Lab 14

Items	Description
Course Title	Object Oriented Programming
Lab Title	Classes (Polymorphism + Abstract classes)
Duration	3 Hours
Tools	Eclipse/ C++
Objective	To get familiar with the use of different concepts in classes in c++

Polymorphism

Definition

Polymorphism is an OOP concept that allows one interface to be used for different types of objects. In C++, polymorphism comes in two forms:

1. **Compile-time Polymorphism** (Static): Achieved through function overloading and operator overloading.
2. **Run-time Polymorphism** (Dynamic): Achieved through inheritance and virtual functions.

What Are Virtual Functions?

A virtual function is a member function in a base class that you expect to be overridden in derived classes. When a base class reference or pointer is used to call a function, the decision about which function to call is made **at runtime** (dynamic binding) rather than compile time (static binding).

Virtual functions allow achieving **runtime polymorphism**, enabling dynamic behavior based on the actual object type.

Why Use Virtual Functions?

1. **Dynamic Behavior**:
   - Virtual functions enable derived classes to define specific behaviors for functions declared in the base class.
2. **Achieve Runtime Polymorphism**:
   - Enables calling functions on base class references or pointers and executing the appropriate derived class's implementation.
3. **Flexible and Extensible Design**:
   - Allows adding new functionality by creating new derived classes without modifying existing code.

Compile-time Polymorphism

```
#include <iostream>
using namespace std;
class Animal {
public:
    void sound() { // Not virtual
        cout << "Animal makes a sound" << endl;
    }
};
class Dog : public Animal {
public:
    void sound() { // Overrides the base class function
        cout << "Dog barks" << endl;
    }
};
```

```
int main() {
    Dog dog;
    dog.sound(); // Calls Dog's version -> Output: Dog barks
    Animal animal = dog; // Object slicing occurs
    animal.sound(); // Calls Animal's version -> Output: Animal
makes a sound
    return 0;}
```

Run-time Polymorphism

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() { // Virtual function
        cout << "Animal speaks generically." << endl;
    }
};
class Dog : public Animal {
public:
    void speak() override { // Overrides the base class function
        cout << "Dog barks." << endl;
    }
};
```

```
int main() {
    Animal* animalPtr = new Dog(); // Base class pointer to derived class object
    animalPtr->speak(); // Calls Dog's speak, due to dynamic binding
    delete animalPtr;
    return 0;
}
```

Key Points About Virtual Functions

1. **Declared Using `virtual`**:
   - Defined in the base class using the `virtual` keyword.
2. **Overridden Using `override` (Optional)**:
   - C++11 introduced `override` to explicitly mark an overriding function in the derived class, making the intent clear and catching potential errors.
3. **Polymorphism**:
   - Enables a base class pointer or reference to call the appropriate derived class function at runtime.

```
#include <iostream>

using namespace std;

class Shape {

public:

    virtual void draw() { // Virtual function
```

```
        cout << "Drawing a generic shape." << endl;
    }
};
class Circle : public Shape {
public:
    void draw() override { // Override the base class method
        cout << "Drawing a Circle." << endl;
    }
};
class Rectangle : public Shape {
public:
    void draw() override { // Override the base class method
        cout << "Drawing a Rectangle." << endl;
    }
};
```

National University of computer and emerging sciences

```
void render(Shape* shape) {
    shape->draw(); // Calls the appropriate draw() based on the object type
}
int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    render(shape1); // Calls Circle's draw
    render(shape2); // Calls Rectangle's draw

    delete shape1;
    delete shape2;
    return 0;
}
```

Binding

Binding:

Binding refers to associating a function call with a function definition. In C++, binding can be of two types:

1. **Static Binding (Early Binding)**:
    - Happens at compile time.
    - Used with normal function calls.
    - Achieved using regular or overloaded functions.
2. **Dynamic Binding (Late Binding)**:
    - Happens at runtime.
    - Used when a base class pointer or reference calls a function defined in a derived class.
    - Achieved using `virtual` functions.

Static Binding (Early Binding):

```
#include <iostream>

using namespace std;

class Animal {

public:

    void speak() {

        cout << "Animal speaks in its own way." << endl;

    }

};
```

```cpp
int main() {
    Animal a;
    a.speak(); // Static binding: Resolved at compile time
    return 0;
}
```

Dynamic Binding (Late Binding):

```cpp
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() { // Virtual function for dynamic binding
        cout << "Animal speaks." << endl;
    }
};
```

National University of computer and emerging sciences

```
class Dog : public Animal {
public:
    void speak() override { // Overrides the base class function
        cout << "Dog barks." << endl;
    }
};
int main() {
    Animal* a = new Dog(); // Base class pointer pointing to derived class
    a->speak(); // Dynamic binding: Resolved at runtime
    delete a;
    return 0;
}
```

Abstract Classes and Pure Virtual Functions

Abstract Class:

An abstract class is a class that cannot be instantiated. It is used as a blueprint for derived classes and usually contains at least one pure virtual function.

Pure Virtual Function:

A pure virtual function is a function declared in a base class but requires derived classes to implement it. It is defined by assigning `= 0` to the virtual function.

Key Points:

- A class with at least one pure virtual function is an abstract class.
- Abstract classes act as interfaces.

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

class Shape { // Abstract class
public:
    virtual void draw() const = 0; // Pure virtual function
    virtual double area() const = 0; // Another pure virtual function
};
```

National University
of computer and emerging sciences

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}

    void draw() const override { // Implements pure virtual
function
        cout << "Drawing a Circle." << endl;
    }

    double area() const override { // Implements pure virtual
function
        return 3.14159 * radius * radius;
    }
};


class Rectangle : public Shape {
    double length, width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
```

```
    void draw() const override { // Implements pure virtual
function
        cout << "Drawing a Rectangle." << endl;
    }
    double area() const override { // Implements pure virtual
function
        return length * width;
    }
};
int main() {
    Shape* s1 = new Circle(5.0); // Pointer to abstract class
    Shape* s2 = new Rectangle(4.0, 6.0);

    s1->draw();
    cout << "Circle Area: " << s1->area() << endl;

    s2->draw();
    cout << "Rectangle Area: " << s2->area() << endl;

    delete s1;
    delete s2;
    return 0;
}
```

National University of computer and emerging sciences

Header File Implementation

Shape.h

```
#ifndef SHAPE_H
#define SHAPE_H

#include <iostream>
using namespace std;
class Shape {
public:
    // Pure virtual function for area
    virtual double area() const = 0;

    // Pure virtual function for displaying the shape
    virtual void display() const = 0;

    // Virtual destructor
    virtual ~Shape() {
        cout << "Shape destroyed" << endl;
    }
};
#endif // SHAPE_H
```

National University of computer and emerging sciences

Circle.h

```cpp
#ifndef CIRCLE_H
#define CIRCLE_H
#include "Shape.h"

class Circle : public Shape {
private:
    double radius;
public:
    // Constructor
    Circle(double r);

    // Override area() function
    double area() const override;

    // Override display() function
    void display() const override;

    // Destructor
    ~Circle() override;
};
#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <cmath> // for M_PI

// Constructor
Circle::Circle(double r) : radius(r) {}

// Override area() function
double Circle::area() const {
    return M_PI * radius * radius;
}

// Override display() function
void Circle::display() const {
    cout << "Circle with radius: " << radius << " has area: " <<
area() << endl;
}
// Destructor
Circle::~Circle() {
    cout << "Circle destroyed" << endl;
}
```

National University of computer and emerging sciences

Rectangle.h

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "Shape.h"

class Rectangle : public Shape {
private:
    double length, width;
public:
    // Constructor
    Rectangle(double l, double w);

    // Override area() function
    double area() const override;

    // Override display() function
    void display() const override;
    // Destructor
    ~Rectangle() override;
};
#endif //  RECTANGLE_H
```

National University of computer and emerging sciences

Rectangle.cpp

```
#include "Rectangle.h"

// Constructor
Rectangle::Rectangle(double l, double w) : length(l), width(w) {}

// Override area() function
double Rectangle::area() const {
    return length * width;
}

// Override display() function
void Rectangle::display() const {
    cout << "Rectangle with length: " << length
         << " and width: " << width
         << " has area: " << area() << endl;
}

// Destructor
Rectangle::~Rectangle() {
    cout << "Rectangle destroyed" << endl;
}
```

Main.cpp

```
#include "Circle.h"
#include "Rectangle.h"
#include <vector>
int main() {
    // Create a vector of Shape pointers
    vector<Shape*> shapes;
    // Add Circle and Rectangle objects
    shapes.push_back(new Circle(5.0));      // Circle with radius 5
    shapes.push_back(new Rectangle(4.0, 6.0));
    // Display information and calculate areas using
polymorphism
    for (Shape* shape : shapes) {
        shape->display();
    }

    // Clean up memory
    for (Shape* shape : shapes) {
        delete shape; // Calls destructors
    }
    return 0;
}
```

National University
of computer and emerging sciences

Lab Task

Task 1: Virtual Functions and Polymorphism (Single Inheritance)

1. **Objective**:
   - Implement runtime polymorphism using virtual functions with single inheritance.
2. **Task**:
   - Create a base class `Shape` with a virtual function `area()` and a non-virtual function `printArea()`.
   - Create derived classes `Circle` and `Rectangle`:
     - `Circle` has a radius and calculates area as $\pi \times r2$
     - `Rectangle` has length and width and calculates area as length×width\text{length} \times \text{width}length×width.
   - Write a function that takes a pointer to `Shape` and calls `printArea()`.
3. **Expected Output**:
   - Show correct area calculations for `Circle` and `Rectangle` objects using a `Shape` pointer.

Task 2: Virtual Functions in Multiple Inheritance

1. **Objective**:
   - Explore virtual functions in a multiple inheritance scenario.
2. **Task**:
   - Create two base classes:
     - `MusicInstrument` with a virtual function `play()`.
     - `Percussion` with a virtual function `beat()`.
   - Create a derived class `Drum`:
     - Override `play()` to print "Drum is being played."
     - Override `beat()` to print "Drum is being beaten."
   - Write a function to take `MusicInstrument` and `Percussion` pointers to a `Drum` object and call both `play()` and `beat()`.
3. **Expected Output**:
   - Show that the `Drum` functions are correctly called through base class pointers.

National University
of computer and emerging sciences

Task 3: Abstract Classes and Pure Virtual Functions

1. **Objective**:
   - Understand abstract classes and pure virtual functions.
2. **Task**:
   - Create an abstract class `Vehicle` with a pure virtual function `move()` and a non-pure virtual function `fuelType()`.
   - Create derived classes `Car` and `Bike`:
     - Override `move()` to print "Car moves on four wheels" and "Bike moves on two wheels".
     - Override `fuelType()` to specify fuel types for both.
   - Write a function to call `move()` and `fuelType()` for different `Vehicle` objects.
3. **Expected Output**:
   - Demonstrate the use of abstract classes by creating instances of `Car` and `Bike`.

Task 4: Polymorphism in a Real-World Scenario

1. **Objective**:
   - Implement polymorphism in a practical scenario involving multiple derived classes.
2. **Task**:
   - Create a base class `Animal` with a virtual function `makeSound()`.
   - Create derived classes `Dog`, `Cat`, and `Cow`:
     - `Dog::makeSound()` prints "Dog barks."
     - `Cat::makeSound()` prints "Cat meows."
     - `Cow::makeSound()` prints "Cow moos."
   - Write a function that takes a vector of `Animal*` objects, iterates through them, and calls `makeSound()`.
3. **Expected Output**:
   - Print the correct sound for each animal in the list using runtime polymorphism.