Academic Year: 2020

Semester: 2nd

Course Code: CS-241L

Course Title: Object Oriented Programming

CS-241L Object Oriented Programming Lab 9

Type of Lab: Open Ended **Weightage: 20%**

CLO: CLO 1 + CLO 2.

Student understand polymorphism, early and late binding, function overriding, abstract and concrete classes	**Cognitive/Understanding**	CLO1, CLO2	Rubric A

Rubric A: Cognitive Domain

Evaluation Method: GA shall evaluate the students for Question according to following rubrics.

CLO	**0**	**1**	**2**	**3**	**4**	**5**
CLO1, CLO2	Unable to understand and implement	Student understand poly morphism conceptually	Student try to implement poly morphism partially	Student implemented poly morphism completely	Understand and implemented half problem sets	Understand and implemented complete problem sets

Lab 9

BS-Computer Science

Object Oriented Programming

Target: Polymorphism, Early and late binding, Function overriding, abstract class vs Concrete Classes and Interface.

Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee etc. So, a same person can have different behaviour in different situations. This is called polymorphism.

Before going to virtual function, let's first have a look at **early binding and late binding.**

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

Early Binding

In early binding, (compile time binding, static linkage, static resolution) the compiler matches the function call with the correct function definition at compile time. It is also known as **Static Binding** or **Compile-time Binding**. By default, the compiler goes to the function definition which has been called during compile time. So, all the function calls you have studied till now are due to early binding.

You have learned about function overriding in which the base and derived classes have functions with the same name, parameters and return type. In that case also, early binding takes place.

In function overriding, we called the function with the objects of the classes. Now let's try to write the same example but this time

calling the functions with the pointer to the base class i.e., refernce to the base class' object.

Example:

```
#include <iostream>

using namespace std;

class Animals
{
	public:
		void sound()
		{
			cout << "This is parent class" << endl;
		}
};

class Dogs : public Animals
{
	public:
		void sound()
		{
			cout << "Dogs bark" << endl;
		}
};

int main()
{
	Animals *a;
	Dogs d;
	a= &d;
	a -> sound();   //  early binding
	return 0;
}
```

Output:

Now in this example, we created a pointer a to the parent class Animals. Then by writing a= &d , the pointer 'a' started referring to the object d of the class Dogs. a -> sound(); - On calling the function sound() which is present in both the classes by the pointer 'a', the function of the parent class got called, even if the pointer is referring to the object of the class Dogs.

This is due to Early Binding. We know that a is a pointer of the parent class referring to the object of the child class. Since early binding takes place at compile-time, therefore when the compiler saw that a is a pointer of the parent class, it matched the call with the 'sound()' function of the parent class without considering which object the pointer is referring to.

Late Binding

In the case of late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as **Dynamic Binding** or **Runtime Binding**.

In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

By default, early binding takes place. So if by any means we tell the compiler to perform late binding, then the problem in the previous example can be solved.

This can be achieved by declaring a **virtual function**.

Virtual function

Virtual Function is a member function of the base class which is overridden in the derived class. The compiler performs **late binding** on this function.

To make a function virtual, we write the keyword **virtual** before the function definition.

Example:

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

class Animals
{
	public:
		virtual void sound()
		{
			cout << "This is parent class" << endl;
		}
};

class Dogs : public Animals
{
	public:
		void sound()
		{
			cout << "Dogs bark" << endl;
		}
};

int main()
{
	Animals *a;
	Dogs d;
	a= &d;
	a -> sound();
	return 0;
}
```

Output:

Since the function sound() of the base class is made virtual, the compiler now performs late binding for this function. Now, the

function call will be matched to the function definition at runtime. Since the compiler now identifies pointer a as referring to the object 'd' of the derived class Dogs, it will call the sound() function of the class Dogs.

Note:

If we declare a member function in a base class as a virtual function, then that function automatically becomes virtual in all of its derived classes.

If we make any function inside a base class virtual, then that function becomes virtual in all its derived classes. This means that we don't need to declare that function as virtual separately in its derived classes.

Note:

We can also call private function of derived class from a base class pointer by declaring that function in the base class as virtual.

Compiler checks if the members of a class are private, public or protected only at compile time and not at runtime. Since our function is being called at runtime, so we can call any type of function, private or public as shown in the following example.

Example

```
#include <iostream>

using namespace std;

class Animals

{

        public:

                virtual void sound()

                {

                        cout << "This is parent class" << endl;
```

```
        }
};

class Dogs : public Animals
{
    private:
        virtual void sound()
        {
            cout << "Dogs bark" << endl;
        }
};

int main()
{
    Animals *a;
    Dogs b;
    a = &b;
    a->sound();
    return 0;
}
```

Output:

Since the same function (virtual function) having different definitions in different classes is called depending on the type of object that calls the function, this is also a part of **Polymorphism**.

Pure Virtual Function

Pure virtual function is a virtual function which has no definition. Pure virtual functions are also called **abstract functions**.

To create a pure virtual function, we assign a value **0** to the function as follows.

virtual void sound() = 0;

Here sound() is a pure virtual area.

Abstract Class

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

An **abstract class** is a class whose instances (objects) can't be made. We can only make objects of its subclass (if they are not abstract). Abstract class is also known as **abstract base class**.

An abstract class has at least one abstract function (pure virtual function).

Abstract classes

- Sole purpose is to provide a base class for other classes
- No objects of an abstract base class can be instantiated
- Can have pointers and references
- A derived class of an abstract base class remains abstract unless the implementation of all the pure virtual functions is not provided.

Concrete Class: classes (Derived) that can instantiate objects

Let's look at an example of abstract class.

Suppose there are some employees working in a firm. The firm hires only two types of employees- either driver or developer. Now, you have to develop a software to store information about them.

So, here is an idea - There is no need to make objects of employee class. We will make objects to only driver or developer. Also, both must have some salary. So, there must be a common function to know about salary.

This need will be best accomplished with **abstract class**.

So, we can make 'Employee' an abstract class and 'Developer' and 'Driver' its subclasses.

Example:

```
#include <iostream>

using namespace std;

class Employee                  //  abstract base class
{
        virtual int getSalary() = 0;    // pure virtual function
};

class Developer : public Employee
{
        int salary;
        public:
                Developer(int s)
                {
                        salary = s;
                }
                int getSalary()
                {
                        return salary;
```

```
        }
};

class Driver : public Employee
{
    int salary;
    public:
        Driver(int t)
        {
            salary = t;
        }
        int getSalary()
        {
            return salary;
        }
};

int main()
{
    Developer d1(5000);
    Driver d2(3000);
    int a, b;
    a = d1.getSalary();
    b = d2.getSalary();
    cout << "Salary of Developer : " << a << endl;
    cout << "Salary of Driver : " << b << endl;
    return 0;
}
```

Output:

The **getSalary()** function in the class **Employee** is a pure virtual function. Since the Employee class contains this pure virtual function, therefore it is an **abstract base class**. Since the abstract function is defined in the subclasses, therefore the function

'getSalary()' is defined in both the subclasses of the class Employee.

You must have understood the rest of the code.

Note:

Subclasses of an abstract base class must define the abstract method, otherwise, they will also become abstract classes.

Note:

In an abstract class, we can also have other functions and variables apart from pure virtual function.

Let's see one more example of abstract base class.

Example:

```
#include <iostream>

using namespace std;

class Animals
{
	public:
		virtual void sound() = 0;
};

class Dogs
{
	public:
		void sound()
		{
			cout << "Dogs bark" << endl;
		}
};
```

```
class Cats
{
	public:
		void sound()
		{
			cout << "Cats meow" << endl;
		}
};

class Pigs
{
	public:
		void sound()
		{
			cout << "Pigs snort" << endl;
		}
};

int main()
{
	Dogs d;
	Cats c;
	Pigs p;
	d.sound();
	c.sound();
	p.sound();
	return 0;
}
```

Output:

```
Dogs bark
Cats meow
Pigs snort
```

Interface

Interface or **Interface class** is a class which is the same as abstract class with a difference that all its functions are pure virtual and it has no member variables. Its derived classes must implement each of its virtual functions i.e., provide definition to each of the pure virtual functions of the base class.

Like an abstract class, we can't create objects of an interface.

We can also say that interface is an abstract class with no member variables and all its member functions pure virtual.

Name of an interface class often begins with the letter I.

Let's see an example of it.

```
class IShape
{
   public:
      virtual getArea() = 0;
      virtual getPerimeter() = 0;
};
```

IShape is an interface because it contains only pure virtual functions.

Example:

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

class IShape
{
        public:
                virtual int getArea() = 0;
                virtual int getPerimeter() = 0;
};

class Rectangle : public IShape
{
```

```
	int length;
	int breadth;
	public:
		Rectangle(int l, int b)
		{
			length = l;
			breadth = b;
		}
		int getArea()
		{
			return length * breadth;
		}
		int getPerimeter()
		{
			return 2*(length + breadth);
		}
};

class Square : public IShape
{
	int side;
	public:
		Square(int a)
		{
			side = a;
		}
		int getArea()
		{
			return side * side;
		}
		int getPerimeter()
		{
			return 4 * side;
		}
```

```
};

int main()
{
	Rectangle rt(7, 4);
	Square s(4);
	cout << "Rectangle :" << endl;
	cout << "Area : " << rt.getArea() << " Perimeter : " << rt.getPerimeter() << endl;
	cout << "Square :" << endl;
	cout << "Area : " << s.getArea() << " Perimeter : " << s.getPerimeter() << endl;
	return 0;
}
```

Output:

So we just saw that IShape is an interface with two pure virtual functions. These virtual functions are implemented (defined) in its subclasses Rectangle and Square according to their requirements.

So, an interface is just an abstract class with all pure virtual methods.

Problem Set 1:

Create a class called **Distance** containing two members feet and inches. This class represents distance
measured in feets and inches. For this class, provide the following functions:

1. **A no-argument** constructor that initializes the data members to some fixed values.
2. **A 2-argument** constructor to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.
3. Provide the following operators as friends:

4. **operator+** to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one feet. Make sure that the result of addition doesn't violate this rule.
5. **operator+=** for addition of two distances.
6. **operator >:** should return a variable of type bool to indicate whether 1st distance is greater than 2nd or not.
7. **operator <:** should return a variable of type bool to indicate whether 1st distance is smaller than 2nd or not.
8. **operator >=:** should return a variable of type bool to indicate whether 1st distance is greater than or equal to 2nd or not.
9. **operator <=:** should return a variable of type bool to indicate whether 1st distance is smaller than or equal to 2nd or not.
10. **operator==:** should return a variable of type bool to indicate whether 1st Distance is equal to the 2nd distance or not.
11. **operator!=:** should a true value if both the distances are not equal and return a false if both are equal.

Problem Set 2:

Create a class **Employee** that has a field for storing the complete name of employee (first and last name), a field for storing the IDentification number of employee, and another field for storing his salary.

Provide

1. **no-argument constructor** for initializing the fields to default values
2. **3-argument constructor** for initializing the fields to values sent from outside. Use strcpy function to copy one string into another.
3. a setter function (mutator) that sets the values of these fields by getting input from user.

4. an accessor function to display the values of the fields.
5. Derive three classes from this employee class: **Manager**, **Scientist**, and **Laborer**. **Manager** class has an additional data member of # of subordinates. **Scientist** class contains additional information about # of publications. **Laborer** class is just similar to the employee class. It has no additional capabilities. Derive a class **foreman** from the **Laborer** class that has an additional data member for storing the percentage of quotas met by a **foreman**. Provide appropriate no-argument and n-argument constructors for all the classes. Provide the overridden getter and setter functions here too to input and output all the fields. Determine whether public, private, or protected inheritance should be used.

Problem Set 3:

1. Create a class **Student** that contains information about a student's name, semester, roll no, and date of admission. To store the date of admission, again reuse the date class that you have already developed. Determine whether you should use inheritance or composition.

❖ **Provide**

2. **no-argument constructor** for initializing the values of data members to some defaults.
3. **4-argument constructor** to initialize the data members sent from the calling function at the time of creation of an object(date should be sent from outside in the form of a date object).
4. An **input** function for setting the status of a student.
5. A **display** function to display all the attributes of a student.
6. Derive a class **Undergraduate** from **Student** class of activity 2 that contains some additional information. This information is about the semester gpa of a student and the

credit points earned per semester. To store this data, provide a 2D array (2x8 array since at maximum there are 8 semesters for an undergraduate program). One dimension of the array should hold information about the SGPA of each

7. semester so far and the other dimension should hold the corresponding credit points earned in that semester.
8. a) Create a no-argument and a 5-argument constructor for data member initialization.
9. b) Provide overridden functions for getting and setting the data members.
10. Provide another function to calculate the CGPA of student on the basis of the information provided by the 2D array.
11. Derive a class **Graduate** from **Student** class that also has the same additional information as the **Undergraduate** class but in this case, the array is 2x4 since at maximum there are four semesters in a Graduate program. There are two additional data members: one to store the title of the last degree held and another to store the area of specialization in graduate program. Provide appropriate constructors and overridden member functions.

Problem Set 4:

1. Define an abstract base class shape that includes protected data members for area and volume of a shape, public methods for computing area and volume of a shape (make the functions virtual), and a display function to display the information about an object. Make this class abstract.

2. Derive a concrete class point from the shape class. This point class contains two protected data members that hold the position of point. Provide no-argument and 2-argument constructors. Override the appropriate functions of base class.

3. Derive a class Circle publicly from the point class. This class has a protected data member of radius. Provide a no-argument constructor to initialize the fields to some fixed values. Provide a 3-argument constructor to initialize the data members of Circle class to the values sent from outside. Override the methods of base class as required.

4. Derive another class Cylinder from the Circle class. Provide a protected data member for height of cylinder. Provide a no-argument constructor for initializing the data members to default values. Provide a 4-argument constructor to initialize x- and y-coordinates, radius, and height of cylinder. Override the methods of base class.

5. Write a driver program to check the polymorphic behaviour of this class.