Part A: Conceptual Questions

1. Composition vs. Aggregation
    - Composition: A "has-a" relationship where the parent class owns the child class. The child object cannot exist without the parent.
    - Example: A Car and its Engine. If the Car is destroyed, the Engine ceases to exist.
    - Aggregation: A "has-a" relationship, but with looser coupling. The parent class references the child class, which can exist independently.
    - Example: A Team and its Player. A Player can belong to multiple teams or none.
    - Composition implies stronger ownership since the lifecycle of the child object is tied to the parent.

2. When to Use
    - Scenario for composition: In a gaming domain, a Character class could contain a Weapon object. If the Character is deleted, its Weapon should also be destroyed. This ensures proper ownership of the Weapon object.
    - Scenario for aggregation: In a banking system, an Account class could reference a Customer. Customers may exist independently of accounts (a customer can have no active accounts or multiple accounts). This is only partial ownership as the account doesn't own the customer, only refers to it.

3. Differences from Inheritance
    - Composition/Aggregation: They establish "has-a" relationships, where a class is composed of or references other objects, as opposed to the “is-a” relationship implied by inheritance.
    - An OOP design may favor composition because it avoids the tight coupling associated with inheritance. Inheritance can lead to inflexible designs when behaviors need to change dynamically.

4. Real-World Analogy
    - Composition: A car has an engine. The engine's life cycle depends on the car.
    - Aggregation: A car has a driver. The driver exists independently of the car.
    - These distinctions matter in code because properly modeling these relationships in code ensures maintainability and avoids unnecessary coupling, making systems easier to extend and debug.

Part B: Minimal Class Design

- Option 1: Minimal Class Example

```
class Person {
private:
   Address* address;  // Aggregation (address can exist independently)
```

```
public:
    Person(Address* addr) : address(addr) {}
    ~Person() { address = nullptr; }  // No ownership of Address object
};

class Address {
private:
    string street;
    string city;

public:
    Address(const string& street, const string& city)
        : street(street), city(city) {}
    string getCity() const { return city; }
};
```

- Address represents aggregation since it can exist without the Person. Lifecycle of Address is independent of Person.

Part C: Reflection & Short Discussion

1. Ownership & Lifecycle
    - Composition: The child object (e.g., Engine) is destroyed when the parent (e.g., Car) is destroyed.
    - Aggregation: The child object (e.g., Player) may outlive the parent (e.g., Team) and continue to exist independently.

2. Advantages & Pitfalls
    - Advantage of composition: It provides strong ownership and guarantees that the child object is cleaned up with the parent, avoiding memory leaks.
    - Pitfall of misusing composition: Overusing it can lead to rigid designs where objects are unnecessarily tied together, reducing flexibility.

3. Contrast with Inheritance
    - In practical code design, “has-a” relationships describe relationships between objects (Car has an Engine), while “is-a” relationships define a hierarchy where one class is a type of another (Car is a Vehicle).
    - We may avoid inheritance because it can lead to tight coupling and fragile base class problems, while composition and aggregation are more adaptable and promote reusable components.