Part A: Conceptual Questions

1. **Inheritance Definition**
    - Inheritance - a mechanism where one class (called the derived or child class) acquires the properties and behaviors of another class (called the base or parent class). It allows the derived class to reuse, extend, or modify the functionality of the base class, enabling code reusability and hierarchical relationships.
    - Inheritance differs from composition and aggregation in that inheritance is a relationship where the derived class becomes a specialized version of the child class. In contrast, composition and aggregation combine objects to build complex structures.
2. **Types of Inheritance**
    - Single Inheritance: A derived class inherits from one base class
        - Example: A Cat class inheriting from an Animal class, where the Animal class provides generic methods like eat() or sleep(), and the Cat class specializes with meow().
    - Multiple Inheritance: A derived class inherits from two or more base classes.
        - Example: A Cyborg class inheriting from both Human and Robot classes, combing attributes like think() from Human and laserEyes() from Robot.
3. **Overriding Methods**
    - Method overriding allows a derived class to redefine a method inherited from the base class. This enables the derived class to adjust the behavior of the method to its own context.
    - One reason why you might override a method instead of adding a new method to a derived class is to maintain consistensy with the base class's interface.
4. **Real-World Analogy**
    - The concept of inheritance can be applied to a human parent and their child. A parent might have the characteristic of blue eyes, and their child inherits this trait while still having their own unique features.
    - The parent represents the base class, providing the initial characteristics (methods and properties), while the child represents the derived class, inheriting the traits but also capable of overriding or extending them, making them unique.

Part B: Minimal Coding Example

```
// Base class
class Vehicle {
public:
    string brand = "Honda";
    void drive() {
        cout << "Vehicle is driving." << endl;
    }
};

// Derived class
class Car : public Vehicle {
public:
    int doors = 4;
    void drive() {
        cout << "Car is driving." << endl;
    }
};

// Driver code
int main() {
    Vehicle v;
    Car c;

    v.drive(); // Calls Vehicle's drive()
    c.drive(); // Calls Car's overridden drive()

    return 0;
}
```

Part C: Short Reflection & Discussion

1. **When to Use Inheritance:**
    - Inheritance is beneficial when there's a clear hierarchical relationship, such as in a system managing employees.
    - Inheritance can be overkill in cases where there’s no natural "is-a" relationship, such as a Car class inheriting from a Wheel class.
2. **Method Overriding vs. Overloading:**
    - Method Overriding (Runtime Polymorphism) occurs when a derived class redefines a method from the base class, which happens at runtime.

- Method Overloading (Compile-time Polymorphism) happens when multiple methods in the same class share the same name but differ in parameter, which happens at compile time.
- Inheritance relies on overriding for *polymorphism*, where derived classes modify base class behavior to adapt to specific needs, maintaining a shared interface and enabling flexible designs.

3. **Inheritance vs. Interfaces/Abstract Classes:**
    - Inheritance differs from implementing an interface/abstract base class in that inheritance builds a hierarchy with shared behavior, while interfaces allow unrelated classes to share common functionality.
4. **Pitfalls of Multiple Inheritance:**
    - One potential problem with multiple inheritance is the diamond problem, which occurs when a class inherits from two classes that both inherit from a common base class, leading to duplicate members or conflicting method definitions.
    - One potential solution to mitigate this problem is to use *interface-based design* to avoid multiple inheritance entirely, as interfaces don’t carry implementation details, preventing ambiguity.