
# Assignment # 4

Submitted by : Hasnain Abbas

Sap Id : 42917

Course : OOP-2A

Section : BSCS – 3

Submitted to : Sir Shehzad Shameer

## ### Part 1: Theory

**(1)**

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

***Answer:***

Polymorphism is a fundamental concept in object-oriented programming that refers to the ability of objects of different types to be treated as if they were the same type. More specifically, polymorphism allows different objects to respond to the same message or method call in different ways based on their specific implementation.

In other words, polymorphism enables us to write code that can work with multiple classes of objects, as long as they have certain methods or properties in common. This makes our code more flexible and reusable, as we don't have to write separate code for each individual class.

Polymorphism is often achieved through inheritance and method overriding in object-oriented programming. When a subclass inherits from a superclass, it inherits all of the methods and properties of the superclass, but can also override or extend them to provide its own

implementation. This allows the subclass to behave like the superclass in some ways, but also have its own unique behaviors and characteristics.

Polymorphism can also be achieved through interface implementation, which allows unrelated classes to be treated as if they have the same set of methods or properties. This enables us to write more generic code that can work with a wide range of objects, as long as they conform to a certain interface.

Overall, polymorphism is a powerful concept that enables more flexible and reusable code in object-oriented programming.

## (2)

2. What is the difference between static and dynamic polymorphism?

***Answer :***

Static polymorphism and dynamic polymorphism are two different types of polymorphism in object-oriented programming.

Static polymorphism, also known as compile-time polymorphism, refers to polymorphic behavior that is resolved at compile-time. This means that the specific implementation of a method or function is determined based on the data types of the arguments passed to it at compile-time. Examples of static polymorphism in programming languages include function overloading and operator overloading.

Function overloading is when multiple functions have the same name but different parameter lists, allowing them to be called with different argument types. For example, a print function might have different implementations for integers, floats, and strings.

Operator overloading allows operators like + and - to be used with user-defined classes, enabling them to behave like built-in types. For example, a Vector class might define its own implementation of the + operator to add two vectors together.

Dynamic polymorphism, also known as run-time polymorphism, refers to polymorphic behavior that is resolved at runtime. This means that the

specific implementation of a method or function is determined based on the actual type of the object that the method is called on at runtime. Examples of dynamic polymorphism in programming languages include method overriding and virtual functions.

Method overriding is when a subclass provides its own implementation of a method that is already defined in the superclass, allowing it to customize the behavior of the method for its specific needs.

Virtual functions are functions that can be overridden by subclasses and called using a pointer or reference to the base class, allowing the specific implementation of the function to be determined at runtime based on the actual type of the object.

In summary, the main difference between static and dynamic polymorphism is that static polymorphism is resolved at compile-time based on the data types of arguments, while dynamic polymorphism is resolved at runtime based on the actual type of the object.

## (3)

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

***Answer :***

C++ supports two types of polymorphism: compile-time or static polymorphism, and run-time or dynamic polymorphism.

1. Compile-time or static polymorphism:

This type of polymorphism is achieved through function overloading and operator overloading, and it is resolved by the compiler at compile-time. Function overloading is when multiple functions have the same name but different parameter lists, allowing them to be called with different argument types. Operator overloading allows operators like + and - to be used with user-defined classes, enabling them to behave like built-in types.

Example of function overloading:

```
```

```
#include <iostream>

int add(int x, int y) {
  return x + y;
}

double add(double x, double y) {
  return x + y;
}

int main() {
  std::cout << add(1, 2) << std::endl;       // calls int add(int, int)
  std::cout << add(1.5, 2.5) << std::endl;   // calls double add(double, double)
  return 0;
}
```

2. Run-time or dynamic polymorphism:

This type of polymorphism is achieved through inheritance and virtual functions, and it is resolved at runtime. Inheritance allows a subclass to inherit the methods and properties of a superclass, and virtual functions allow a subclass to override the implementation of a method in the superclass.

Example of run-time polymorphism:

```
#include <iostream>

class Shape {
public:
  virtual double area() { return 0; }
};

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

```
  virtual double area() override { return 3.14 * radius * radius; }
private:
  double radius;
};

class Square : public Shape {
public:
  Square(double s) : side(s) {}
  virtual double area() override { return side * side; }
private:
  double side;
};

int main() {
  Shape* s1 = new Circle(3);
  Shape* s2 = new Square(4);
  std::cout << s1->area() << std::endl;    // calls Circle::area()
  std::cout << s2->area() << std::endl;    // calls Square::area()
  delete s1;
```

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

In this example, the Shape class has a virtual function called area(), which is overridden in the Circle and Square classes. The main function creates instances of both the Circle and Square classes and assigns them to Shape pointers. When the area() function is called on these pointers, the implementation of the function is determined at runtime based on the actual type of the object, resulting in dynamic polymorphism.

## **(4)**

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

***Answer:***

A virtual function is a member function in a base class that is declared with the keyword "virtual" and is intended to be overridden by a corresponding function in a derived class. The virtual function enables run-time polymorphism, allowing the implementation of the function to be determined at run-time based on the actual type of the object.

Here's an example of a virtual function in C++:

```
class Animal {
public:
    virtual void speak() {
        std::cout << "Animal is speaking" << std::endl;
    }
};

class Dog : public Animal {
public:
```

```
    void speak() override {
        std::cout << "Dog is barking" << std::endl;
    }
};
```

In this example, the Animal class has a virtual function called speak(). This function is intended to be overridden by a derived class, in this case, the Dog class. When a pointer to an Animal object is used to call the speak() function, the implementation of the function is determined at runtime based on the actual type of the object. So if the pointer points to an Animal object, the Animal::speak() function is called. If the pointer points to a Dog object, the Dog::speak() function is called instead.

Virtual functions are used to provide a common interface for a group of related classes, while allowing each class to implement the interface in its own way. This enables the implementation of functions to be determined at run-time based on the actual type of the object, resulting in polymorphic behavior. Virtual functions are also essential for implementing the concept of inheritance, which is a fundamental feature of object-oriented programming.

## (5)

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

***Answer:***

Yes, a class can have both virtual and non-virtual functions. In fact, most classes in object-oriented programming have both virtual and non-virtual functions.

A non-virtual function is a regular member function that is bound at compile-time, which means that the function call is resolved based on the type of the object pointer or reference. This allows for faster function call resolution and is often used for simple, low-level functions.

A virtual function, on the other hand, is a member function that is bound at run-time, which means that the function call is resolved based on the

type of the actual object pointed to or referenced. This allows for dynamic polymorphism and is often used for more complex functions that are intended to be overridden in derived classes.

Here's an example of a class with both virtual and non-virtual functions:

```
class Animal {
public:
    void eat() {
        std::cout << "Animal is eating" << std::endl;
    }
    virtual void speak() {
        std::cout << "Animal is speaking" << std::endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
```

```
        std::cout << "Dog is barking" << std::endl;
    }
    void speak() override {
        std::cout << "Dog is barking" << std::endl;
    }
};
```

In this example, the Animal class has a non-virtual function called eat() and a virtual function called speak(). The Dog class inherits from the Animal class and overrides the speak() function. The Dog class also has a non-virtual function called bark().

So, in conclusion, it is perfectly acceptable and common for a class to have both virtual and non-virtual functions. Non-virtual functions are useful for simple functions that do not require dynamic polymorphism, while virtual functions are useful for complex functions that require dynamic polymorphism and are intended to be overridden in derived classes.

Here's an example C++ program that demonstrates the concept of function overloading:

## ***### Part 2: Implementation***

**(1)**

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

**Answer:**

```c++
#include <iostream>

using namespace std;

// Function to calculate the area of a rectangle
```

```
int area(int length, int width) {
    return length * width;
}

// Function to calculate the area of a circle
float area(float radius) {
    return 3.14 * radius * radius;
}

int main() {
    int length = 5;
    int width = 6;
    float radius = 3.5;

    cout << "Area of rectangle: " << area(length, width) << endl;
    cout << "Area of circle: " << area(radius) << endl;

    return 0;
}
```

```

In this program, we have two functions named `area()` that calculate the area of a rectangle and a circle, respectively. The first `area()` function takes two arguments, `length` and `width`, and returns their product. The second `area()` function takes a single argument, `radius`, and returns the area of a circle using the formula `πr²`.

In the `main()` function, we declare variables `length`, `width`, and `radius`, and pass them as arguments to the appropriate `area()` functions. When we call `area(length, width)`, the first `area()` function is called with the values of `length` and `width`, and when we call `area(radius)`, the second `area()` function is called with the value of `radius`.

This demonstrates function overloading, where two functions have the same name but different argument types or numbers. The compiler decides which function to call based on the arguments passed to it.

**(2)**
```

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

***Answer:***

Here's an example C++ program that demonstrates the concept of operator overloading:

```c++
#include <iostream>

using namespace std;

class Complex {
private:
    int real;
    int imaginary;
public:
    Complex(int r = 0, int i =0) {
        real = r;
```

```
        imaginary = i;
    }
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imaginary = imaginary + obj.imaginary;
        return res;
    }
    Complex operator - (Complex const &obj) {
        Complex res;
        res.real = real - obj.real;
        res.imaginary = imaginary - obj.imaginary;
        return res;
    }
    void display() {
        cout << real << " + i" << imaginary << endl;
    }
};
```

```
int main() {
    Complex c1(5, 3);
    Complex c2(2, 7);
    Complex c3 = c1 + c2;
    Complex c4 = c1 - c2;

    c3.display();
    c4.display();

    return 0;
}
```

In this program, we define a `Complex` class that represents a complex number with a real and imaginary part. We then overload the `+` and `-` operators for the `Complex` class using the `operator +` and `operator -` functions.

In the `operator +` function, we create a new `Complex` object `res` and add the real and imaginary parts of the two `Complex` objects `this` and `obj`. We then return `res`.

In the `operator -` function, we create a new `Complex` object `res` and subtract the real and imaginary parts of the two `Complex` objects `this` and `obj`. We then return `res`.

In the `main()` function, we create two `Complex` objects `c1` and `c2`, and use the overloaded `+` and `-` operators to add and subtract them. We then display the results using the `display()` function of the `Complex` class.

This demonstrates operator overloading, where we define new meanings for existing operators. In this example, we define how the `+` and `-` operators should work for the `Complex` class, allowing us to add and subtract complex numbers using the `+` and `-` operators just like we would with integers or floats.

**(3)**

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

***Answer:***

Here's an example C++ program that demonstrates the concept of runtime polymorphism using virtual functions:

```c++
#include <iostream>

using namespace std;

class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape." << endl;
    }
};
```

```
class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing a circle." << endl;
    }
};

class Square : public Shape {
public:
    void draw() {
        cout << "Drawing a square." << endl;
    }
};

int main() {
    Shape *shape;

    Circle circle;
    Square square;
```

```
    shape = &circle;
    shape->draw();

    shape = &square;
    shape->draw();

    return 0;
}
```

In this program, we define a `Shape` class that has a virtual `draw()` function. We then define two derived classes, `Circle` and `Square`, that override the `draw()` function to draw a circle or square, respectively.

In the `main()` function, we create a pointer `shape` of type `Shape`. We then create `Circle` and `Square` objects, and assign their addresses to the `shape` pointer.

When we call `shape->draw()`, the `draw()` function of the appropriate derived class is called based on the object that `shape` is pointing to at runtime. This is runtime polymorphism, where the function to be called is determined at runtime based on the object being referred to, rather than at compile time.

In this example, when `shape->draw()` is called with `shape` pointing to a `Circle` object, the `draw()` function of the `Circle` class is called, and when it is called with `shape` pointing to a `Square` object, the `draw()` function of the `Square` class is called. This allows us to write more flexible and extensible code, as we can use a single pointer or reference to refer to objects of different classes and call their appropriate functions based on the object being referred to.

## (4)

4. Write a C++ program that demonstrates the concept of compile-time polymorphism using templates.

***Answer:***

Here's an example C++ program that demonstrates the concept of compile-time polymorphism using templates:

```c++
#include <iostream>

using namespace std;

template<typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    int intSum = add<int>(5, 3);
    float floatSum = add<float>(2.5, 1.5);

    cout << "Integer sum: " << intSum << endl;
    cout << "Float sum: " << floatSum << endl;
```

```
    return 0;
}
```

In this program, we define a template function `add()` that takes two arguments of the same type `T` and returns their sum. The `T` type is a template parameter that is determined at compile time based on the types of the arguments passed to the function.

In the `main()` function, we call the `add()` function twice, once with two `int` arguments and once with two `float` arguments. We specify the type of the template parameter using angle brackets `< >` and pass the two arguments to the function.

At compile time, the `add()` function is instantiated twice, once with the `int` type and once with the `float` type, and the appropriate code is generated for each instantiation. This is compile-time polymorphism, where the appropriate function to be called is determined at compile time based on the types of the arguments passed to the function.

In this example, when `add<int>(5, 3)` is called, the `add()` function with `int` template parameter is instantiated and the sum of the two `int` arguments is returned. Similarly, when `add<float>(2.5, 1.5)` is called, the `add()` function with `float` template parameter is instantiated and the sum of the two `float` arguments is returned.

This allows us to write generic code that can handle different types of arguments without having to write separate functions for each type.

## ### Part 3: Application

### (1)

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.

***Answer:***

Here's an example C++ program that uses polymorphism to create a hierarchy of shapes:

```c++
#include <iostream>
#include <cmath>

using namespace std;

class Shape {
public:
    virtual double area() = 0;
};

class Circle : public Shape {
```

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

    double area() {
        return M_PI * pow(radius, 2);
    }
};

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

    double area() {
        return length * width;
    }
```

```
};

class Triangle : public Shape {
private:
    double base, height;
public:
    Triangle(double b, double h) : base(b), height(h) {}

    double area() {
        return 0.5 * base * height;
    }
};

int main() {
    Shape *shapes[3];

    shapes[0] = new Circle(3);
    shapes[1] = new Rectangle(4, 5);
    shapes[2] = new Triangle(6, 7);
```

```
    for (int i = 0; i < 3; i++) {
        cout << "Area of shape " << i+1 << ": " << shapes[i]->area() << endl;
    }

    return 0;
}
```

In this program, we define a base class `Shape` with a pure virtual function `area()`. We then define three derived classes, `Circle`, `Rectangle`, and `Triangle`, each with their own implementation of the `area()` function that calculates the area of the shape.

In the `main()` function, we create an array of `Shape` pointers and assign them to objects of the three derived classes. We then loop through the array and call the `area()` function for each shape, which is determined at runtime based on the type of the object being referred to. This allows us to calculate the area of each shape using the same method, without having to write separate code for each type of shape.

In this example, we create a `Circle` object with radius `3`, a `Rectangle` object with length `4` and width `5`, and a `Triangle` object with base `6` and height `7`. We then loop through the `shapes` array and call the `area()` function for each shape. The appropriate implementation of the `area()` function is called based on the type of the object being referred to, and the area of each shape is calculated and printed to the console.

## (2)

2. 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.

***Answer:***

Here's an example program that extends the previous one to include a function that sorts an array of shapes based on their area:

```c++
#include <iostream>
#include <cmath>
#include <algorithm>

using namespace std;

class Shape {
public:
    virtual double area() const = 0;
    virtual ~Shape() {} // virtual destructor
};

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

```
    double area() const {
        return M_PI * pow(radius, 2);
    }
};

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

    double area() const {
        return length * width;
    }
};

class Triangle : public Shape {
private:
    double base, height;
```

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

    double area() const {
        return 0.5 * base * height;
    }
};

bool sortByArea(const Shape* s1, const Shape* s2) {
    return s1->area() < s2->area();
}

int main() {
    const int numShapes = 4;
    Shape *shapes[numShapes];

    shapes[0] = new Circle(3);
    shapes[1] = new Rectangle(4, 5);
    shapes[2] = new Triangle(6, 7);
```

```
    shapes[3] = new Circle(4);

    // Sort shapes by area
    sort(shapes, shapes + numShapes, sortByArea);

    // Print sorted shapes
    for (int i = 0; i < numShapes; i++) {
        cout << "Shape " << i+1 << " area: " << shapes[i]->area() << endl;
    }

    // Free memory
    for (int i = 0; i < numShapes; i++) {
        delete shapes[i];
    }

    return 0;
}
```

In this program, we add a new function `sortByArea()` that takes two `Shape` pointers as arguments and compares their areas using polymorphism. The `sort()` function from the `<algorithm>` library is used to sort the `shapes` array based on their areas.

We also add a virtual destructor to the `Shape` class, which ensures that the destructors of the derived classes are called properly when a `Shape` object is deleted.

In the `main()` function, we create an array of `Shape` pointers and assign them to objects of different derived classes. We then sort the `shapes` array based on their areas using the `sortByArea()` function, and print the sorted array to the console.

In this example, we create an array of four shapes (a `Circle` object with radius `3`, a `Rectangle` object with length `4` and width `5`, a `Triangle` object with base `6` and height `7`, and another `Circle` object with radius `4`). We then sort the array based on their areas and print the sorted array to the console.

## ### Part 4: Reflection

# (1)

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

***Answer:***

As an AI language model, I cannot reflect on my learning, but I can provide some general insights.

This assignment covered various important concepts of object-oriented programming (OOP), including polymorphism, inheritance, and templates. Understanding these concepts is essential for building complex and scalable software systems.

One of the challenging aspects of this assignment was understanding the different types of polymorphism and their applications in C++. It required a thorough understanding of virtual functions, operator overloading, and templates, among others.

I found it interesting to see how these concepts can be applied in real-world scenarios. For instance, the example program that creates a hierarchy of shapes using polymorphism demonstrates how OOP concepts can be used to create reusable and extensible software components.

Overall, this assignment provided a good opportunity to practice and reinforce my understanding of OOP concepts in C++.

## (2)

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

***Answer:***

The concepts learned in this assignment are fundamental to object-oriented programming and are widely used in software development. Understanding and applying these concepts can help in developing scalable, maintainable, and extensible software systems.

In future projects or my future career, I can apply the concept of polymorphism to create reusable and extensible software components. For example, I can create a base class for a system that processes data and implement different derived classes that inherit from the base class to process different types of data. By doing so, I can ensure that the system can handle different data types in a flexible and extensible manner.

I can also apply the concept of operator overloading to define operators for custom classes, making them behave like built-in types. This can make the code more readable and intuitive.

Finally, templates can be used to create generic functions and classes that can work with different data types, which can help in creating reusable code that is not tied to specific data types. This can also lead to better performance as it avoids the overhead of runtime polymorphism.

Overall, the concepts learned in this assignment are applicable in a wide range of software development projects and can help in creating better software systems.

# End

# Thank You So Much