# Lab 2: Inheritance, Polymorphism, and Abstract Classes in C++

## Objective:

Explore and implement the OOP principles of inheritance, polymorphism, and abstract classes in C++. These concepts facilitate the creation of a hierarchical class structure that promotes code reuse, flexibility, and scalability.

## IDE: Visual Studio 2022

## Introduction to Concepts:

**Inheritance** allows a new class (derived class) to take on the properties and methods of an existing class (base class). This relationship helps in code reuse and is a fundamental concept in OOP.

#include <iostream>

using namespace std;

class Animal { // Base class

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

class Dog : public Animal { // Derived class

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog myDog;

myDog.eat(); // Inherited method

myDog.bark(); // Own method

return 0;

}

**Polymorphism** is the ability of a function to do different things based on the object that it is acting upon. In C++, polymorphism is achieved mainly through the use of virtual functions and function overriding.

#include <iostream>

using namespace std;

class Animal {

public:

virtual void sound() {

cout << "Some sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Woof" << endl;

}

};

int main() {

Dog myDog;

Animal& refAnimal = myDog;

refAnimal.sound(); // Outputs: Woof

return 0;

}

**Virtual functions** are member functions which are declared in a base class and are overridden by a derived class. They allow for runtime polymorphism.

#include <iostream>

using namespace std;

class Base {

public:

virtual void print() {

cout << "Base function" << endl;

}

};

class Derived : public Base {

public:

void print() override {

cout << "Derived function" << endl;

}

};

int main() {

Derived myDerived;

myDerived.print(); // Calls Derived's print function, outputs: Derived function

return 0;

}

**Pure Virtual functions** is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declarationare member functions which are declared in a base class and are overridden by a derived class. They allow for runtime polymorphism.

#include <iostream>

using namespace std;

class AbstractClass {

public:

virtual void pureVirtualFunction() = 0; // Pure virtual function

};

class ConcreteClass : public AbstractClass {

public:

void pureVirtualFunction() override {

cout << "Implemented pure virtual function" << endl;

}

};

int main() {

ConcreteClass myConcrete;

myConcrete.pureVirtualFunction(); // Calls the implemented function

return 0;

}

**Abstract Classes** are classes that cannot be instantiated and are designed to be specifically used as base classes. An abstract class typically includes at least one pure virtual function.

#include <iostream>

using namespace std;

class Shape { // Abstract class

public:

virtual double area() const = 0; // Pure virtual function

};

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return 3.14159 \* radius \* radius;

}

};

int main() {

Circle myCircle(5);

cout << "Area of circle: " << myCircle.area() << endl; // Outputs the area

return 0;

}

## Scenario

Imagine you are developing a simple graphics application where you need to manage different shapes like circles, rectangles, and triangles. Each shape has its method to calculate the area, but you want to treat all shapes uniformly in your application. This scenario is perfect for applying inheritance, polymorphism, and abstract classes.

## Sample Code Demonstrations and Expected Outputs:

To illustrate the application of inheritance, polymorphism, abstract classes, virtual functions, and pure virtual functions in a unified scenario, let's develop a simple graphics application to manage shapes like circles, rectangles, and triangles.

### Step 1: Defining the Abstract Base Class

Start by defining an abstract base class Shape with a pure virtual function for calculating the area. This class will serve as a template for all specific shape classes.

**- Code:**

#include <iostream>

using namespace std;

class Shape {

public:

virtual double area() const = 0; // Pure virtual function making Shape an abstract class.

};

### Step 2: Creating Derived Classes

Create derived classes for Circle, Rectangle, and Triangle, each overriding the area method to calculate their specific area.

**- Circle:**

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return 3.14159 \* radius \* radius;

}

};

**- Rectangle:**

class Rectangle : public Shape {

private:

double width, height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

double area() const override {

return width \* height;

}

};

**- Triangle:**

class Triangle : public Shape {

private:

double base, height;

public:

Triangle(double b, double h) : base(b), height(h) {}

double area() const override {

return 0.5 \* base \* height;

}

};

### Step 3: Demonstrating Polymorphism

Show how polymorphism allows treating all shapes uniformly in your application, even though each has a different method of calculating the area. Use a function that accepts a reference to the Shape base class to demonstrate this.

**- Code:**

void printArea(const Shape& shape) {

cout << "Area: " << shape.area() << endl;

}

### Step 4: Implementing the Scenario

Combine all the concepts in a main function to demonstrate how the shapes can be managed uniformly using polymorphism and abstract classes.

**- Code and Expected Output:**

int main() {

Circle circle(5);

Rectangle rectangle(10, 5);

Triangle triangle(10, 5);

printArea(circle); // Outputs: Area: 78.5398

printArea(rectangle); // Outputs: Area: 50

printArea(triangle); // Outputs: Area: 25

return 0;

}

This step-by-step example demonstrates the principles of OOP—inheritance, polymorphism, and the use of abstract classes. By abstracting the concept of a shape and leveraging polymorphism, the application can treat all shapes uniformly, simplifying the management of different types of shapes and their interactions.

## In-Lab Tasks:

### Managing a Zoo Animal Hierarchy

Imagine you are developing a system for a zoo to manage different types of animals. Each animal has its own way of communicating, but you want to treat all animals uniformly in your system. This scenario is perfect for applying inheritance, polymorphism, and abstract classes.

1. **Basic Inheritance and Abstract Class**

Define an abstract class Animal with a pure virtual function makeSound(). Create derived classes Lion, Elephant, and Monkey that override the makeSound method to produce a sound unique to each animal type.

1. **Polymorphism in Action**

Create a function hearSound that accepts a reference to an Animal and invokes the makeSound method. Demonstrate polymorphism by creating instances of Lion, Elephant, and Monkey, and use the hearSound function to hear the sounds of different animals.

1. **Adding a New Animal Class**

Extend the zoo system by adding a new derived class Bird that also overrides the makeSound method. Ensure that your system can handle the new Bird class without any changes to the hearSound function, demonstrating the power of polymorphism and abstract classes.

## Extra Tasks for Home:

1. **Implementing a Simple Ecosystem**

Scenario: You are tasked with creating a simple ecosystem simulation where different types of plants grow at different rates. The goal is to calculate the growth over time for each plant type.

Define an abstract base class Plant with a pure virtual function grow(), which calculates and updates the height of the plant based on its growth rate. Create derived classes Flower, Tree, and Bush, each with a specific growth rate.

**- Partially Implemented Code:**

class Plant {

public:

virtual void grow() = 0; // Pure virtual function

virtual void displayHeight() const = 0;

};

class Flower : public Plant {

private:

double height; // in centimeters

double growthRate; // cm per day

public:

Flower(double initialHeight) : height(initialHeight), growthRate(0.5) {}

void grow() override {

height += growthRate;

}

void displayHeight() const override {

cout << "Flower height: " << height << " cm" << endl;

}

// TODO: Implement the Tree and Bush classes based on the Flower class template.

};

Complete the implementation by adding the Tree and Bush classes, each with their unique growth rates (e.g., Tree: 0.2 cm/day, Bush: 0.7 cm/day). Implement a simulation in main that creates instances of each plant type, simulates 30 days of growth, and displays the final height of each plant.

## Conclusion:

This manual guides you through understanding and applying inheritance, polymorphism, and abstract class concepts in C++. By completing the scenarios and tasks, students will gain practical experience in implementing fundamental OOP principles in C++.