Part A: Conceptual Questions

1. Inheritance Definition
    - Inheritance is a concept where a class can acquire the attributes and methods of another class. This allows code reuse and hierarchical organization.
    - While inheritance focuses on establishing an "is-a" relationship (e.g., a Car is a Vehicle), composition/aggregation is more about a "has-a" relationship (e.g., a Car has a Engine or has a GPS system). Composition is generally more flexible and less tightly coupled than inheritance.

2. Types of Inheritance
    - Single Inheritance Example: A Dog class inherits from an Animal class to acquire traits like "eat" or "sleep." Appropriate for cases where a class specializes in a single base class.
    - Multiple Inheritance Example: A FlyingCar class inherits from both Vehicle and Aircraft to combine functionalities. Appropriate for cases where a class requires characteristics from multiple base classes.

3. Overriding Methods
    - Method overriding allows a derived class to provide its own specific implementation of a method defined in the base class. For example, a base Vehicle class has a drive() method, but a derived Car class overrides it to add specific behaviors like "starting the engine."
    - You may override a method instead of adding a new method because overriding ensures polymorphism, allowing the derived class's implementation to be called dynamically at runtime, preserving the interface defined by the base class.

4. Real-World Analogy
    - A child inheriting traits from their parents is a simple analogy. For example, a child might inherit physical traits like eye color (analogous to base class attributes) or behaviors like language proficiency (analogous to base class methods).
    - This aligns with OOP inheritance as the child class extends the capabilities of the parent class.

Part B: Minimal Coding

- Option 1: Minimal Coding

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

// Derived class
class Car : public Vehicle {
public:
    int doors;
    Car(string b, int d) : Vehicle(b), doors(d) {}
    void drive() override {
        cout << "Car is driving. Brand: " << brand << ", Doors: " << doors << endl;
    }
};

// Driver code
int main() {
    Vehicle v("GenericBrand");
    Car c("Toyota", 4);

    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
    - Beneficial: When creating a hierarchical relationship such as Animal -> Mammal -> Dog. It promotes code reuse and logical classification.
    - Overkill/Fragile: For unrelated classes, forcing inheritance can cause tight coupling and a fragile design. For example, using inheritance to share

methods between Car and Boat might not make sense if they aren’t logically related.

2. Method Overriding vs. Overloading
    - Overriding: Redefines a base class method in a derived class (runtime polymorphism).
    - Overloading: Same method name, but different parameters within the same class (compile-time).
    - Inheritance relies on overriding to dynamically change behavior without altering the base class.

3. Inheritance vs. Interfaces/Abstract Classes
    - A class can implement multiple interfaces (abstract classes), but inheritance binds it to one base class.

4. Pitfalls of Multiple Inheritance
    - Problem: The "diamond problem" occurs when multiple base classes define the same attribute/method, leading to ambiguity.
    - Solution: Use virtual inheritance (C++) or favor interfaces over multiple base classes.