OBJECT ORIENTED PROGRAMMING

LAB 2 – INTRODUCTION TO CLASSES AND OBJECTS

Theoretical introduction

When we want to build *new types of data* that did not exist before in the C++ language, we should use the *Class* type. The keyword used to create it is “class”, and it is reserved by C++ (it can’t be used as a variable name in the C++ source code). The *Class* is the newly defined data type. The Objects are the instances of a class. We can create as many objects (instances) of a class type we defined in our program.

A class has two main elements: **Attributes** (data members) and **Methods** (member functions)

- The **Attributes** identify the properties that the new data type we define can have. The attributes are actually implemented via the variables known in C++. Just as any variables, attributes must have a type and a name (egg: int weight, double real, string name …). Example: a newly defined class named “Complex” can have as attributes: the real part of the number and the imaginary part of the number. Another example, a newly defined class called “Person” can have a set of attributes including: surname, name, age, weight.

- The **Methods** are possible actions that can be accomplished using the attributes. The methods are identified in procedural programming as functions that have a data type, a name, a function body (that describes how the function works) and sometimes a return value (computed based on the function’s data types).

For any class we can write a set of **special methods**:

- The “*setter*” methods are used to indirectly set the values for class attributes
- The “*getter*” methods are used to indirectly get the values for class attributes

The order of operations is to first use a setter then a getter. The setter is used first to place in the attribute a known value, and avoid having bugs caused by uninitialized variables in our source code, which can be difficult to debug and can seem to appear randomly.

Setter methods will have a void return type, because they do not return anything, but will have an argument: the value to be set. Getter methods will have a return type for the type of variable (from the available attributes) they return, and doesn’t need to have an argument.

As example, if we consider the complex class described above, the getter for the real part of the number will return a double and the getter for the imaginary pat of the number will return a double. If we consider the person class, the getter for surname will return a string, the getter for age will return an int, etc…

The compiler will deduce that the method is a getter or a setter based on the presence or absence of the function argument. If the method has an argument it will call the setter, if it has no argument it will call the getter. The concept behind this compiler behavior is called polymorphism, and it will be discussed in detail in a latter lab.

A special kind of methods are represented by the *constructors*. Each class has an implicit constructor that is always called upon the creation of a new object. If the programmer does not specify one's own constructor, the compiler will use the implicit constructor (which does not add any instructions to the source code). As programmers, we can write our own implicit constructor. In this later case, the compiler will not automatically create a constructor. The implicit constructor can be used to provide default values for the object instance attributes, upon the creation of the object, and thus avoiding calling the setter method for that object.

There are three ways to set values in the attributes:

- Directly: it is not the desired way in object oriented programming because we want to protect the attributes in order to avoid receiving unwanted values. We used to set variables directly in procedural programming.
- Indirectly via the setter: it is the preferred way to be used by object oriented programmers, since we can perform additional checks (filters) regarding the value (and sanity) of the attribute values. Example: if we want to set the attribute denominator of a fraction object, then in the setter method we should check the value to write is non-zero (to avoid divide by 0 in the fraction – leading to undefined results).
- Indirectly via the implicit constructor. This method can be used when we don't want to use a setter and we are ok with each object of the given class type being initialized to the same default values.

The last kind of standard methods are specific to each class type in particular. For example, for the class Person we can associate the methods: Eat(), Diet(), Exercise(), PersonalProfile(). Another example, for the class Gun we can associate the methods: LoadBullets(), Fire (), SetSafety ()..etc.

We can use access modifiers (sometimes called accessors) for both attributes and methods of a class. The access modifiers are: public, protected, private. Everything aware of the class is aware of the public attributes and methods and thus can access them directly. Only the children (and their children) classes are aware of protected attributes and methods. Only within the class instance we have the ability to access private attributes or methods. This restriction allows the programmer to better control access into classes. Some classes can be part of proprietary code or code not intended for future support. As such, only certain methods are accessible for outside code, and access into attributes is done only via those methods. This also allows for better source control during development, and helps with debugging (by reducing the source code pathways that can change a piece of data).

This is different than in the C (or C++) language behavior of access into the struct data type. The data members inside the structure is always accessible from the outside (so the access modifier for struct is always public).

For classes, the opposite is true. Implicitly, all data members of a class are pirvate, so they can't be accesses outside the class unless we use accessors.

In the current lab we will illustrate the effect of the "*public*" accessor. For a class, we want the attributes to remain private, the setter, getter and constructor methods to be done inside the class, and we will define these methods outside the class because the public accessor allows it.

As a first example we will create a class to model a complex number, named "Complex".

The attributes (data members) of the Complex class are: the real part of the complex number ***_re*** and the imaginary part of the complex number ***_im***. These don't have accessors (access modifiers) so they will be private, inaccessible from outside the class. As a form of good practice we will prefix the attributes with the "_" character, to distinguish attributes from methods. The methods are called Re(), and Im(). As a form of good practice we use the CamelCase naming convention for method names (designation formed by concatenating the words of the method name contain upper for the first letter and lower for the other, for each word in the designation except the first word, which can be all lowercase).

We will define a setter and a getter for both Re() and Im(), for a total of 4 methods.

The fifth method will be the constructor.

The following example source code creates the Complex class:

```
#include <iostream>


using namespace std;


class Complex //create a new class type named Complex

{

// no accessor here means we are using the default - private for the
following (until the next accessor) - so for _re and _im

      double _re; // declare the two attributes as private _re and _im
      double _im; // prefix the attributes with _ to distinbguish them from
methods

      //from here onward the accessor is public (methods are public for this
class example)

      public:

            //declaring the class methods inside the class
            //we can define them outside the class due to the "public:"
accessor


            void Re(double val); //setter method for Re() - public

            void Im(double val); //setter method for Im() – public

            double Re(); //getter method for Re() - public

            double Im(); //getter method for Im() - public


            Complex(); //implicit constructor for class Complex


};

//the scope resolution (::) operator below says Re() belongs to class Complex

void Complex::Re(double r)//define setter method for real part, argument r

{

      _re=r; //attribute _re is set with value from method Re() argument r

}


void Complex::Im(double i)// define setter method for imaginary part,
argument i

{

      _im=i; // attribute _im is set with value from method Im() argument i

}


double Complex::Re() // define getter method for real part

{

      return _re;

}


double Complex::Im() // define getter method for imaginary part

{

      return _im;
```

```
}



Complex::Complex()//define class Complex constructor

{

      //Need the scope operator (::) since we are outside the Complex class

      // scope. Useful for multi-class multi-file .cpp programs

      _re =0.0; //set default values for attributes

      _im =0.0; //to avoid uninitialization problems if not using setter
method

      //upon object creation they are initialized with values from
constructor

}


int main()

{

      double real,imag;

      Complex c0;//create object named c0, as instance of class Complex

                  //using setters

      cout<<"real part is:"; //cout is iostream's printf equivalent

      cin>>real; //cin is iostream's scanf equivalent

// notice the difference in direction of the  >> << operators

// used for cout and cin

      cout<<"imaginary part is:";

      cin>>imag;

      c0.Re(real);

      c0.Im(imag);

      // print attributes using getters

      cout <<"Complex number entered by keyboard is: "<<c0.Re()<< " + " <<
c0.Im()<<"i\n";


      Complex c1; //create object named c1, as instance of class Complex
```

```
                //use setters to set values in c1

        c1.Re(2.0); //use setter to set C1’s real part

        c1.Im(3.5); // use setter to set C1’s imaginary part

        // print attributes using getters

        cout <<"Complex number obtained by using setters is: "<<c1.Re()<< " + "
<< c1.Im()<<"i\n";


        Complex c2;//create object named c2, as instance of class Complex

        //will get it’s values from implicit constructor

        cout <<" Complex number obtained by using implicit constructor
is:"<<c2.Re()<< " + " << c2.Im()<<"i\n";

        return 0;

}
```

The second example studied is class Person. This class will have as attributes: Surname, Name, Age, Weight. We will code the setter and getter for each of the 4 attributes, and then the implicit constructor to initialize object instances of class Person with attributes of default value.

As standard methods we will implement:

- Method TestAge() – it will take a number from the keyboard and test if it is between thresholds set by the programmer and only if it passes the sanity check it can be set for the person

- Method Eat() – if the person has its weight between thresholds set by the programmer, it will increase the weight attribute by 10 (kilograms). If the upper threshold is met or exceeded (the person is overweight), we return an error message (instead of increasing parameter weight).

- Method Diet() – if the person has its weight between thresholds set by the programmer, it will decrease the weight attribute by 10 (kilograms). If the lower threshold is reached we return an error message instead. By imposing the lower threshold, we can avoid obtaining a negative weight value (as a consequence of applying the Diet() method and decreasing the weight by 10).

- Method Profile() – will return the state of the person’s weight: thin, normal, overweight etc.

Let’s examine the source code for this class:

```
#include <iostream>

using namespace std;  //std namespace includes definitions for iostream

class Person
{
      string _surname,_name; //class Person attributes - private
      int _age,_weight;

      public:

            void Surname(string n); //setter for surname
            void Name(string p); //setter for name
            string Name(); //getter for name – notice the string type used
            string Surname(); //getter for surname
            void Age(int a); //setter for age
            int Age(); //getter for age

            int TestAge(); //public standard methods
            int Eat();
            int Diet();
            int Profile();

            Person();//declaration of implicit constructor for class Person

};

void Person::Surname(string n) //setter for surname
{
      _surname=n;
```

```
}

void Person::Name(string p) //setter for name
{
	_name=p;
}

string Person::Surname() //getter for surname
{
	return _surname;
}

string Person::Name() //getter for name
{
	return _name;
}

int Person::Age() //getter for age
{
	return _age;
}

void Person::Age(int a)
{
	_age=a;
}

int Person::TestAge()
{//set age if within thresholds
	int a;
```

```
    do

    {

        cout<<"Insert person age : ";

        cin>>a;


    }while(a<0 || a>200); //test if age is valid

    _age=a;

    return _age;

}


int Person::Eat()//standard method

{

     if(_weight<=80)//test for upper threshold - person overweight

     {

        _weight=_weight+10;

     }

     else{

             cout<<"Person can’t eat anymore and need to diet first.\n";

         }


     return _weight;

}


int Person::Diet()

{

    if(_weight<=30)//test for lower threshold - person too thin

    {

        cout<<"Person too thin to diet and needs to eat first.\n";


    }
```

```
        else{
                _weight=_weight-10;
            }
        return _weight;
}



int Person::Profile()
{
        switch(_weight)
        {
                case 30:
                        cout<<"Person is underweight.\n";
                        break;
                case 40:
                        cout<<"Person is too thin.\n";
                        break;
                case 50:
                        cout<<"Person is thin.\n";
                        break;
                case 60:
                        cout<<"Person is normal.\n";
                        break;
                case 70:
                        cout<<"Person is a little overweight.\n";
                        break;
                case 80:
                        cout<<"Person is overweight\n";
                        break;
                default:
                    cout<<"Error establishing profile.\n";

                    break;

        }

}



Person::Person() // define the constructor for class Person

{

        _surname="Doe";

        _name="John";

        _age=30;

        _weight=60;

}


int main()

{

        Person standard;//using implicit constructor

        cout <<"Complete implicit student name is : "<<standard.Name()<<"
"<<standard.Surname()<<"\n";


        Person student;//using setters

        student.Surname("Smith");//object student calls setter method Surname()
and the _surname attribute for the object will take the value "Smith"

        student.Name("John");

        cout <<"Student  "<<student.Name()<<" "<<student.Surname()<<" is
"<<student.Age()<<" years old.\n";


        student.Age(40);//using setters

        cout<<"New student age is "<<student.Age()<<" years old.\n";


        //setting the age after checking for sane limits

        cout<<"New student age is "<<student.TestAge()<<" years old. \n";
        Person p4; //create object p4 of type Person
        p4.Profile(); //
        cout<<"New weight after eating is "<<p4.Eat()<<"kg.\n";
        p4.Profile();
        cout<<"New weight after eating is "<<p4.Eat()<<"kg. \n";
        p4.Profile();//use profile between successive calls to eat/diet
        cout<<"New weight after eating is "<<p4.Eat()<<"kg. \n";
        p4.Profile();
        cout<<"New weight after eating is "<<p4.Eat()<<"kg. \n";
        cout<<"New weight after eating is "<<p4.Eat()<<"kg. \n";
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        p4.Profile();
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        p4.Profile();
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        cout<<"New weight after dieting is "<<p4.Diet()<<"kg. \n";
        p4.Profile();
}
```

Homework:

Model in C++ the classes Car and Wildlife by choosing attributes and writing for each class: setters, getters, implicit constructor as well as standard methods.

Write each class in a separate .cpp file.