

**Adan Bin Waheed**

**45426**

**OOP – Assignment No.4**

**THEORY**

**Explain what polymorphism is and how it relates to object-oriented programming.**

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios. For example:

The + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition. And when we use the + operator with strings, it performs string concatenation.

**What is the difference between static and dynamic polymorphism?**

* Static polymorphism is a type of polymorphism that collects the information to call a method during compile time
* Dynamic polymorphism is a type of polymorphism that collects information to call a method at run time.

**Describe the two types of polymorphism in C++.**

1. **Compile Time Polymorphism**
2. **Runtime Polymorphism**

**Compile Time Polymorphism:**

In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding.Function overloading and operator overloading is the type of Compile time polymorphism.

**Runtime Polymorphism:**

In a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.Function overriding is a part of runtime polymorphism.

**What is a virtual function? Explain why it is used.**

A virtual function in C++ is a base class member function that you can redefine in

a derived class to achieve polymorphism. You can declare the function in the base

class using the virtual keyword.A virtual function is used in C++ helps ensure you call the correct function via a

reference or pointer.

**Can a class have both virtual and non-virtual functions? Explain your answer.**

Yes, a class can have both virtual and non-virtual functions.

A virtual function is a member function in the base class that we expect to redefine

in derived classes. Basically, a virtual function is used in the base class in order to

ensure that the function is overridden.

Non-virtual functions are regular member functions that are not declared as virtual

in the base class and cannot be overridden by the derived classes.

**IMPLEMENTATION**

**Write a C++ program that demonstrates the concept of function overloading.**

#include <iostream>

using namespace std;

void sum(int a, int b)

{

cout << "sum = " << (a + b);

}

int main()

{

sum(14, 6);

return 0;

}

**Write a C++ program that demonstrates the concept of operator overloading.**

#include <iostream>

Using namespace std;

class Distance {

private:

int inches;

public:

Distance(int inches = 0) : inches(inches) {}

// Overloading the '+' operator

Distance operator+(const Distance& other) const {

Distance result;

result.inches = inches + other.inches;

return result;

}

// Overloading the '-' operator

Distance operator-(const Distance& other) const {

Distance result;

result.inches = inches - other.inches;

return result;

}

// Overloading the '<<' operator for output

friend ostream& operator<<(ostream& os, const Distance& distance) {

os << distance.inches << " inches";

return os;

} };

int main() {

Distance dist1(5);

Distance dist2(3);

Distance sum = dist1 + dist2;

Distance difference = dist1 - dist2;

cout << "Distance 1: " << dist1 <<endl;

cout << "Distance 2: " << dist2 <<endl;

cout << "Sum: " << sum <<endl;

cout << "Difference: " << difference <<endl;

return 0;

}

**Write a C++ program that demonstrates the concept of runtime polymorphism using virtual functions.**

#include<iostream>

using std::endl;

using std::cout;

class Pen{

public:

virtual void colour();

};

class Eraser:public Pen{

public:

void colour();

};

void Pen::colour()

{

cout<<"BLUE COLOUR"<<endl;

}

void Eraser::colour()

{

cout<<"ERASER STYLE"<<endl;

}

int main(){

Pen \*pEraser =new Pen;

pEraser->colour();

delete pEraser;

pEraser= new Eraser;pEraser->colour(); delete pEraser; return 0;

}

**Write a C++ program that demonstrates the concept of compile-time**

**polymorphism using templates.**

#include <iostream>

using namespace std;

class Value {

public :

void fun(int a, int b) {

cout << "The value of a is" << a<<endl;

cout << "The value of b is" << b<<endl;

} };

int main() {

value obj;

cout<< "Two int values"<<endl;

obj.fun(5, 1);

return 0;

}

**Application**

1. **Write a C++ program that uses polymorphism to create a hierarchy of shapes. The program should have a base class called `Shape` and derived classes for different types of shapes (e.g. `Circle`, `Rectangle`, `Triangle`). Each derived class should implement a function called `area()` that calculates the area of the shape. The program should allow the user to create objects of different shapes and calculate their areas using polymorphism.**

#include <iostream>

using namespace std;

class Shape {

public:

virtual double area() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return 3.14159 \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

double area() const override {

return width \* height;

}

};

class Triangle : public Shape {

private:

double base;

double height;

public:

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

double area() const override {

return 0.5 \* base \* height;

}

};

int main() {

Shape\* shapes[3];

shapes[0] = new Circle(5.0);

shapes[1] = new Rectangle(4.0, 6.0);

shapes[2] = new Triangle(3.0, 8.0);

for (int i = 0; i < 3; i++) {

cout << "Shape " << i + 1 << " area: " << shapes[i]->area() <<endl;

delete shapes[i];

}

return 0;

}

1. **Extend the previous program to include a function that sorts an array of shapes based on their area. The function should use polymorphism to determine the area of each shape and compare them. The program should allow the user to create an array of shapes of different types and sizes and sort them by area.**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Shape {

public:

virtual double area() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return 3.14159 \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

double area() const override {

return width \* height;

}

};

class Triangle : public Shape {

private:

double base;

double height;

public:

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

double area() const override {

return 0.5 \* base \* height;

}

};

bool compareArea(const Shape\* shape1, const Shape\* shape2) {

return shape1->area() < shape2->area();

}

int main() {

vector<Shape\*> shapes;

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 8.0);

shapes.push\_back(&circle);

shapes.push\_back(&rectangle);

shapes.push\_back(&triangle);

cout << "Unsorted Shapes:" << endl;

for (const auto& shape : shapes) {

cout << "Area: " << shape->area() << endl;

}

sort(shapes.begin(), shapes.end(), compareArea);

cout << "\nSorted Shapes:" << endl;

for (const auto& shape : shapes) {

cout << "Area: " << shape->area() << endl;

}

return 0;

}

**Reflection**

1. **Reflect on what you learned in this assignment. What was challenging, and what did you find interesting?**

It was difficult to learn and understand the concept of polymorphism, but in the end I figured it out, and I have now learned and understood it quite well.

1. **How can you apply what you learned in this assignment to future projects or your future career?**

The knowledge I gained from this assignment will be useful to me in our upcoming OOP Project. In our upcoming project, we will also include classes, polymorphism, and inheritance.