Part A: Conceptual Questions

1. Composition vs. Aggregation (Definitions & Examples)

Composition:

- Composition is when one class is made up of one or more objects of another class, and those objects cannot exist independently of the parent. It's a strong ownership relationship.

Aggregation:

- Aggregation is when one class uses or refers to another class, but the child object can exist on its own. It's a weaker, more flexible relationship.

Examples:

Composition:

```
class Engine {
   // Engine exists only inside a Car
};

class Car {
   private:
      Engine engine; // Car "has-a" Engine (strong ownership)
};
```

Aggregation:

```
class Driver {
   // A Driver can exist without a Car
};
```

```
class Car {
    private:
        Driver* driver; // Car "has-a" Driver (loose connection)
};
```

Which is stronger?

- Composition is stronger when the parent object is destroyed, its child objects are also destroyed.

2. When to Use

Composition Scenario (Gaming):

In a game, a Character class might have a Weapon object. The weapon only exists as part of the character and is destroyed with them.

Aggregation Scenario (School system):

A Classroom can have multiple Student objects, but students can exist outside of a classroom. The relationship is looser, so aggregation is a better fit.

3. Differences from Inheritance

Inheritance is an “is-a” relationship. For example, a Dog is an Animal.

Composition and aggregation are “has-a” relationships. For example, a House has a Door.

Why choose composition over inheritance sometimes?

- Composition gives more flexibility. If a class changes later, it doesn’t break the hierarchy like inheritance can. It also avoids the tight coupling that inheritance creates.

4. Real-World Analogy
Example: A car uses both composition and aggregation.

Composition: A car has an engine, and when the car is gone, the engine is gone too.

Aggregation: A car has a driver, but the driver can exist without the car.

Why it matters in code:

- These distinctions help us manage memory, object life cycles, and system flexibility. Misunderstanding them can lead to tight coupling or memory issues.

Part B: Minimal Class Design

Example: Person and Address

```
#include <iostream>
#include <string>
using namespace std;

class Address {
public:
    string street;
    string city;

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

class Person {
private:
    Address* address; // Aggregation (we're using a pointer)
```

```
public:
    string name;

    Person(string n, Address* addr) : name(n), address(addr) {}

    void display() {
        cout << name << " lives at " << address->street << ", " << address->city
<< endl;
    }
};
```

Why Aggregation?

- In this example, Address exists outside of Person. It can be shared between people or live on after the person object is deleted. The lifecycle of the address is not controlled by Person, so this is aggregation.

Part C: Reflection & Short Discussion

1. Ownership & Lifecycle

Composition: If the parent object (like Car) is destroyed, its component (like Engine) is destroyed too.

Aggregation: If the parent is destroyed, the child (like Driver) can still exist — they're not tightly bound.

2. Advantages & Pitfalls

Advantage of composition:

You have full control over the child object's lifecycle, which is useful when building self-contained systems.

Pitfall if misused:
If you use composition where aggregation is better, you might end up destroying or duplicating objects that were meant to be shared — which can waste memory or cause logic errors.

3. Contrast with Inheritance

- Has-a (composition/aggregation): Focuses on building systems by combining objects (like a Computer has a Keyboard).

Is-a (inheritance): Focuses on creating hierarchies where one class is a specialized version of another (like Cat is an Animal).

Why avoid inheritance sometimes:

- Inheritance tightly links classes together. If a base class changes, it might break all subclasses. Composition and aggregation give more flexibility and are easier to manage in large systems.