpp

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

Dreptunghi::Dreptunghi(int a1, int b1, int a2, int b2) {
	x1 = a1;
	y1 = b1;
	x2 = a2;
	y2 = b2;
}

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 a, int b, int raza) {
	x = a;
	y = b;
	r = raza;
}

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"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;
      return 0;
}
```

Pentru a evita transformarea clasei Dreptunghi într-o clasă abstractă, trebuie să redefinim toate funcțiile virtuale pure din clasa de bază. Dacă am încerca să creăm un obiect de tip `Figura` în funcția `main()`, am obține o eroare de compilare, deoarece clasa `Figura` este abstractă și nu poate fi instanțiată.

În continuare vom folosi aceste 3 clase într-un exemplu care demonstrează mai elocvent avantajele polimorfismului. Vom adăuga la program o funcție globală care va determina figura cu arie maximă dintr-un vector de pointeri la figuri:

figuri.h

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

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 funcția `main()` sunt instanțiate câteva figuri. Apoi este determinată și afișată figura cu arie maximă.

figuriMain.cpp

```
#include<iostream>
#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];
	}
	return 0;
}
```

Ieșire

```
    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 funcția `figCuArieMax()` poate să determine figura cu arie maximă chiar și dintr-o listă care conține atât dreptunghiuri cât și cercuri. Acest lucru a fost posibil datorită apelului polimorfic a funcției `arie()` în funcția `figCuArieMax()`:

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

Putem să extindem programul și să adăugam noi tipuri de figuri, iar funcția `figCuArieMax()` va ști automat să determine aria maximă și pentru ele.

2. Type casting

Conversia unei expresii dintr-un anumit tip în alt tip este cunoscută sub numele de **Type casting**.
Modalități de conversie:

- Downcast - conversia dintr-o clasă de bază către o clasă derivată,
- Upcast – Conversia dintr-o clasă derivată către o clasă de bază,

- Crosscast – conversia de la o clasă derivată către o altă clasă derivată.

Conversia se realizează cu ajutorul operatorilor:

- `dynamic_cast <>`
- `static_cast <>`
- `reinterpret_cast <>`
- `const_cast <>`

Operatorul `dynamic_cast`

Acest operator are nevoi de doi operanzi: un tip de date încadrat între două paranteze unghiulare **<T*>** și un pointer încadrat între doua paranteze **(p)**.

dynamic_cast<T*>(p)

Dacă p este de tipul T* sau a unui tip D* derivat din T atunci rezultatul este exact ca și cum s-ar atribui p unui T*. Operatorul `dynamic_cast<T*>(p)` analizează obiectului pointat de p. Dacă acel obiect este de clasa T ori are o clasă unică T, `dynamic_cast` returnează un pointer de clasă T*, altfel `nullptr`. Dacă p este `nullptr` atunci `dynamic_cast<T*>(p)` returnează `nullptr`.

Operatorul `dynamic_cast` necesită un pointer sau o referință către un tip polimorfic pentru downcast ori crosscast.

```
class B1
{
	public:
	virtual void f(void){};
}
```

```
class DB1:public B1
{
	public:
	void f(void){};
}
```

```
void g(B1* pb1)
{
	DB1* pdb1 = dynamic_cast<DB1*>(pb1); //OK
}
```

3. Exerciții

Continuați dezvoltarea ultimului exemplu din acest laborator. Clasa `Figura` și derivatele sale. Efectuați următoarele îmbunătățiri:

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

   Triunghiul va fi definit de cele 3 vârfuri, fiecare având coordonatele x și y. (Câți 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 semi-perimetrul: p = (a + b + c) / 2.

   Adăugați în clasa `Triunghi` funcția `semiperimetru()`.
2. Adăugați la clasa `Figura` funcția virtuală pură `perimetru()`. Afișați figura cu perimetrul maxim.

3. Scrieți o funcție care sortează un vector de pointeri la figuri crescător după arie. Afișați figurile sortate.
4. Utilizați operatorul `dynamic_cast` pentru a apela funcția `semiperimetru()` (declarată în clasa `Triunghi`) pentru fiecare pointer din vectorul de pointeri la figuri. În ce condiții este posibil acest apel?