OBJECT ORIENTED PROGRAMMING

LAB 5 – COPY CONSTRUCTOR (Cont.), DESTRUCTOR (Cont.), FRIEND FUNCTIONS, SIMPLE INHERITANCE

FIRST EXAMPLE

A first program we will study in this lab will seek to verify the notions learned from the prior lab (lab4) in terms of copy constructor and destructor. We use as example through the class Book.

This class has as attributes the total number of pages, the current page number, the publisher's name associated with that Book. We construct Getter methods for all 3 attributes, but we do not need Setters since we will use a constructor with parameters.

The copy constructor, as usual, has the class name and as argument a constant reference to an object of the same type Book Via the argument reference we tell the compiler the memory address of that original object.

The destructor will have a message through which we mark the fact it is used during execution.

We also implemented two standard methods:

Read: this gets as argument an integer value to specify how many pages to read after verifying we do not exceed the total amount of pages. If the total number of pages is exceeded we will display a message on the screen, if not we will add to the current page number the number of pages to read (updating the total number of pages read up until then).

Difference: via this method we will compute how many pages we still need to read in order to finish the book.

Following is the source code for the Book class example:

```
#include <iostream>

using namespace std;


class BOOK

{

        int total_pages;

        int crt_page;

        string publisher;
        public:
                //no need for setters since we use a constructor with parameters (only in
                //this example)
        //following is public
        //until the next access modifier (usually private: )
        //so for this scenario all will be public until the end of class definition
                int GetTotalPag();
                int GetCrtPag();
                string GetPublisher();
                BOOK(int nr, string pub);
                BOOK(const BOOK&);//prototype for copy constructor
                //copy constructors are constructors = same name as the class
                //copy constructors = get argument in form of reference to an object
                // - an implementation of the class
                //- of the same class type
                ~BOOK();//prototype destructor

                int READ(int pages);//prototype for standard method
                int DIFFERENCE();//prototype for standard method

};//end class definition with semicolon character ;

int BOOK::GetTotalPag()
{
        return total_pages;
}

int BOOK::GetCrtPag()
{
	return crt_page;
}

string BOOK::GetPublisher()
{
	return publisher;
}
BOOK::BOOK(int nr, string pub)
{
	//initialize the total number of pages with nr
	total_pages=nr;
	//initialize current page with 0
	//(how many pages I read so far)
	crt_page=0;
	//initialize publisher with value from string pub
	publisher=pub;
	//display this message upon each call of the constructor
	cout<<"Called constructor with arguments \n";
}

BOOK::BOOK(const BOOK& c)
{
	total_pages=c.total_pages;
	crt_page=c.crt_page;
	publisher=c.publisher;
	cout<<"Called copy constructor \n";
}

BOOK::~BOOK()
{
        cout<<"Called destructor \n";

}



int BOOK::READ(int pages)

{//test if the number of pages to read exceeds total_pages

        if(crt_page + pages >= total_pages)

        {//if it exceeds only display message

                cout<<"done reading the book "<<endl;

        }

        else{//if it does not exceed the total number of pages

        //increase the current page -crt_page-with the number of pages read -pages-

        crt_page =crt_page + pages;

        }


}


int BOOK::DIFFERENCE()

{//how many pages I got left from the book starting from -crt_page- to -total_pages-

        int difference;

        difference=total_pages-crt_page;

        return difference;

}


int main()

{

        BOOK c1(60,"Packt");//create an object of type Book which will take its

        //attributes from the constructor with 2 arguments

        BOOK c2=c1;//effect 1: create object c2 in same way as c1

        //effect2: copy each attribute of c1 into c2
        c2.READ(20);//apply READ method to object c2

        cout<<"I read "<<c2.GetCrtPag()<<" pages "<<endl;

        cout<<"I still need to read "<<c2.DIFFERENCE ()<<endl;//compute how many pages
I got left from book c2

        c2.READ(20);//apply READ method to object c2, again

        cout<<"I read  "<<c2.GetCrtPag()<<" pages"<<endl;

        //display current page/number of pages read via getter




}
```

SECOND EXAMPLE

A second example from this lab consists in illustrating the effect of a friend function over private data attributes. The program will present a way to transform from Polar to Cartesian coordinates for two points in space.

The friend function has the following properties: IS NOT a member of the class but it has access to all types of member data of the class! no matter what access modifier is defined with each data member. In real-life it is good to be careful in using the friend keyword, since it uncovers all member data and exposes them to accidental overwrite/modification.

The source code of the program is:

```
#include <math.h>

#include <iostream>


using namespace std;


class POINT

{

        double ro, fi, theta;// polar coordinates

        double x, y, z;//Cartesian coordinates


        public:
```

```
        //setter for the 3 polar coordinates

        void SetPol(double rr, double ff, double tt);

        //transform from polar to Cartesian coordinates via function Xyz

        void Xyz(void);

        // friend function is not member of the class

        //  (even if it is located in the class definition)

        //   for this example, it has 2 arguments pointers to POINT objects

        friend double distp1p2(POINT* p1, POINT* p2);


};


void POINT::SetPol(double rr, double ff, double tt)

{//setter implementation

        ro=rr;

        fi=ff;

        theta=tt;

}


void POINT::Xyz(void)

{//apply coordinate transformation polar → cartesian

        double t;

        t=ro*cos(theta);

        x=t/cos(fi);

        y=t/sin(fi);

        z=ro*sin(theta);

}


double distp1p2(POINT* p1,POINT* p2)

{//Attention ! The friend function does not use the scope operator (so no POINT:: in
it's declaration)
```

```
//because it does not belong to the class BUT has access to member data, including
private (or protected) member data

//the friend operator is not used on the external definition

       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)

                       );

       //p1 and p2 are pointers to objects, not actual objects.

       //  Because of this reason, we use -> operator instead of .

       //  to select a member of the class

       //same as when using the this pointer

}


int main()

{

       double dist;

       POINT u,v;//two POINT objects

       u.SetPol (10.0,0.5,0.5);//apply SetPol method for object u

       v.SetPol(20.0,1.0,1.0);// apply SetPol method for object v

       u.Xyz();//transform point u coordinates from polar to cartesian

       v.Xyz();//transform point u coordinates from polar to cartesian

       dist=distp1p2(&u, &v);//apply friend function to compute distance

              //between u and v

              //and write result in variable dist

       cout<<"Distance is "<<dist;//print out the value of distance –dist-

}
```

Conclusions:

Friend functions are non-member functions of a class that have access to member data no matter of their class access type (public, protected, private). Friend functions of a class need to be mentioned within the class definition. For this purpose, the prototypes for friend functions are

preceded by the keyword *friend*. Unlike member functions, friend functions do not implicitly possess the pointer *this*. That is why, when comparing two means to implement a feature via member function or friend function, the friend function has 1 extra argument versus the member function.
A function can be at the same time member function into a class and friend function to another class.

HOMEWORK (OPTIONAL)

In the following example, *fA1* is member function into class *cls1* and friend function into class *cls2*.

```
class cls1

{

public:

    void fA1(A& a); // prototype

    void fA2(A& a); // prototype

};


class cls2

{

public:

    friend void cls1::fA1(A& a);

    void fB2(){}

};
```

As an additional assignment, fix this code to work.

THIRD EXAMPLE

A third example wishes to illustrate the order of calls of constructors and destructors in base and derivate classes. We will place in each of the base and derivate classes a constructor and a destructor. Each one will have a display message to show the order of calling.

```
#include <iostream>


using namespace std;
```

```
class BASE

{

       public:

              BASE()

              {cout<<"calling base class constructor "<<endl;}


              ~BASE()

              {cout<<"calling base class destructor "<<endl;}

};


class DERIVATE:public BASE

{

       public:

              DERIVATE()

              {cout<<"calling derivate class constructor "<<endl;}


              ~DERIVATE()

              {cout<<"calling derivate class destructor "<<endl;}

};


int main()

{//create object of type DERIVATE

       DERIVATE D1;

}
```

Conclusion is that upon creating object D1,

1. we first call the **base** class constructor
2. then we call the **derivate** class constructor

and upon the end of the program we call them in reverse order

1. we call first the **derivate** class destructor
2. then we call the **base** class destructor

FOURTH EXAMPLE

A fourth example to study in this lab is to create a program that proves that constructors are not inherited as it happens with regular or special function members.

```
#include <iostream>


using namespace std;


class BASE

{

    public:

          BASE() {cout << "calling implicit constructor " << endl;}

          BASE(int a) {cout << "calling constructor with 1 argument "<< a <<endl;}

          BASE(int a, double b) {cout << "calling constructor with 2 arguments: "<< a
<< "," << b <<endl;}

};


class DERIVATE : public BASE

{

   //the compiler will offer a constructor implicit for DERIVATE

   //since we do not create one ourselves.

};


int main()

{

   DERIVATE d1; //OK, using the compiler-created default constructor

   DERIVATE d2(42);//would like to call the 1 argument BASE constructor but this

       //is not being inherited we get a compile-time error

   DERIVATE d3(42, 3.14);//would like to call the 2 argument BASE constructor but

       //is not being inherited so we get another compile-time error
   return 0;

}
```

HOMEWORK (OPTIONAL)

As an additional assignment, fix the above code to work.

FIFTH EXAMPLE

The last exercise from this lab aims to study the simple inheritance of type **public** in base class and **private** in derivate class. It is similar to the last example from lab 4 (where we studied the public-public scenario), only this time around we change the inheritance access modifier of the class After the initial source code we display the error that is normal to expect in this case.

FIFTH EXAMPLE (NON COMPILING VERSION)

```
#include <iostream>

using namespace std;

class BASE

{//create class BASE

       int x;// private member

       public://the following are public

              void initX(int n)//Setter for x

                     {x=n;}

              void getX()//Getter for X

                     {cout<<x;}


};//end class definition with semicolon character ; !


class DERIVATE:private BASE

{//create a derivate class

       int y;// private member

       public://the following are public

              void initY(int n)//setter for y

                     {y=n;}
              void getY()//getter for y
                     {cout<<y;}
};

int main()
{//create the DERIVATE object
       DERIVATE D1;
       D1.initX(10);//Error: initX is not accessible
       D1.initY(20);//set the value for y: OK(no errors)
       D1.getX();//Error: initX is not accessible
       D1.getY();//get the value for y: OK(no errors)
}
```

Errors that appear in this scenario are similar to:

```
D:\POO\LAB5\lab5-5.cpp In function 'int main()':
9       8       D:\POO\LAB5\lab5-5.cpp [Error] 'void BASE::initX(int)' is inaccessible
29      13      D:\POO\LAB5\lab5-5.cpp [Error] within this context
29      13      D:\POO\LAB5\lab5-5.cpp [Error] 'BASE' is not an accessible base of 'DERIVATE'
11      8       D:\POO\LAB5\lab5-5.cpp [Error] 'void BASE::getX()' is inaccessible
31      10      D:\POO\LAB5\lab5-5.cpp [Error] within this context
31      10      D:\POO\LAB5\lab5-5.cpp [Error] 'BASE' is not an accessible base of 'DERIVATE'
32      11      D:\POO\LAB5\lab5-5.cpp [Error] expected '}' at end of input
```

As an exercise, we want to change this source code so as to no longer receive errors, but still keep the public access in the BASE class and private access in the DERIVATE class.

Solution is that instead of creating a setter for Y to make a common setter for both classes, and when it is time to set x=n (for which x is private) to call the setter for x from within the common setter. Similarly for the getter: instead of a getter for y we create a common getter for x and y, and in this one I call the getter for x which is public.

The modified source code looks like this:

FIFTH EXAMPLE (WORKING VERSION)

```
#include <iostream>

using namespace std;


class BASE

{//create class BASE

        int x;// private member

        public://the following are public

                void initX(int n)//Setter for x

                        {x=n;}

                void getX()//Getter for X

                        {cout<<x;}


};//end the class with semicolon character ; !


class DERIVATE:private BASE

{//create derivate class

        int y;// private member

        public://the following are public

                void initXY(int n, int m)//setter for y, calling setter for x

                        {initX(n); y=m;}

                void getXY()//getter for y, calling getter for x

                        {getX(); cout<<y<<endl;}

};

int main()

{//create object of DERIVATE class type

        DERIVATE D1;

        D1.initXY(10,20);//ok

        D1.getXY();//ok

}
```