Part A: Conceptual Questions

1. Definition of Polymorphism

- Polymorphism in object-oriented programming means "many forms." It allows us to use one interface (like a method name) to represent different behaviors depending on the object that's calling it. For example, a draw() function can behave differently depending on whether it's called by a Circle or Rectangle object.

Why it's important:

- Polymorphism is one of the core pillars of OOP because it allows flexibility, reduces redundancy, and helps write cleaner, reusable code by letting different objects respond in their own way to the same method call.

2. Compile-Time vs. Runtime Polymorphism

- Compile-time polymorphism occurs through method overloading, where multiple methods have the same name but different parameters. The compiler determines which method to call.

- Runtime polymorphism occurs through method overriding, where a derived class provides a different implementation of a method defined in the base class. The decision is made during program execution.

Which one requires inheritance?

- Runtime polymorphism requires an inheritance relationship because the method must be defined in the base class and overridden in the derived class.

3. Method Overloading

Why use it?

- To allow the same method to handle different types or numbers of parameters. This makes the class more user-friendly and versatile.

Example:

```
class Printer {
public:
   void print(int num) {
      cout << "Printing number: " << num << endl;
   }

   void print(string text) {
      cout << "Printing text: " << text << endl;
   }
};
```

Calling print(5) and print("Hello") works without needing two different method names.

4. Method Overriding

How it works:

- When a derived class redefines a method from the base class to give it specific behavior. The function name and parameters stay the same, but the method body is different.

Why use virtual in C++?

- In C++, the virtual keyword is needed in the base class to enable runtime polymorphism. Without it, the base class version would run even if the object is actually of the derived class.

Part B: Minimal Demonstration (C++ Code)

```
#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() = 0; // Pure virtual method
};

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];
    shapes[0] = new Circle();
    shapes[1] = new Rectangle();

    for (int i = 0; i < 2; i++) {
```

```
        shapes[i]->draw(); // Runtime polymorphism
        delete shapes[i];
    }

    return 0;
}
```

Explanation:

- Even though we're using a base class pointer (Shape*), the correct draw() method is called depending on the actual object (either Circle or Rectangle). This demonstrates runtime polymorphism — the method is chosen at runtime, not at compile-time.

Part C: Overloading vs. Overriding Distinctions

1. Overloaded Methods (Compile-Time)

```
class Calculator {
public:
    int calculate(int a, int b) {
        return a + b;
    }

    double calculate(double a, double b) {
        return a * b;
    }
};
```

The compiler decides which version of calculate() to use based on the types of the arguments. This decision is made at compile time.

2. Overridden Methods (Runtime)

From our earlier example:
Shape* shape = new Circle();
shape->draw(); // Output: Drawing a Circle.
The actual version of draw() is determined at runtime, based on the object's actual type (Circle), even though the pointer is of type Shape*.

Why it matters:

- Runtime polymorphism allows writing generic code that still behaves correctly for specific objects, which makes large projects easier to manage and extend.

Part D: Reflection & Real-World Applications

Practical Example
In a simulation game, you might have a base class Character with a method interact(). Different characters like NPC, Enemy, and Trader inherit from it and override interact() with different behaviors.

You could then loop through a list of Character* objects and calls interact() on each one — the correct behavior happens automatically, without writing a big if-else or switch block.

Potential Pitfalls

Overloading Pitfall:
If methods are too similar (e.g., print(int) vs. print(float)), the compiler might get confused or pick the wrong one.

Overriding Pitfall:
Forgetting to make the base method virtual in C++ can cause the wrong method (usually the base one) to be called, leading to unexpected results and difficult-to-find bugs.

Adding a Triangle — Why Polymorphism Helps

If I add a new class Triangle that also inherits from Shape:

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

I don’t need to change the existing code that uses Shape* — the loop in main() still works, and draw() for Triangle is called automatically. This proves how polymorphism keeps code extensible and maintainable.