Part A: Conceptual Questions

1. **Definition**
   - Polymorphism is the ability to change the behavior of a function or operator based on their types or number of arguments., which allows objects of different classes to respond to the same function calls in unique ways.
   - Polymorphism is considered one of the pillars of OOP because it provides flexibility and a consistent interface for working with objects, which are critical for building scalable and maintainable code.
2. **Compile-Time vs. Runtime**
   - Compile-time polymorphism (method overloading) occurs when multiple methods in the same class share the same name but differ in parameters, allowing the compiler to determine which method to call based on the method signature.
   - Runtime polymorphism (method overriding) occurs when a derived class provides a specific implementation of a method declared in the base class, and the method to be executed is determined at runtime through dynamic dispatch.
   - Runtime polymorphism requires an inheritance relationship because overriding involves redefining a base class method in a derived class to achieve dynamic behavior, which is central to polymorphism's flexibility.
3. **Method Overloading**
   - A class might have multiple methods with the same name but different parameter lists to provide a more intuitive interface for users, allowing them to interact with the class using the same method name for similar but slightly different tasks, depending on the context.
   - Example: A Calculator class could have multiple add() methods, including add(int a, int b) to add two integers, add(double a, double b) to add two decimal numbers, etc.
4. **Method Overriding**
   - A derived class overrides a base class's method by redefining the method with the same name, parameters, and return type to provide specialized behavior.
   - The virtual keyword is used to mark methods that can be overridden by derived classes. This ensures that the appropriate method in the derived class is invoked, even when the object is accessed through a base class pointer or reference.

Part B: Minimal Demonstration

```
// Base class
class Shape {
public:
    virtual void draw() const = 0; // Abstract method
    virtual ~Shape() {} // Virtual destructor
};

// Derived classes
class Circle : public Shape {
public:
    void draw() const {
        cout << "Drawing Circle" << endl;
    }
};

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

// Demonstration
int main() {
    Shape* shapes [2] = { new Circle(), new Rectangle() };

  for (int i = 0; i < 2; i++) {
        shapes[i] -> draw(); //Runtime polymorphism
    }

    for (int i = 0; i < 2; i++) {
        delete shapes[i]; //Deallocate memory
    }

    return 0;
}
```

<u>Part C: Overloading vs. Overriding Distinctions</u>

1. **Overloaded Methods**
    - For the Calculator class with multiple calculate() methods accepting different parameter types, compile-time resolution is used to determine which specific version of the method to use based on the method signature at the time of compilation.
2. **Overridden Methods**
    - In the Shape example, the decision about which draw() method to call occurs at **runtime** because the method is declared as virtual in the base class.
    - This matters for flexible code design because it ensures that we can add new derived classes without altering existing code and enables decisions to be made dynamically.

<u>Part D: Reflection & Real World Applications</u>

1. **Practical Example**
    - Imagine a game where different types of characters (Player, Enemy, NPC) share common actions like move() and attack(). Each character type has unique behavior - such as Player allowing for direct user control. Polymorphism is essential in this case as we can override methods from a base class Character to provide derived classes like Player and Enemy to have specific behaviors, allowing the game engine to invoke these methods without needing to know exact character types.
    - This reduces code duplication by allowing derived characters to share common attributes from the base Character class and improves design by allowing the implementation of new character behavior while leaving the rest of the game's code untouched.
2. **Potential Pitfalls**
    - One possible confusion when using method overloading occurs when overloaded methods have similar signatures, potentially leading to unexpected behavior if the compiler chooses the wrong method.
    - One potential pitfall of relying too heavily on runtime polymorphism is performance issues.
3. **Checking Understanding**
    - Polymorphism ensures that existing code utilizing Shape references or pointers does not require modification when a new Triangle class is added because the existing code already interacts with objects through the base class interface.