PART A

1.Definition: In your own words, define polymorphism. Why is polymorphism often considered one of the pillars of OOP?

Polymorphism is the ability of different objects to be treated as the same type while maintaining their unique behaviors.
Polymorphism is often considered one of the pillars of object-oriented programming (OOP) because it makes code more flexible, reusable, and easier to extend. Alongside encapsulation, inheritance, and abstraction, it helps us build systems that can adapt to change.

2. Compile-Time vs. Runtime: Provide a one-sentence explanation of compile-time polymorphism (method overloading). Provide a one-sentence explanation of runtime polymorphism (method overriding). Which type requires an inheritance relationship, and why?

Compile-time polymorphism occurs when multiple methods in the same class have the same name but different parameter lists. The method to be executed is determined by the compiler at compile time based on the arguments provided in the method call.

Runtime polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method that gets executed is determined at runtime based on the actual type of the object, enabling dynamic method dispatch.

Runtime polymorphism (method overriding) requires an inheritance relationship because the overriding method must be defined in a subclass, replacing or extending the behavior of the method inherited from the parent class.

3. Method Overloading: Why might a class have multiple methods with the same name but different parameter lists? Give a brief example (no full code needed) of how method overloading can simplify user interactions with a class.

A class may have multiple methods with the same name but different parameter lists to enhance code readability, maintainability, and usability. It allows developers to define multiple ways of performing a similar action while keeping method calls intuitive and reducing the need for different method names. Example:

A Print class can have overloaded print methods:

print(String text) =Prints a text message

print(int number) =Prints a number

4. Method Overriding: Describe how a derived class overrides a base class's method to provide specialized behavior. In some languages (e.g., C++), the virtual keyword or annotations are used. Why might this be needed?

A derived class overrides a base class's method by redefining the method with the same signature (name, parameters, return type) but providing a new implementation. This allows the derived class to offer its specific behavior, which is executed instead of the base class's method when the method is called on an instance of the derived class.

The virtual keyword (in C++) is used to explicitly indicate that a method is meant to be overridden. This ensures that the correct method is called at runtime instead of a method being statically bound during compilation. Without the virtual keyword, C++ will use static binding and may not call the overridden method in derived classes.

PART B

1. Option 1: Minimal Code (No More Than 20 Lines)

Base Class & Derived Classes:Create a base class Shape with an abstract or virtual method: draw().Create two derived classes, Circle and Rectangle, each overriding draw() to print or return a shape-specific action.

Demonstration:Show a short snippet where you create a list/array of Shape* or references, store both a Circle and a Rectangle, and then call draw() on each.Emphasize how the correct draw() method is chosen at runtime.

```
class Shape {
public:
    virtual void draw() { }  // Virtual function
    virtual ~Shape() {}  // Virtual destructor function
};

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

class Rectangle : public Shape {
public:
    void draw() override { cout << "Drawing a Rectangle" << endl; }
};

int main() {
    Shape* shapes[2];  // Array of Shape pointers

    Circle circle;
    Rectangle rectangle;

    shapes[0] = &circle;     // Store Circle object
    shapes[1] = &rectangle;  // Store Rectangle object

    // Demonstrating runtime polymorphism
    for (int i = 0; i < 2; i++) {
        shapes[i]->draw();  // Calls the appropriate draw() method at runtime
    }

    return 0;
}
```

The correct draw() method is chosen at runtime because the draw() method is virtual in the base class Shape. When the draw() method is called on a Shape* pointer, C++ uses dynamic dispatch to call the appropriate method based on the actual object type (either Circle or Rectangle), not the type of the pointer. This allows runtime polymorphism.

PART C

1. Overloaded Methods:Imagine a class Calculator that has multiple calculate() methods, each accepting different parameter types (e.g., (int, int), (double, double)).How is the compile-time resolution used here?

```
class Calculator {
public:
    // Method to calculate sum of two integers
    int calculate(int a, int b) {
        return a + b;
    }

    // Method to calculate sum of three integers
    int calculate(int a, int b, int c) {
        return a + b + c;
    }

    // Method to calculate the area of a rectangle (length * width)
    double calculate(double length, double width) {
        return length * width;
    }
};

int main() {
    Calculator calc;

    // Using method with two integers
    cout << "Sum of 2 numbers: " << calc.calculate(5, 3) << endl;

    // Using method with three integers
    cout << "Sum of 3 numbers: " << calc.calculate(1, 2, 3) << endl;

    // Using method with two doubles (area of rectangle)
    cout << "Area of rectangle: " << calc.calculate(5.5, 3.2) << endl;

    return 0;
}
```

The Calculator class has three overloaded calculate() methods with different parameter types and counts:

Two integers (calculate(int, int))

Three integers (calculate(int, int, int))

Two doubles (calculate(double, double))

2. Overridden Methods:In your Shape example (or another scenario), the draw() method is overridden in derived classes.When does the decision for which method to call occur (compile time or runtime)?Why does this matter for flexible code design?

```
class Shape {
public:
    virtual void draw() { }  // Virtual function
    virtual ~Shape() {}  // Virtual destructor function
};

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

class Rectangle : public Shape {
public:
    void draw() override { cout << "Drawing a Rectangle" << endl; }
};

int main() {
    Shape* shapes[2];  // Array of Shape pointers

    Circle circle;
    Rectangle rectangle;

    shapes[0] = &circle;     // Store Circle object
    shapes[1] = &rectangle;  // Store Rectangle object

    // Demonstrating runtime polymorphism
    for (int i = 0; i < 2; i++) {
        shapes[i]->draw();  // Calls the appropriate draw() method at runtime
    }

    return 0;
}
```

The decision for which overridden method to call occurs at runtime. This is because the method call is determined by the actual object type that the base class pointer or reference is pointing to, not the type of the reference itself.

It matters for flexible code design because runtime polymorphism allows you to write code that works with multiple derived classes without knowing the specific types in advance. This makes the code more extensible, as you can add new derived classes without modifying existing code, supporting the open and closed extension and modification.

PART D

1.Practical Example: Briefly outline a scenario (in a game, a UI framework, or a simulation) where polymorphism is essential. Why does it reduce code duplication or improve design?

The vehicle is the base class with a virtual function startEngine().Car, Motorcycle, and Truck are derived classes, each overriding startEngine() to give their behavior.

```
class Vehicle {
public:
    virtual void startEngine() { // Virtual function for polymorphism

    }
};

// Derived class: Car
class Car : public Vehicle {
public:
    void startEngine() override { // Overriding the base class function

};

// Derived class: Motorcycle
class Motorcycle : public Vehicle {
public:
    void startEngine() override { // Overriding the base class function

    }
};

// Derived class: Truck
class Truck : public Vehicle {
public:
    void startEngine() override { // Overriding the base class function

    }
};
```

Polymorphism reduces duplication and improves design by allowing you to use a single interface (e.g., startEngine()) for different object types (e.g., Car, Motorcycle, Truck). This eliminates the need for separate code for each type, making the code more reusable, modular, and easier to

extend. New vehicle types can be added without changing existing code, keeping the system simpler and more maintainable.

2.Potential Pitfalls:List one possible confusion or pitfall when using method overloading.List one potential pitfall when relying heavily on runtime polymorphism (e.g., performance, debugging complexity).

When you overload methods, it can sometimes be unclear which method the program will call, especially if the parameters are similar but not identical. This can lead to unexpected behavior, where the wrong method is invoked.

With runtime polymorphism, since the actual method to be called isn't determined until the program is running, it can be harder to debug. We may not know which method is being called, making it tricky to trace errors, especially in large programs with many classes.

3.Checking Understanding:If a new Triangle class is added to your Shape hierarchy, how does polymorphism help you not modify the existing code that uses Shape references?

```
class Shape {
public:
    virtual void draw() { // Virtual function for polymorphism
    }
};

// Derived class: Circle
class Circle : public Shape {
public:
    void draw() override { // Overriding the base class function
    }
};

// Derived class: Rectangle
class Rectangle : public Shape {
public:
    void draw() override { // Overriding the base class function
    }
};

// New derived class: Triangle
class Triangle : public Shape {
public:
    void draw() override { // Overriding the base class function
        cout << "Drawing a triangle!" << endl;
    }
};
```

Polymorphism allows you to add new derived classes (like Triangle) without modifying existing code that uses Shape references. The program automatically calls the correct draw() method at runtime based on the actual object type, ensuring the existing code remains unchanged.