OBJECT ORIENTED PROGRAMMING

LAB 6 – FUNCTION POLYMORPHISM, MULTIPLE INHERITANCE

A first program in this lab is used to study the concept of polymorphism. This concept involves reusing a function or an operator to have a different meaning.

In case of **operator polymorphism**, we can provide the example of the << operator, as it was used during prior lab sessions. By default it is used to shift by a number of digits in a fixed point number ($1<<2 = 2^2 = 4$). One example of C++ reuse of the << operator is in *cout* sentences to redirect streams to the console. Another example involves the + operator. It's default use case is to add two fixed-point (char, integer, long …) or floating point (float, double …) numbers, but it cannot add complex numbers. In order for + to add complex numbers we can use polymorphism to offer a new functionality to the + operator via C++.

In case of function polymorphism, we can use the same name, in order to implement various working functionalities under the same name. One such example of function polymorphism that you already used during these lab sessions is the constructor for which we already have seen that we can achieve different functionalities if the constructor has zero or more than zero arguments. The degrees of freedom with which we can work to create these modes of operation of a function are:

- By using a **different return type** for the function
- By using a **variable number of arguments** for the function
- By **changing the data type** for a given argument
- **Combination of the 3 methods** presented above

We as the question how does the compiler know to decide which of the functions to use, if in the main function we call the function by the same name. The answer, the compiler identifies the function not just by its name but by the returned type and the number and type of arguments. The compiler analyses all 3 of these (**returned type, number of arguments, type of arguments)** and "seeks" out a proper fit within the available function prototypes and seeks the corresponding declaration. If the prototype is found the body of the function corresponding to that prototype is called, and if no exact match is found, we will receive a compiler error EVEN IF THERE ARE FUNCTIONS WITH THAT NAME within our source code!

Attention ! the concept of polymorphism does not refer to the scenario where functions have the same name but written with letters in different caps (lower, upper). C and C++ are case sensitive languages and names with variations in lower case upper case are considered to be different. Foe example the names *sum*, *Sum* and *SUM* are different in C/C++, and any functions with these names is considered different. If we define a prototype using "*Sum*" and we call it by using "*SUM*" we

will get an error of type undefined function, signaling that "SUM" does not exist within our code (and we have not defined it).

If we restrict the discussion to polymorphism for **object oriented programming**, on constructor creation, invariably we use this concept because the **name of the constructors need to match the name of the class**, so all the constructors need to have the same name, and they differ by the name and type of arguments. As we know, if we don't have arguments we create the implicit constructor, and if we do have arguments, then we transform it into a constructor with parameters (arguments). Another particular use case is when we have exactly one argument and the type of it is `const` pointer to the same class (which we know to be the copy constructor).

In the following source code, we provide an example of polymorphism for object oriented programming applied on the methods of a class:

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


class printData //class name
{
	public: //all methods public with the same name
	//print method is overloaded by polymorphism
		void print(int i)
		{//method print with 1 argument of type int
			cout << "Printing int: " << i << endl;
		}


		void print(double f)
		{//method print with 1 argument of type double
			cout << "Printing float: " << f << endl;
		}


		void print(char* c)
		{//method print with 1 argument of type pointer to char (char *)
			cout << "Printing character: " << c << endl;
		}
```

```
};


int main(void)
{
	printData pd;//create object of type printData
	char *sir = "This is C++"; // we can do dynamic allocation here

	// Call function variant with 1 integer argument
	pd.print(5);

	// Call function variant with 1 float argument
	pd.print(500.263);

	// Call function variant with char * (to a set of characters) argument
	pd.print(sir);

	return 0;
}
```

==========

A second program in this lab refers to the type of public inheritance of **protected data in the base class**. The effect of using the *protected* access modifier is the data will be **accessible in the base class directly**, but inaccessible **outside it and its derivate**.

The implementation of the source code is depicted below:

==========

```
#include <iostream>


using namespace std;


class BASE
{
	protected:
```

```
            int a,b;//2 variables of access modifier type protected


//an access modifier's effect spans until the next one,
//public: in our example
      public:
            void SetAB(int n, int m)
            //int n from here has visibility only within SetAB function
            //so is inaccessible from derivate class
            {//multiple setter for a and b of type public
                  a=n, b=m;
                  //give a value from n
                  //give b value from m
            }
};


class DERIVATE:public BASE//inherited class with name DERIVATE
{//inherits class BASE with public modifier
//public inheritance means I have access to BASE public and protected members
//
      int c;//no modifier access before, means it is private
      public://setter for c declared with public access modifier
            void SetC(int n)
                  {//attention! I can use variable n again
                  //because for the prior use n has visibility only within SetAB
                  //not function SetC,
                        c=n;//give c value of n
                        //n is of type integer
                  }


            void GetABC()
```

```
                        {//display all attributes,
                        // how many attributes are in class DERIVATE?
                        //      answer: 3 attributes
                        //question: how many attributes are inherited?
                        //      answer: 2 attributes
                        //question: how many attributes are own to DERIVATE?
                        //      answer: 1 attribute
                               cout<<a<<" "<<b<<" "<<c<<endl;
                               //a and b can be displayed
                               //because they are public in the base class
                               //and are inherited public
                        }
};


int main()
{
       DERIVATE D1;//create an object of derivate type with name D1
       D1.SetAB(4,5);//call the setter for a and b,
       //public in base class, public in inheritance, so visible in main
       D1.SetC(8);//call setter for c,
       //public in base class, public in inheritance, so visible in main
       D1.GetABC();
       //public in base class, public in inheritance, so visible in main
}
================================================================================
```

A third program wants to implement a multiple inheritance on two levels.

We build a class Resistor, which has as **protected attribute** the **value of resistance R**. We then build a constructor for this class with argument to set the default value for R. We need a function GetR to get (for the purpose of displaying on the console) the value of the resistance.

We build another class Voltage, derived in public mode from class Resistor, by using the `virtual` keyword. Here, the virtual keyword will cause us, upon the creation of the inherited object, **to have a single copy in memory of the base class (access being done via reference)**. This has 2 protected attributes that will be accessible only to inherited classes (like the OHMsLaw class described below). We create a constructor for this class with name Voltage with 2 arguments: one inherited by calling the constructor from the base class (R) and another belonging to Voltage and set locally (U). With the 2 we then compute the current I using the formula from OHMs law for a portion of circuit. I=U/R. After these steps I print the value of the current by using a getter.

We then build a third class, called Current, derived in public mode AGAIN from class Resistor again using the word `virtual`, so as to have upon creation of the inherited object a single copy of the base class (employing access by reference) – **in fact being the same base copy used by the class Voltage**. This has 2 protected arguments which will be accessible just to inherited classes (like OHMsLaw, defined below). We create a constructor for this class with the name Current, with 2 arguments: one inherited by calling the constructor of the base class (R), another belonging to Current and set locally (I). With the two I compute the voltage for a portion of circuit using OHMs Law formula: U=I*R. At the end I use a getter to display the value computed for U.

A final class in this example is the class OHMsLaw that performs **multiple inheritance from both Voltage and Current, both publicly. This class no longer employs the keyword virtual since it is inheriting different classes**. Keyword virtual can only be applied when 2 classes inherits (point to) the same source class. This class has no intrinsic attributes, only inherited ones. This class has a constructor with parameters.

In order to inherit the attributes we call the constructors for the base classes from the constructor with arguments. Firstly, we call the ones from the first level up (Current and Voltage), then the one inherited from two levels up (Resistor). In this constructor with arguments we display a message to signal the call of this constructor upon object creation.

In main function, we declare an object of the derivate class type of second level (OHMsLaw) upon whom we pass 3 arguments.

We compute the values for voltage and current and the display the values for all 3 physics metrics.

We follow the flow of the source code below:

```
#include <iostream>
```

```
using namespace std;


class Resistor
{
	protected://protected data accessible within derivate classes
			//but not outside them
		int R;
	public:
		Resistor(int r)
		{//constructor with 1 argument for base class
			R=r;//give R value from r (not vice-versa!)
			//if I would have written r=R,
			//it would allocate a value for R randomly from memory
		}
		virtual void GetR()
		{
			cout<<R<<endl;//display value for R
		}
};


class Voltage:virtual public Resistor //inherit base class by reference
{//upon creating Voltage objects (first derivate class), due to the
//keyword virtual
//we won't create a copy of the base class and it will be referenced instead.
//all objects of type Voltage will reference class Resistor without
//creating copies in memory for each of these objects
	protected:
		double U;
		double I;// protected data accessible within derivate classes
				// but not outside them
```

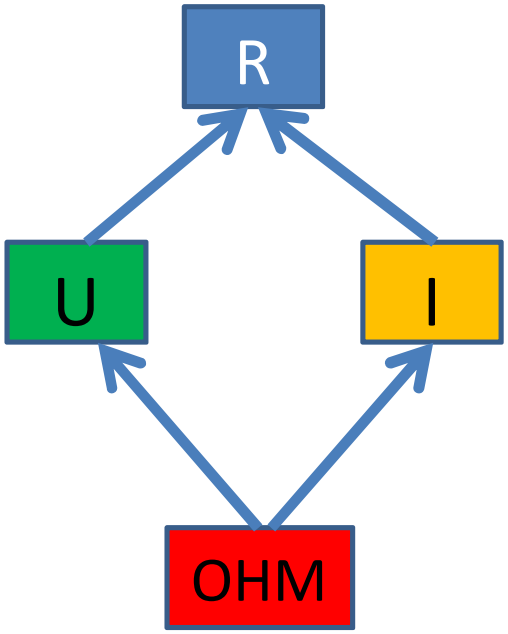
```
	public:
		Voltage(double u,int r):Resistor(r)//call constructor for class Resistor
		//because it does not implicitly get inherited by derivate classes
		{
			U=u;//constructor with 2 parameters, one inherited (R) and another
				//one who's value is attributed here in the constructor (U)
		}

		virtual double ComputeCurrent()
		{//compute value of Current from Voltage (set in this derivate class)
		//and by the Resistor (inheritance)
			I=U/R;
		}

		virtual double GetI()
		{//display computed value
			cout<<I<<endl;
		}
};//end of class with semicolon character ; !

class Current:virtual public Resistor
{//upon creating Current objects (first derivate class), due to the
//keyword virtual
//we won't create a copy of the base class and it will be referenced instead.
//all objects of type Current will reference class Resistor without
//creating copies in memory for each of these objects
	protected:
		double U;
		double I;// protected data accessible within derivate classes
				// but not outside them
```

```
	public:
		Current(double i,int r):Resistor(r)//call constructor of Resistor class
		//as it is not implicitly inherited in derived classes
		{
			I=i; //constructor with 2 parameters, one inherited (R)
			// and another one who's value is attributed here in the
			constructor (I)


		}


		virtual double ComputeVoltage()
		{
			U=I*R;//compute Voltage from Current
			//(set in this derived class) and from Resistor (inherited)
		}


		virtual double GetU()
		{//display computed value
			cout<<U<<endl;
		}
};	//end of class with semicolon character ; !


class OHMsLaw:public Voltage, public Current
{//class OHMsLaw inherits 2 classes, so multiple inheritance from both
//Voltage class and Current class
	public:
		OHMsLaw(double i, double u, int r):Current(i,r), Voltage(u,r),Resistor(r)
		 //constructor with 3 parameters, all inherited
		{
			cout <<"passing through constructor with 3 parameters "<<endl;
```

```
                    //this constructor only displays a message, does nothing else
                }
}; //end of class with semicolon character ; !


int main()
{
        OHMsLaw L1(0.01, 10, 1000);//create an multiple inheritance object
        L1.ComputeVoltage();//call Voltage compute function
        L1.ComputeCurrent();//call Current compute function
        L1.GetR();//display R
        L1.GetU();//display U
        L1.GetI();//display I
}
```

If we want to build the tree to describe the multiple inheritance from this program it would look like this:

Case 1 (having written virtual on inheritance). On creating object L1 we create a copy of class OHM then a copy of classes U and I, followed by a copy of R, which U and I use via reference due to the `virtual` keyword.



Case 2 (WITHOUT the keyword virtual on inheritance). Upon creating object L1 we create a copy of the class OHM and then a copy of classes U and I, and then for each copy of U and I a separate copy of R (so we will end up with 2 copies of R) that the compiler will consider ambiguous as there will be 2 copies of GetR in memory. This code would generate the error *"[Error] request for member 'GetR' is ambiguous*".