PROGRAMARE OBIECT-ORIENTATA

LABORATOR 3 – CONSTRUCTOR DE COPIERE, DESTRUCTOR, FUNCTII CU ARGUMENT OBIECTE SI REFERINTE LA OBIECTE, FUNCTII FRIEND

Constructorii de copiere sunt constructori care primesc ca parametru o referinţă la un obiect din aceeaşi clasă. În cazul în care nu există un asemenea constructor, compilatorul va genera un constructor de copiere care va copia bit cu bit atributele. Costructorul de copiere default copiaza fiecare variabila membru a obiectului transmis ca parametru in noul obiect. Se cheama ca se face o copiere inteligenta a membrilor.

Cu toate ca aceasta este bine pentru variabilele membre obisnuite, pentru variabilele membre de tip pointer nu se poate afirma acelasi lucru. Variabilele pointer ale celor doua obiecte (cel original si cel clonat) vor pointa catre aceleasi zone de memorie, ceea ce poate constitui un avantaj sau o sursa de mari probleme atunci cand se elebereaza aceasta zona de memorie insa raman obiecte al caror membre de tip pointer pointeaza catre aceste zone.

Solutia la aceasta problema este construirea propriului constructor de copiere ca in exemplu urmator:

```
#include<iostream>

using namespace std;


class FORTA

{//Atributele clasei Forta (sunt private by default)

	double masa;

	double acceleratia;

//Metodele clasei Forta

	public:

		FORTA(double m, double a);//Initializarea constructorului cu argumente

		FORTA( const FORTA&);//Initializarea constructorului de copiere

		~FORTA();//Initializarea destructorului

		double Masa(void) { return masa;}//Metoda ce returneaza masa obiectului

};
```

```
FORTA::FORTA(double m, double a)//Se scrie constructorul cu argumente
{
	masa=m;//se retine valoarea parametrului 1 dat de utilizator constructorului in
		//atributul masa
	acceleratia=a;//se retine valoarea parametruilui 2 dat de utulizator constructorului
		 //in atributul masa
	cout<<"Trec prin constructorul cu argumente\n";
		//Se afiseaza un mesaj pentru a cunoaste
		//ordinea apelarii instructiunilor in program
}

FORTA::FORTA(const FORTA &g)//Se scrie constructorul de copiere
{
	masa=g.masa;
	// se copiaza valoarea atributului masa al obiectului generic g de tip FORTA in
	 //variabila masa
	acceleratia=g.acceleratia;
	// se copiaza valoarea atributului acceleratia al obiectului generic g de tip
	 //FORTA in variabila acceleratia
	cout<<"Trec prin constructorul de copiere\n";
}

FORTA::~FORTA()//Se scrie destructorul
{
	cout<<"Trec prin destructor\n";
}
```

```
void idle1(FORTA f)
//Functie idle1 apelata in main, utila pentru intelegerea ordinii apelarii instructiunilor
{
	cout<<"Sunt in functia Idle UNU\n\n";
}
FORTA idle2(FORTA &f)
//functie idle2 ce preia un obiect de tip FORTA si apoi il returneaza dupa mesajul
//catre utilizator
{
	cout<<"Sunt in functia Idle DOI\n\n";
	return f;
}


int main()
{//Apelarea constructorilor cu parametri
/*1*/  FORTA f1(0,0);
/*2*/   FORTA f2(4.3,9.9);
	f1=FORTA(f2);//Folosim constructorul de copiere pentru a copia valorile atributelor
lui f2 in obiectul f1


	idle1(f2);//Se apeleaza functia idle1
//Apelarea constructorilor cu parametri
/*3*/  FORTA f3(7.5,9.8);
/*4*/  FORTA f4(0,0);
    f4=FORTA(f3);//Folosim constructorul de copiere pentru a copia valorile atributelor lui
f3 in obiectul f4


	FORTA f5 = idle2(f4);	//  Preiau rezultatul lui idle2(), pentru a demonstra ca si
aici s-a folosit constructorul de copiere
	cout << "The mass of the final force (#5): " << f5.Masa() << endl;//Se returneaza masa
fortei finale 5
```

```
	return 0;//Terminarea programului
}
```

Concluzii:

La Functia Idle la care numele nu are o semnificatie anume, exista doua cazuri:

1. **Cand primeste o referinta ca argument atunci se afiseaza mesajul de informare apoi mesajul din constructor. Acesta din urma se afiseaza din cauza ca a returna un obiect de tipul unei clase este echivalent cu a apela un contructor. Ceea ce se intampla de fapt in memorie este de fapt o copiere pe stiva. La returnarea obiectului f, se copie obiectul dat ca parametru in alta locatie de memorie , ca dovada ca din nou se apeleaza constructorul de copiere, deci implicit mesajul din interiorul acestuia.**
2. **Cand primeste un obiect de tipul clasei , atunci se apeleaza prima data constructorul deci se afiseaza mesajul din interiorul acestuia; in memorie se face copierea pe stiva a obiectului existent dat ca parametru , dar intr-o alta locatie de memorie, apoi se afiseaza mesajul de informare din functia Idle.**

Functia Idle nu este o functie membra a clasei deci nu apare in declaratia clasei, la inceputul programului.

Un al doilea exemplu din acest laborator este unul in care dorim sa prezentam efectul unei functii **friend** asupra unor tipuri de atribute private. Programul prezinta o modalitate de a transforma din coordinate polare in coordinate carteziene pentru doua puncte in spatiu.

Functia friend are urmatoarele proprietati: NU este membra a clasei dar are acces la toate tipurile de date membre ale clasei ! indiferent de modificatorul de acces aplicat asupra acestor date. In real-life este bine sa avem grija cand folosim cuvantul cheie friend, deoarece descopera toate datele si le expune la suprascrieri/modificari directe.

Codul sursa al programului este :

```
#include <math.h>

#include <iostream>
```

```
using namespace std;


class PUNCT
{
	double ro,fi, teta;//coordonatele polare
	double x, y, z;//coordonatele carteziene


	public:
	//setter pentru toate cele 3 coordonate polare
		void Setare(double rr, double ff, double tt);
	//transform din coordonate polare in carteziene cu functia Xyz
		void Xyz(void);
	//functia friend care nu este membra a clasei
	//iar in acest caz primeste ca argumente doi pointeri la obiecte de tip PUNCT
	//intrucat functia distp1p2 nu e membra a clasei (chiar daca e localizata in clasa)
		friend double distp1p2(PUNCT* p1, PUNCT* p2);


};


void PUNCT::Setare(double rr, double ff, double tt)
{//implementam setterul
	ro=rr;
	fi=ff;
	teta=tt;
}


void PUNCT::Xyz(void)
{//aplicam formulele de transformare din polar in cartezian
	double t;
	t=ro*cos(teta);
```

```
	x=t/cos(fi);
	y=t/sin(fi);
	z=ro*sin(teta);
}

double distp1p2(PUNCT* p1,PUNCT* p2)
{//Atentie ! functia friend nu se foloseste cu operatorul de rezolutie
//pentru ca nu apartine clasei DAR are acces la datele membre inclusiv cele private
//sau protected. in plus operatorul friend nu se mai foloseste inafara clasei (la definire)
	return sqrt((p1->x-p2->x)*(p1->x-p2->x) +
				(p1->y-p2->y)*(p1->y-p2->y) +
				(p1->z-p2->z)*(p1->z-p2->z)
			);
	//din cauza ca p1 si p2 sunt pointeri singura metoda sa exprim coordonata x1 este
	//prin operatorul sageata, este acelasi operator sageata pe care il utilizez
	//si cand folosesc pointerul this
}

int main()
{
	double dist;
	PUNCT u,v;//doua obiecte de tip PUNCT u.Setare(10.0,0.5,0.5);//aplic metoda
	Setare pentru obiectul u v.Setare(20.0,1.0,1.0);//aplic metoda Setare pentru
	obiectul v u.Xyz();//transform coordonatele punctului u din polar in cartezian
	v.Xyz();//transform coordonatele punctului v din polar in cartezian
	dist=distp1p2(&u, &v);//aplic functia friend de calcul a distantei dintre u si v
	//rezultatul il scriu ca valoare in variabila dist
	cout<<"distanta este "<<dist;//afisez valoarea variabilei dist
}
```

Concluzii:

Functiile prietene (friend) sunt functii ne-membre ale unei clase, care au acces la datele membre private ale unei clase. Functiile prietene ale unei clase trebuie precizate în definitia clasei. În acest sens, prototipurile unor astfel de functii sunt precedate de cuvântul cheie friend. Spre deosebire de functiile membre, functiile prietene ale unei clase nu poseda pointerul implicit this. De aceea, deosebirea esentiala doua functii care realizeaza aceleasi prelucrari, o functie membra si o functie prietena, consta în faptul ca functia prietena are un parametru în plus fata de functia membru.

Un al treilea exemplu din laboratorul curent, il reprezinta codul unui program care evidentiaza un polinom:

Ne propunem sa creeam o clasa Polinom ce sa defineasca un polinom, cat si realizarea unei functii ce sa realizeze operatia de inmultire a doua polinoame. Pentru inceput trebuie sa initiem in cadrul clasei gradul si termenii liberi ai unui polinom prin declararea unei variabile de tip unsigned “grad” si a unui vector de tip . coef[50].

Functiile ce vor fi implementate si folosite:

- Setarea gradului polinomului - void setGrad(unsigned x);
- Setarea unui coeficient specific - void setCoef (int x, int y);
- Obtinerea gradului polinomului - int getGrad();
- Obtinerea unui coeficient specific - int getCoef (int x);
- Constructor fara parametru - Polinom();
- Constructor cu parametru - Polinom(int x);
- Functie friend pentru inmultirea a doua polinoame;

Forma unui polinom:

$$\mathbf{a_0x^n+a_1x^{n-1}+\ldots+a_nx^0;}$$

$a_0, a_1, \ldots a_n$ – *termeni liberi*

n – *gradul polinomului*

```
#include <iostream>

using namespace std;

class Polinom
{

    unsigned grad; //declararea variabilei grad
    float coef[50]={0}; //declararea vectorului ce va retine coeficientii polinomului

    public:


        void setGrad(int x); // functie de setare gradului polinomului
        void setCoef ( int x, float y); // functie de setare a unui coeficient specific y pe
pozitia x
        int getGrad(); //functie de obtinere a gradului polinomului
        float getCoef (int x); //functie de obtinere a unui coeficient specific aflat pe
pozitia x
        Polinom(int x); //constructor cu parametru, ce seteaza gradul polinomului (grad = x);
        Polinom(); //constructor fara parametru, ce seteaza gradul polinomului 0
        friend void inmultire(Polinom *p1,Polinom *p2, Polinom *p3); //functie de inmultire a
polinoamelor
};
Polinom::Polinom()
{
    grad=0; //stabilim in cazul constructorului fara parametru ca gradul sa fie 0

}
Polinom::Polinom(int x)
{
    grad=x; //stabilim in cazul constructorului cu parametru ca gradul sa fie x

}
void Polinom::setGrad(int x)
{
    grad=x; //setarea parametrului pentru a fi x.
}
void Polinom::setCoef(int x, float y)
{
    coef[x]=y; //setam ca elementul de pe pozitia x a vectorului coef sa ia valoarea lui y
}
int Polinom::getGrad()
{
    return grad;
}
float Polinom::getCoef(int x)
{
    return coef[x]; //returneaza valoarea elementului x din cadrul vectorului coef
}
void inmultire(Polinom *p1,Polinom *p2, Polinom *p3)        //Inmulteste 2 polinoame p1 si p2
si rezultatul lor il scrie in p3, un al treilea polinom
{
```

```
    p3->grad=p1->grad+p2->grad; //gradul noului polinom va fi suma celor 2 polinoame
inmultite (in cazul neintelegerii puteti incerca sa inmultiti 2 polinoame de ordin mic:
x^2+x+2 si x+3 , iar gradul va fi 3)


        for( unsigned i=0;i<=p1->grad;i++)
            for(unsigned j=0;j<=p2->grad;j++)
                p3->coef[i+j]+=p1->coef[i]*p2->coef[j];

}int main()
{
    Polinom p1,p2,p3;
    int i=0;
    int x;
    cout<<"Gradul primului polinom este : ";
    cin>>x;
    if(x<0)
        {

        cout<<"Polinoamele nu pot avea grad negativ";
        return 0;

        }

    p1.setGrad(x);

    for(unsigned i=0;i<=p1.getGrad();i++)
    {
        cout<<"Coeficientul lui "<<i<<" este :";
        cin>>x;
        p1.setCoef(i,x);
    }

    cout<<"Gradul la al doilea polinom este : ";
    cin>>x;
    p2.setGrad(x);
        if(x<0)
        {

        cout<<"Polinoamele nu pot avea grad negativ";
        return 0;

        }

    for(unsigned i=0;i<=p2.getGrad();i++)
    {
        cout<<"Coeficientul lui "<<i<<" este :";
        cin>>x;
        p2.setCoef(i,x);
    }
    inmultire(&p1,&p2,&p3);
    i=p3.getGrad();
    if(p3.getCoef(i)<0)
    cout<<"-";
    x=0;
```

```
    for(;i>=0;i--)      // Afisarea
    {
        if(p3.getCoef(i)!=0)
        {

        if((p3.getCoef(i)==1||p3.getCoef(i)==-1)&&i!=0)
            {
            cout<<"x^"<<i;
            x++;
            }
            else if((p3.getCoef(i)==1||p3.getCoef(i)==-1)&&i==0)
            {
                cout<<p3.getCoef(i);
                x++;
            }
        else if(p3.getCoef(i)!=1)
        {
            cout<<p3.getCoef(i)<<"x^"<<i;
        }
        }
        if(i!=0&&p3.getCoef(i-1)>0)
            cout<<"+";
    }
    if(x==0)
        cout<<x;

    return 0;
}
```