OBJECT ORIENTED PROGRAMMING

LAB 4 – COPY CONSTRUCTORS (Cont.), DESTRUCTORS, FUNCTIONS WITH OBJECT ARGUMENT AND OBJECT REFERENCE

EXAMPLE 1

The copy constructors are constructors which receive as parameters a reference to an object of the same class. In case such constructors do not exist, the compiler will automatically generate a copy constructor and will copy bit by bit the attributes. The default copy constructor will copy each member variable of the object received as parameter into the newly created object. This is called *performing an intelligent member copy*.

While this bit by bit copy is good for regular variables, this can't be stated for pointer –type variables. The pointer-type variable for the two objects (the original and the clone) will point to the same memory region, which can be an advantage or a source of many errors (when this memory area is released but member objects whose member variables point to this memory area).

The solution to having the compiler generate the copy constructor (and potentially causing problems on classes with pointer-type data members) is building our own copy constructor like in the following example.

This example will also introduce

- the destructor, as a special function member (method) that is called when the object is about to be destroyed.
- other functions (not member functions), that reference objects and/or take an object as an argument – we will demonstrate the order of calling for the constructor and other operation with these function

On simple examples it does nothing, but on more complex examples it handles cleaning up after the object - including releasing memory, restoring program state (to prequel of object creation).

```
#include<iostream>

using namespace std;


class FORCE

{//Attributes for class Force (are private by default)

       double mass;

       double acceleration;
```

```
//Methods for class Force

	public:

		FORCE(double m, double a);//Initialization of constructor with
arguments

		FORCE( const FORCE&);//Initialization of copy constructor

		~FORCE();//Initialization of destructor

		double Mass(void) { return mass;}//Method returning the mass of the
object

};


FORCE::FORCE(double m, double a)// Expand the constructor with arguments

{

	mass=m;//retain value of 1st parameter  passed to the constructor

		 //in attribute mass

	acceleration=a;//retain value of 2nd parameter passed to the constructor

		 //in attribute acceleration

	cout<<"Passing through the constructor with arguments \n";//Print out a
debug

		 //message to know the order of calling instructions in our program

}


FORCE::FORCE(const FORCE &g)//Write the copy constructor

{

	mass=g.mass;// copy value of mass attribute of generic object g

			// of class type FORCE g

			//into variable mass

	acceleration=g.acceleration;// copy value of attribute acceleration

			// of generic object g of type FORCE

			//into variable acceleration

	cout<<"Passing through copy constructor \n";

}


FORCE::~FORCE()//Write the destructor

{
```

```
	cout<<"Passing through destructor \n";
}

void idle1()//Function idle1 called by main()
		//useful to understand the order of instructions
{
	cout<<"Entering function idle ONE \n\n";
}
FORCE idle2(FORCE &f)//function idle2 that acquires a ref to an object
			//of type FORCE and then it returns it
			//after a message to the user
{
	cout<<"Entering function idle TWO \n\n";
	return f;
}

FORCE idle3(FORCE f)//function idle3 that calls an object (not ref)
			//of type FORCE and then it returns it
			//after a message to the user
{
	cout<<"Entering function idle THREE \n\n";
	return f;
}

int main()
{//Call constructors with parameter
	cout << " f1" <<endl;
/*1*/  FORCE f1(0,0);

	cout << " f2" <<endl;
/*2*/   FORCE f2(4.3,9.9);
	f1=FORCE(f2);//Using the copy constructor to copy attribute values
```

```
                    //from source object f2 into destination object f1
       cout << "calling idle1" <<endl;
       idle1();//Call function idle1
//Call constructors with parameters
       cout << "f3" <<endl;
/*3*/  FORCE f3(7.5,9.8);
       cout << "f4" <<endl;
/*4*/  FORCE f4(0,0);


       cout << "f4 <- copy <- f3" <<endl;
    f4=FORCE(f3);//Using copy constructor to copy values for attributes
                //from object f3 into object f4
       cout << "calling idle2" <<endl;
       FORCE f5 = idle2(f4); //  Get the result of idle2(),
                     //passing a reference to class type
    FORCE f6 = idle3(f4);  //  Get the result of idle3(),
                        // passing an argument of class type
                        // to show usage of copy constructor in this scenario as
well
       cout << "The mass of the final force (#6): " << f6.Mass() << endl;//Return
final force from object f5
       return 0;//Program will end returning 0 to the environment that called it
}
```

Conclusions:

In function Idle for which the name has no meaning, there are two cases:

1. **When receiving a reference as an argument,** the function (Idle) displays the info message followed by the constructor message. The latter is caused by the fact returning an object of the class type is similar to calling a constructor *(a temporary variable/object will be created to hold the return value).*
2. **When receiving an object of the class type,** it will call the constructor first so it will show the constructor message, and only afterward it will show the info message inside the idle function.

Note: Function Idle is not a member function of the class so it does not show up in class declaration, in the beginning of the program.

EXAMPLE 2

The second example from this lab deals with inheritance: we want to show the simplest inheritance scenario, **public** in the base class followed by **public** in the inherited (public-public).

In lab 5 we will re-instantiate this example for the public-private scenario.

```
#include <iostream>
using namespace std;


class BASE
{//create class BASE
	int x;//private member / attribute
	public://the following will be public
		void initX(int n)//setter for  x
			{x=n;}
		void getX()//getter for x
			{cout<<x;}

};//always close the class with semicolon ;!

class DERIVATE:public BASE
{//create a derivate class
	int y;//private member / attribute
	public://the following will be public
	void initY(int n)//setter for y
			{y=n;}
		void getY()//getter for y
			{cout<<y;}
};

int main()
{//create the derivate class object.
	DERIVATE D1;
	D1.initX(10);//set the value for x: OK(no errors)
	D1.initY(20);//set the value for y: OK(no errors)
	D1.getX();//display the value for x: OK(no errors)
	D1.getY();//display the value for y: OK(no errors)
}
```

Because all the data is visible due to public inheritance, we have no error on accessing the methods of the derived class nor the base class.

A third example in today's lab is the source code for a program implementing a polynomial.

We want to create a class *Polynomial* to define a polynomial, as well as creating a function to implement the polynomial multiplication. For starters, we need to initiate within the class the degree and free coefficient of a polynomial via unsigned *grad* variable and an array of type *coef[50]*.

The functions to be implemented and used are:

- Setting the polynomial degree - void setDegree(unsigned x);
- Setting a specific coefficient - void setCoef (int x, int y);
- Getting the polynomial degree - int getDegree();
- Getting a specific coefficient - int getCoef (int x);
- Constructor without parameter - Polynomial();
- Constructor with parameter - Polynomial(int x);
- Friend function to multiply to polynomials;

Form of a polynomial:

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

$a_0, a_1, \ldots a_n$ – *coefficients*

n – *polynomial degree*

```
#include <iostream>

using namespace std;

class Polynomial
{

    unsigned degree; //declare variable:  degree
    float coef[50]={0}; //declare array for the coefficients of the polynomial

    public:


        void setDegree(int x); // setter for the degree
        void setCoef ( int x, float y); // setter for a specific coefficient y on position
x
        int getDegree(); //getter for polynomial degree
        float getCoef (int x); //getter for a specific polynomial coefficient on position
x
        Polynomial(int x); //constructor with parameters, setting x degree for the
polynomial
        Polynomial(); //no parameter constructor, setting polynomial degree to 0
        friend void multiply(Polynomial *p1,Polynomial *p2, Polynomial *p3); //polynomial
multiplication function
};
Polynomial::Polynomial()
{
    degree=0; //set degree 0 for constructor with no parameters

}
```

```
Polynomial::Polynomial(int x)
{
    degree=x; //set degree to parameter for constructor with parameter x

}
void Polynomial::setDegree(int x)
{
    degree=x; //set degree to argument value
}
void Polynomial::setCoef(int x, float y)
{
    coef[x]=y; //set coef[x] to value y
}
int Polynomial::getDegree()
{
    return degree;
}
float Polynomial::getCoef(int x)
{
    return coef[x]; //return value of x from array coef
}
void multiply(Polynomial *p1,Polynomial *p2, Polynomial *p3)        //Multiply 2
polynomials p1 and p2 and write the result into p3, a third polynomial
{
    p3->degree=p1->degree+p2->degree; //degree of new polynomial is the sum of the two
polynomials to be multiplied (egg: the degree of the multiplication of: x^2+x+2 and x+3,
will be 3)


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

}
int main()
{
    Polynomial p1,p2,p3;
    int i=0;
    int x;
    cout<<"Degree of first polynomial : ";
    cin>>x;
    if(x<0)
        {

        cout<<" Polynomial can't have a negative degree. ";
        return 0;

        }

    p1.setDegree(x);

    for(unsigned i=0;i<=p1.getDegree();i++)
    {
        cout<<"Coefficient "<<i<<" is :";
        cin>>x;
        p1.setCoef(i,x);
    }

    cout<<"Degree of second polynomial : ";
```

```
    cin>>x;
    p2.setDegree(x);
        if(x<0)
        {

        cout<<"Polynomial can't have a negative degree. ";
        return 0;

        }

    for(unsigned i=0;i<=p2.getDegree();i++)
    {
        cout<<"Coefficient "<<i<<" is :";
        cin>>x;
        p2.setCoef(i,x);
    }
    multiply(&p1,&p2,&p3);
    i=p3.getDegree();
    if(p3.getCoef(i)<0)
    cout<<"-";
    x=0;

    for(;i>=0;i--)      // Display
    {
        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;
}
```

Homework.

H4.1. Create an example class Product, with data members string product_type, string product_name, string barcode, float price. Implement setters, getters and a friend function to compare prices.

H4.2. Modify the example to implement an inherited class DiscountedProduct, which also has a data member float discount which applies to the product_name, if the product_type is “discount”. By default, products have no discount (unless otherwise set).

H4.3. Implement a friend function that computes the price for the following array of DiscountedProducts (it should discount Cola from 3.99 to ~0.99 and the total should become ~4.99).

```
soft_drink  Cola  1234  3.99   1
food   Fruit  1356  3.99    1
discount  Cola  0000    0   0.25
```