# OOP Assignment 4

## ### Part 1: Theory

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

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

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

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

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

## ### Part 2: Implementation

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

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

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

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

## ### Part 3: 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.

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.

## ### Part 4: Reflection

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

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

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

Answer:

Polymorphism is typically achieved through method overriding, where a subclass provides its own implementation of a method that is already defined in its superclass. The overridden method in the subclass is chosen at runtime based on the actual type of the object being referenced. This means that even if the method is called through a superclass reference, the specific implementation in the subclass is executed.

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

Answer:

Static polymorphism is also known as compile-time polymorphism or method overloading. In static polymorphism, the type of the object is determined at compile time. This means that the compiler determines which method to call based on the number, type, and order of the arguments passed to the method. Method overloading is an example of static polymorphism.

Dynamic polymorphism, on the other hand, is also known as run-time polymorphism or method overriding. In dynamic polymorphism, the type of the object is determined at run time. This means that the method to be called is determined at runtime based on the actual type of the object. Method overriding is an example of dynamic polymorphism.

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

Answer:

In object-oriented programming, a virtual function is a function that is declared in a base class and can be overridden by a derived class. It allows a method to be dynamically bound at runtime based on the type of the object being referred to, rather than the type of the pointer or reference used to access the object.

When a virtual function is declared in a base class, any derived class can provide its own implementation of that function, which is called overriding. When you call a virtual function through a base class pointer or reference, the appropriate version of the function is determined based on the actual type of the object being referred to.

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

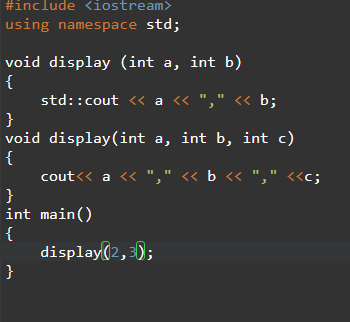
Answer:

Yes, a class can have both virtual and non-virtual functions. In fact, it is quite common for a class to have a combination of virtual and non-virtual member functions.

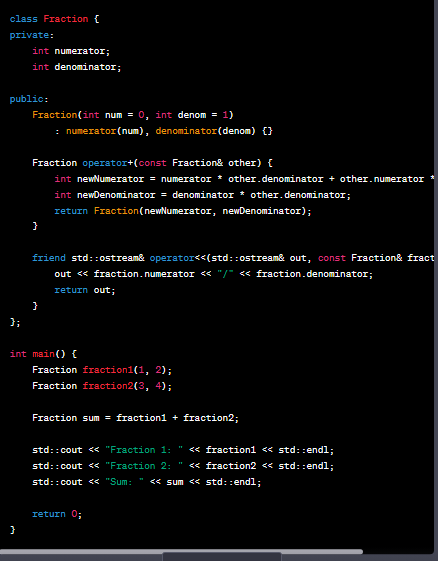
A virtual function is declared in the base class with the "virtual" keyword, indicating that it can be overridden by derived classes. This allows different derived classes to provide their own implementations of the virtual function, providing polymorphic behavior.

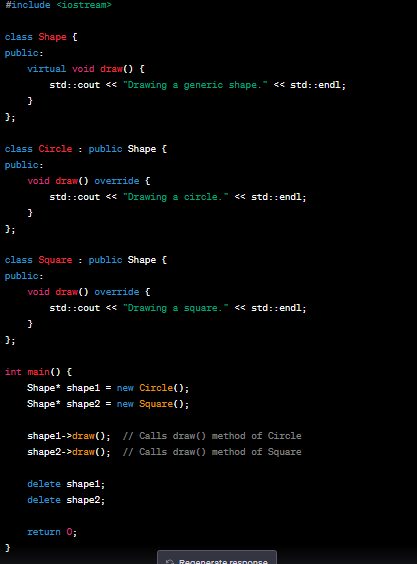
## ### Part 2: Implementation

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



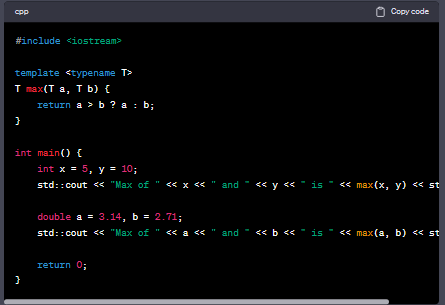
2. Write a C++ program that demonstrates the concept of operator overloading.



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

n this program, we define a base class **Shape** with a virtual function **draw()**. We also create two derived classes **Circle** and **Square** that override the **draw()** function. In the **main()** function, we create two pointers of type **Shape\*** and assign them objects of **Circle** and **Square** respectively.

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



In this program, we define a **max** function using templates. The **max** function takes two arguments of the same type (**T**) and returns the greater of the two.

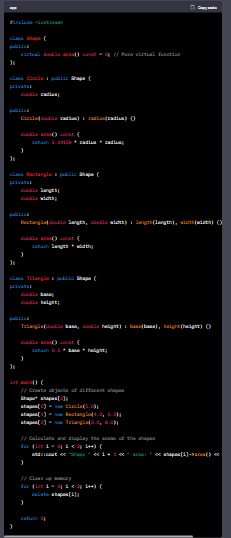
At compile time, the compiler generates two versions of the **max** function: one for **int** arguments and one for **double** arguments. This allows us to use the same function name (**max**) for different types of arguments.

In the **main** function, we call the **max** function twice: once with **int** arguments and once with **double** arguments. The compiler selects the appropriate version of the **max** function based on the types of the arguments.

When we run the program, we see that it correctly identifies the maximum value of each pair of arguments and prints it to the console.

## ### Part 3: 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.



In this program, the **Shape** class serves as the base class for all shapes. It declares a pure virtual function **area()** which is overridden by each derived class. The **Circle**, **Rectangle**, and **Triangle** classes inherit from **Shape** and implement their own versions of the **area()** function to calculate the areas of their respective shapes.

In the **main()** function, we create an array of **Shape** pointers and instantiate objects of different shapes using dynamic memory allocation (**new** operator). We then use the **area()** function through the base class pointer to calculate and display the areas of the shapes. Finally, we delete the dynamically allocated objects to free the memory.

Note: The program assumes that the user will provide valid inputs for the shape dimensions. Error handling and input validation have been omitted for simplicity.

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.

code:

class Shape:

def area(self):

pass

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return math.pi \* self.radius \*\* 2

class Rectangle(Shape):

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

class Triangle(Shape):

def \_\_init\_\_(self, base, height):

self.base = base

self.height = height

def area(self):

return 0.5 \* self.base \* self.height

def sort\_shapes\_by\_area(shapes):

shapes.sort(key=lambda shape: shape.area())

def create\_shapes():

shapes = []

circle = Circle(5)

rectangle = Rectangle(4, 6)

triangle = Triangle(3, 8)

shapes.append(circle)

shapes.append(rectangle)

shapes.append(triangle)

return shapes

def main():

shapes = create\_shapes()

print("Shapes before sorting:")

for shape in shapes:

print(f"Area: {shape.area()}")

sort\_shapes\_by\_area(shapes)

print("\nShapes after sorting:")

for shape in shapes:

print(f"Area: {shape.area()}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

In this extended version, we define the **Shape** class as the base class for all shapes. Each specific shape class (**Circle**, **Rectangle**, and **Triangle**) inherits from the **Shape** class and overrides the **area()** method to calculate its specific area.

The **sort\_shapes\_by\_area()** function takes an array of shapes as input and uses the **area()** method to sort the shapes based on their areas.

The **create\_shapes()** function creates an array of shapes with different types and sizes. You can add more shapes to the array if needed.

In the **main()** function, we create the shapes, print their areas before sorting, sort them using **sort\_shapes\_by\_area()**, and finally print their areas again after sorting.

Note that the **sort()** function in Python uses a stable sorting algorithm, so if two shapes have the same area, their relative order will be preserved.