Part A: Conceptual Questions

1. **Composition vs. Aggregation**
    - Composition is a relationship where one class "owns" the other. The lifecycle of the owned object is bound to the owner's lifecycle. If the owner object is destroyed, the owned object is also destroyed.
    - Aggregation is a relationship where one class references the other. However, the lifecycle of the referenced object is independent of the owner.
    - *Composition* implies a stronger form of ownership than aggregation.
2. **When to use**
    - In a banking application, a BankAccount class might use a TransactionHistory class via composition. Transaction history is tied to the account, and when an account is deleted, its transaction history should also be deleted, making composition more appropriate than inheritance.
    - Consider a multiplayer game where a Player class has a reference to a Team class. Aggregation is sufficient as players are associated with a team without owning the team. A player may leave a team, but the team is independent of the player.
3. **Differences from Inheritance**
    - The "has-a" relationship of composition/aggregation differs from the "is-a" relationship of inheritance. Inheritance describes the hierarchy of derived classes that inherit the behavior and attributes of a base class, whereas composition/aggregation describes relationships where one class contains another as part of its definition.
    - An OOP design may favor composition over inheritance for code flexibility and maintainability. Composition allows you to change behavior dynamically by replacing or modifying components whereas inheritance locks classes into a firm hierarchy.
4. **Real-World Analogy**
    - Composition: A car has an engine, and the engine is an essential part of the car. If the car is destroyed, the engine is also destroyed, reflecting the strong ownership that the Car object has over the Engine.
    - Aggregation: A car may have a driver, but the driver exists independently of the car. A driver can switch cars, and their lifecycle is not tied to a specific car.
    - These distinctions matter in code for lifecycle management and maintainability. You need to choose between composition and aggregation depending on whether the objects in your code require a strong coupling or a loose coupling

Part B: Minimal Class Design

```
class Address {
private:
    string street;
    string city;

public:
    Address(string s, string c) : street(s), city(c) {}
};

class Person {
private:
    string name;
    Address* address; // Aggregation: Person holds a reference to Address

public:
    // Constructor accepts a reference to an existing Address
    Person(string n, Address* addr) : name(n), address(addr) {}
};

int main() {
    // Address object created independently of Person
    Address addr("123 South St", "Denver");

    // Person object refers to the independent Address object
    Person p("Eli Coker", &addr);
    p.displayPerson();

    // The Address object continues to exist even after Person is out of scope
    cout << "Address still exists independently:" << endl;
    addr.displayAddress();

    return 0;
}
```

Part C: Reflection & Short Discussion

1. **Ownership & Lifecycle**
    - In a composition relationship, if the "parent" is destroyed or deallocated, the "child" is also destroyed.
    - In an aggregation relationship, the "child" object exists independently of the "parent." Even if the "parent" is destroyed, the "child" remains intact.
2. **Advantages & Pitfalls**
    - Advantage: Composition provides direct control over object lifecycles, ensuring that the "child" cannot exist without the "parent."
    - Pitfall: If composition is used where looser coupling is required, you risk creating unnecessary dependencies, reducing code flexibility.
3. **Contrast with Inheritance**
    - A “has-a” relationship indicates that an object contains or is associated with another object. An “is-a” relationship indicates that a derived class is a specialized form of the base class.
    - In situations that can be solved through composition or aggregation, we may avoid inheritance to ensure that our code is modular and flexible. Inheritance creates tighter coupling between base and derived classes, making the system harder to modify.