PROGRAMARE OBIECT-ORIENTATA

LABORATOR 4

- **MOSTENIRE SIMPLA DE TIP PUBLIC-PUBLIC SI PUBLIC-PRIVATE**
- **INTEGRAREA ELEMENTELOR DIN LABORATOARELE 1-4**

Un prim program pe care il vom studia in acest laborator urmareste verificarea notiunilor invatate despre constructorul de copiere si destructor in laboratorul 3. Evidentiem acest exemplu prin clasa Carte.

Aceasta preia ca atribute numarul total de pagini, numarul curent de pagini (cate au fost citite pana la acel moment) si editura din care face parte acea carte. Metode Getter construim pentru toate cele 3 atribute insa de Setteri nu avem nevoie pentru ca initializam datele prin constructorul cu argumente.

Constructoul de copiere , are intotdeauna numele clasei. El primeste mereu ca argument o referita constanta la un obiect de tip carte. Prin intermediul referintei ii precizam compilatorului adresa unde se va afla in memorie acel obiect.

Destructorul va avea un mesaj prin care semnalizam faptul ca este folosit in executie.

Am mai implementat doua metode standard:

Citesc : aceasta primeste ca argument o valoare integer pentru a preciza cate pagini citesc dar inainte verifica ca nu cumva sa fi depasit numarul de pagini total. Daca s-au depasit afisam un mesaj pe ecran, daca nu s-au depasit atunci adaugam numarul de pagini citit la numarul curent de pagini(adica la cate pagini am citit pana in acel moment)

Diferenta: prin aceasta metoda calculez numarul de pagini pe care il mai am de citit pana termin cartea.

In continuare avem scris codul acestei clase Carte:

```
#include <iostream>

using namespace std;


class CARTE

{

       int nr_pag; int

       pag_crt; string

       editura;
```

```
	public:
		//nu am nevoie de setteri deoarece atributele primesc valori doar
		//prin constructorul cu parametri(in acest exemplu)
	//urmatoarele date sunt publice
	//daca nu mai exista nici un modificator de acces
	//vor fi toate publice pana la inchiderea clasei
		int GetNrPag();
		int GetPagCrt(); string
		GetEditura(); CARTE(int nr,
		string edit);
		CARTE(const CARTE&);//prototip constructor de copiere
		//fiind constructor are acelasi nume cu clasa
		//primeste ca argument o referinta constanta la un obiect
		//de tipul clasei
		~CARTE();//prototip destructor

		int CITESC(int pagini);//prototip metoda standard
		int DIFERENTA();//prototp metoda standard

};//clasa se incheie mereu cu punct si virgula

int CARTE::GetNrPag()
{
	return nr_pag;
}

int CARTE::GetPagCrt()
{
	return pag_crt;
```

```
}

string CARTE::GetEditura()
{
	return editura;
}
CARTE::CARTE(int nr, string edit)
{
	nr_pag=nr;
	//initializez numarul de pagini maxim cu valoarea
	//lui nr
	pag_crt=0;
	//initializez numarul de pagini curent cu 0
	//(cate pagini am citit pana in acest moment)
	editura=edit;
	cout<<"Am apelat constructorul cu argumente\n";
	//afisez mesaj ori de cate ori este apelat acest constructor
}

CARTE::CARTE(const CARTE& c)
{
	nr_pag=c.nr_pag;
	editura=c.editura;
	cout<<"Am apelat constructorul de copiere\n";
}

CARTE::~CARTE()
{
	cout<<"Am distrus obiectul\n";
}
```

```
int CARTE::CITESC(int pagini)
{//testez daca numarul de pagini citite deja plus
//numarul de pagini pe care il mai citesc depasteste
//numarul total de pagini
	if(pag_crt + pagini >= nr_pag)
	{//daca depaseste atunci dau doar un mesaj
		cout<<"am terminat de citit cartea"<<endl;
	}
	else{//daca nu depaseste adun numarul de pagini citite la
	//numarul de pagini curent(unde am ajuns cu citirea)
			pag_crt=pag_crt + pagini;
		}

}

int CARTE::DIFERENTA()
{//aflu cate pagini mai am de citit din carte
//scad din total pagini numarul deja citit de pagini
	int diferenta;
	diferenta=nr_pag-pag_crt;
	return diferenta;
}

int main()
{
	CARTE c1(60,"nemira");//creez un obiect de tip carte care
	//isi ia valorile din constructorul cu 2 argumente
	CARTE c2=c1;//efect 1: creez obiectul c2 tot in acelasi stil ca c1
	//efect2: copiez fiecare atribut al
	c2.CITESC(20);//aplic metoda citesc obiectului C2
```

```
	cout<<"am citit "<<c2.GetPagCrt()<<" de pagini"<<endl;

	cout<<"mai am de citit "<<c2.DIFERENTA()<<endl;//calculez cat mai am de citit
din cartea c1

	c2.CITESC(20);//aplic din nou metoda citesc tot obeictului c2

	cout<<"am citit "<<c2.GetPagCrt()<<" de pagini"<<endl;

	//afisez cate pagini am citit prin getter



}
```

Un al doilea exemplu din acest laborator il constituie un exemplu simplu de mostenire de clase .

Pentru aceasta evidentiem lucrurile importante care trebuie retinute despre mostenire in acest laborator:

Mostenirea este procedeul specific programarii orientate pe obiect prin care putem refolosi cod deja scris. Codul pe care il refolosim se afla in intr-o clasa pe care o numim CLASA DE BAZA ce are atribute si metode. Refolosirea codului presupune scrierea unei CLASE DERIVATE (sau a mai multor clase derivate) care vor avea atribute si metode proprii dar in acelasi timp se vor folosi si de atributele / metodele din clasa de baza. Exista o relatie numerica intre aceste date membre (attribute/metode):

Numarul de date membre ale clasei derivate este suma dintre numarul de date membre din clasa de baza(cele mostenite) si numarul de date membre proprii ale clasei derivate.

Relatia dintre clasa derivata si cea de baza este de tipul “este un” sau “este o” mai exact clasa derivata “Revista” daca este mostenita din clasa “Carte”, inseamna ca putem spune “Revista este o carte”.

Aceasta relatie intre clase este insotita si de un modificator de acces care poate fi: public, protected sau private si care descrie modul in care clasa derivata are acces la datele membre din clasa de baza (cum mosteneste).

Pentru testul de laborator trebuie sa stim sa construim doar clase derivate care mostenesc public , clasa de baza. Acest tip public de mostenire , ofera acces total la datele din clasa de baza cu conditia ca acestea din urma sa fie declarate tot public.

Sintaxa pe care trebuie sa o cunoastem la mostenire pentru testul de laborator este:

```
class nume_derivata : public nume_baza
{     //date membre pentru clasa derivate
      //modul de construire este a datelor membre nu se schimba fata de ce stiam
      //pana acum la clasele in general
};
```

Atentie ! Clasa derivata se inchide si aceasta cu punct si virgula.

Dupa cum se poate observa in aceasta sintaxa, dupa cuvantul rezervat "class" urmeaza numele clasei derivate dupa care se foloseste operatorul "doua puncte" – " : " urmat de modificatorul de acces pentru mostenire apoi numele clasei de baza.

Atentie ! In nici un caz in sintaxa evidentiata cu galben mai sus, nu se foloste operatorul de rezolutie "::" in loc de operatorul ":" !

Sa urmarim acum un prim exemplu legat de mostenire public-public :

```
#include <iostream>
using namespace std;
class BAZA
{//creez clasa BAZA
      int x;//membru privat
      public://urmatoarele date sunt publice
            void initX(int n)//Setter pentru x
                  {x=n;}
            void getX()//Getter pentru X
                  {cout<<x;}

};//inchid clasa mereu cu punct si virgula !
```

```
class DERIVATA:public BAZA
{//creez clasa derivata
      int y;//membru privat
      public://urmatoarele date sunt publice
      void initY(int n)//setter pentru y
                  {y=n;}
            void getY()//getter pentru y
                  {cout<<y;}
};

int main()
{//creez obiectul de tip clasei derivate.
      DERIVATA D1;
      D1.initX(10);//setez valoarea pt x: OK(fara erori)
      D1.initY(20);//setez valoarea pt y: OK(fara erori)
      D1.getX();//afisez valoarea pt x: OK(fara erori)
      D1.getY();//afisez valoarea pt y: OK(fara erori)
      D1.x=30;
      //EROARE ! x nu e accesibil pentru ca e private in clasa de baza
      //chiar daca mostenirea este publica
      D1.y=40;//EROARE ! y nu e accesibil in mod direct !}
```

Concluzii :

- metodele initX() si getX sunt accesibile pentru ca DERIVATA mosteneste public clasa BAZA iar in clasa BAZA , aceste metode sunt publice.
- Atributele x si y nu sunt accesibile fiind private, nici chiar modificatorul public de la mostenire nu permite accesul lui x din afara clasei de baza.

Un al treilea exercitiu din acest laborator isi doreste sa reuneasca elementele din laboratoarele 1 , 2 si 3 cu o mostenire simpla:

Practic , materia pentru testul de laborator va fi:

- Creare clasa de baza avand:
  - Date membre private: **atribute**
  - Date membre publice
    - **Setter**
    - **Getter**
    - **Constructor implicit**
    - **Constructor cu parametri**
    - **Constructor de copiere**
    - **Destructor**
    - **Alte metode specifice clasei**
  - Date non-membre ale clasei
    - **Functii cu argument obiect**
    - **Functii cu argument referinta la obiect**
    - **Functii Friend**
- Creare clasa derivata avand:
  - Date membre private: **atribute**
  - Date membre publice
    - **Setter**
    - **Getter**
    - **Alte metode specifice clasei**
- **Creare functie main() in care:**
  - Cream obiecte ale clasei de baza care isi iau datele :
    - Din constructorul implicit
    - Din constructorul cu parametri
    - Prin copierea dintr-un obiect existent
  - Accesam functiile membre si non-membre pentru obiectele create
  - Cream obiecte de tipul clasei derivate si accesam metodele celor doua clase pentru aceste obiecte.

De mentionat ca intr-un subiect nu se vor gasi toate aceste elemente, ci doar o selectie din ele astfel incat sa se poata rezolva intr-un timp de 50 de minute.

Pentru a evidentia toate aceste elemente , construim clasa Patrat din care vom deriva clasa Piramida.

Explicatiile sunt notate sub forma de comentarii in codul sursa:

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

using namespace std;

class PATRAT
{//atributele vor fi private
	int latura;//sunt atribute private
	//pentru ca nu dorim sa fie modificate direct
	//private la atribute
	string culoare;
	public:
		//iar metodele vor fi publice
		void SetLatura(int dim);//setterul este in general void
		//pentru ca nu returneaza nimic
		void SetCuloare(string clr);
		int GetLatura();//getterul care returneaza o variabila
		//trebuie sa aiba ca tip , exact tipul variabilei returnate
		string GetCuloare();
		//urmeaza constructorii
		//constructorii nu au tip returnat!
		//constructorii au intotdeauna numele clasei !!!
		PATRAT();//declaratia (prototipul) constructorului implicit
		PATRAT(int dim, string clr);//declaratia (prototipul) constructorului cu
parametri
		//un constructor cu parametri nu ne nevoie sa aiba ca numar de parametri
		//exact numarul de atribute
		//unele atribute pot fi atribuite prin setter
		//pot folosica parametri aceleasi variabile ca la Setteri pentru ca
		//dim si clr au vizibilitate locala (nu sunt variable globale)
		PATRAT(const PATRAT &p);//declaratia constructorului de copiere
		//primeste ca parametru o referinta constanta de tipul PATRAT
		~PATRAT();//declaratia destructorului


		friend int Perimetru(PATRAT *P);
		//functia friend pentru clasa PATRAT
			//functiile friend au acces la toate datele membre ale clasei
			//inclusiv cele PROTECTED si PRIVATE
			//daca stergem cuvantul cheie rezervat friend de la functia
Perimetru
			//aceasta nu mai are acces direct prin sageata -> la atributul
privat latura

		int Arie();

		//functiile Generic1 si Generic2 nu se declara in clasa
		//pentru ca nu sunt functii membre ale clasei

};//clasa se inchide mereu cu punct si virgula

//inafara clasei definesc metodele clasei PATRAT accesandule cu operatorul de
rezolutie
```

```
void PATRAT::SetLatura(int dim)//definitie (implementare) setter pentru latura
{
	latura=dim;
}

void PATRAT::SetCuloare(string clr)//definitie (implementare) setter pentru culoare
{
	culoare=clr;
}

int PATRAT::GetLatura()//definitie (implementare) getter pentru latura
{//returneaza intotdeauna atribute , nu variabile locale (dim)
	return latura;
}

string PATRAT::GetCuloare()//definitie (implementare) getter pentru culoare
{//returneaza intotdeauna atribute , nu variabile locale (clr)
	return culoare;
}

PATRAT::PATRAT()//definitie (implementare) constructor implicit
{//constructorul implicit atribuie valori default obiectelor
//care sunt create fara parametri
	latura=10;
	culoare="verde";
	cout<<"S-a creat un obiect implicit"<<endl;
}

PATRAT::PATRAT(int dim, string clr)//definitie (implementare) constructor cu
parametri
{//constructorul cu parametri atribuie valorile variabilelor date ca parametri
//este folosit de obiectele create cu valori efective
	latura=dim;
	culoare=clr;
	cout<<"S-a creat un obiect cu parametri"<<endl;
}

PATRAT::PATRAT(const PATRAT &p)
{//fiecare atribut al obiectului nou preia (latura)
//valoarea corespunzatoare a aceluiasi atribut din obiectul vechi(p.latura)
	latura=p.latura;
	culoare=p.culoare;
	cout<<"S-a copiat un obiect"<<endl;

}

PATRAT::~PATRAT()
{//destructor care sterge obiectele din memorie cand
//nu mai sunt folosite de metodele clasei
//si afiseaza un mesaj
	cout<<"S-a sters un obiect"<<endl;
}
```

```
//functie care NU este membra a clasei deci nu se acceseaza cu operatorul de
rezolutie
//nu se mai foloseste cuvantul cheie rezervat friend inafara clasei
int Perimetru(PATRAT *P)//definitia functiei friend
{//care calculeaza perimetrul
	int perimetru;
	perimetru=4*P->latura;//P fiind pointer se acceseaza cu operatorul sageata ->
	//latura este private, functia friend are acces la datele PROTECTED si PRIVATE
	return perimetru;
}

//este functie membra a clasei deci o accesam cu operatorul de rezolutie
int PATRAT::Arie()
{//care calculeaza aria
	int arie;
	arie=pow(latura,2);
	return arie;
}

//Generic1 nu se acceseaza cu operatorul de rezolutie :: pentru ca
//pentru ca nu este functie membra a clasei
void Generic1(PATRAT P)//primeste ca parametru un obiect de tipul PATRAT
{//copie pe stiva intr-o alta locatie de memorie datele obiectului primit ca
parametru
//deci apeleaza constructorul de copiere adica
//se afiseaza INTAI mesajul din contructorul de copiere
	cout<<"S-a folosit functia Generic1"<<endl;
	//APOI se afiseaza mesajul din Generic1
}

//Generic2 nu se acceseaza cu operatorul de rezolutie :: pentru ca
//pentru ca nu este functie membra a clasei
PATRAT Generic2(PATRAT &P)//primeste ca parametru o referinta la un obiect de tipul
PATRAT
{//nu mai copie pe stiva, ci doar pastreaza adresa data prin referinta
	cout<<"S-a folosit functia Generic2"<<endl;
	//se afiseaza INTAI mesajul din Generic2
	return P;
	//dupa care se copie pe stiva intr-o alta locatie de memorie datele obiectului
primit ca referinta
	//deci apeleaza constructorul de copiere adica
	//se afiseaza APOI mesajul din contructorul de copiere
}
```

```
//creez clasa derivata PIRAMIDA
//PIRAMIDA mosteneste datele membre din PATRAT
class PIRAMIDA : public PATRAT// se foloseste operatorul : , nu operatorul ::
{
	int inaltime;
	public://declar si definesc in interiorul clasei Setterul, Getterul si functia proprie Volum
		void SetInaltime(int dim)
		{
			inaltime=dim;//pot folosi din nou dim pentru ca e variabila locala
		}
		int GetInaltime()
		{
			return inaltime;
		}
		int Volum()
		{
			int volum;
			volum=Arie()*inaltime/3;
			//metoda Volum are acces la metoda Arie pentru ca mostenirea
			//public-public permite acest lucru
			return volum;
		}

};//si clasa derivata trebuie inchisa tot cu punct si virgula ;



int main()
{	//creez intai 4 obiecte de tip PATRAT (clasa de baza)
	PATRAT P1;//creez un obiect P1 care ia datele din constructorul implicit
	cout<<"Patratul P1 este de culoare "<<P1.GetCuloare()<<", are latura de "<<P1.GetLatura()<<" metri, perimetrul de "<<Perimetru(&P1)<<" metri si aria de "<<P1.Arie()<<" metri patrati"<<endl ;
	//la functia friend Perimetru , argumentul il oferim ca adresa (referinta) pentru ca la declaratie aceasta functie primea acest argument ca pointer

	PATRAT P2(15, "albastru");//creez un obiect P2 care ia datele din constructorul cu parametri
	cout<<"Patratul P2 este de culoare "<<P2.GetCuloare()<<", are latura de "<<P2.GetLatura()<<" metri, perimetrul de "<<Perimetru(&P2)<<" metri si aria de "<<P2.Arie()<<" metri patrati"<<endl ;

	PATRAT P3=P1;//creez un obiect P3 care copie datele obiectului P1; sintaxa standard
	cout<<"Patratul P3 este de culoare "<<P3.GetCuloare()<<", are latura de "<<P3.GetLatura()<<" metri, perimetrul de "<<Perimetru(&P3)<<" metri si aria de "<<P3.Arie()<<" metri patrati"<<endl ;

	PATRAT P4(P2);//creez un obiect P4 care copie datele obiectului P2; sintaxa alternativa;
	cout<<"Patratul P4 este de culoare "<<P4.GetCuloare()<<", are latura de "<<P4.GetLatura()<<" metri, perimetrul de "<<Perimetru(&P4)<<" metri si aria de "<<P4.Arie()<<" metri patrati"<<endl ;
```

```
	Generic1(P3);//intai apare mesajul din constructorul de copiere apoi mesajul
din Generic1
	Generic2(P4);//intai apare mesajul din Generic2 apoi mesajul din constructorul
de copiere(invers ca la Generic1)
	//functiile Generic1 si Generic2 reprezinta o modalitate de a folosi
constructorul de copiere, nu doar la crearea altor obiecte
	//ci si la copierea datelor din obiectele existente in alte locatii de memorie
decat cele initiale

	//creez acum un obiect de tipul clasei derivate PIRAMIDA
	PIRAMIDA PD1;//desi nu am constructori in clasa derivata, compilatorul imi va
oferi unul implicit
	//pentru ca nu am alti constructori declarati explicit in clasa PIRAMIDA
	PD1.SetLatura(25);//obiectul de tip piramida are acces la Setterii din clasa
PATRAT
	PD1.SetCuloare("galben");
	PD1.SetInaltime(5);
	cout<<"Piramida PD1 are baza de culoare "<<PD1.GetCuloare()<<", latura bazei
de "<<PD1.GetLatura()<<" metri, inaltimea de "<<PD1.GetInaltime()<<" metri, "<<endl;
	cout<<" perimetrul de "<<Perimetru(&PD1)<<" metri, aria bazei de
"<<PD1.Arie()<<" metri patrati si volumul "<<PD1.Volum()<<" metri cubi ."<<endl ;
	//Perimetru(&PD1) nu da eroare deoarece la functia friend au au acces atat
obiectele clasei de baza cat si obiectele clasei derivate


}
```