## Week # 10

# Composition & Aggregation in OOP

### Objective:

Understand the concepts of composition and aggregation in Object-Oriented Programming (OOP) using C++ and learn how to implement these concepts in practical scenarios.

### Introduction:

Composition and Aggregation are two types of associations that represent relationships between objects. Composition is a strong form of association with a strict lifecycle dependency between the parent and child objects. If the parent object is destroyed, the child objects are also destroyed. Aggregation is a weaker form of association where the child objects can exist independently of the parent object.

| Feature | Composition | Aggregation |
|---|---|---|
| **Definition** | A strong form of association where the lifetime of the contained object is managed by the container object. | A weaker form of association where the contained objects can exist independently of the container object. |
| **Lifetime Dependency** | Contained objects do not exist independently. When the container is destroyed, contained objects are also destroyed. | Contained objects can exist independently. When the container is destroyed, contained objects may continue to exist. |
| **Ownership** | The container class owns the contained objects. | The container class does not own the contained objects. |
| **Example** | A `Person` class containing a `Heart` object. When the `Person` is destroyed, the `Heart` is also destroyed. | A `Library` class containing a collection of `Book` objects. The `Books` can exist independently of the `Library`. |
| **Code Example** (C++) | `cpp class Heart { /*...*/ }; class Person { private: Heart heart; public: Person() : heart() {} /*...*/ };` | `cpp class Book { /*...*/ }; class Library { private: vector<Book> books; public: void addBook(const Book& book) { books.push_back(book); } /*...*/ };` |

| | | |
|---|---|---|
| **UML Representation** | Filled diamond (black diamond) | Empty diamond (white diamond) |
| **Strength of Relationship** | Strong relationship | Weak relationship |
| **Reusability** | Low reusability of the contained object outside the container | High reusability of the contained object outside the container |
| **Implementation Complexity** | Generally more complex due to lifecycle management | Generally simpler due to independent lifecycle management |

**Composition in C++**

In composition, the contained objects are usually part of the parent object. This is often described as a "has-a" relationship.

**Aggregation in C++**

In aggregation, the contained objects can exist independently of the parent object. This is also a "has-a" relationship but with a weaker dependency.

**Part 1: Composition**

Create a Date class that contains day, month, and year.

Create a Person class that contains a name, age, and a Date object representing the birthdate.

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

class Date {
private:
   int day, month, year;
public:
   Date(int d, int m, int y) : day(d), month(m), year(y) {}
   void display() {
      cout << day << "/" << month << "/" << year << endl;
   }
};
class Person {
private:
```

```
    string name;
    int age;
    Date birthdate;
public:
    Person(string n, int a, Date b) : name(n), age(a), birthdate(b) {}
    void display() {
        cout << "Name: " << name << ", Age: " << age << ", Birthdate: ";
        birthdate.display();
    }
};
int main() {
    Date birthDate(1, 1, 2000);
    Person person("John Doe", 24, birthDate);
    person.display();
    return 0;
}
```

**Part 2: Aggregation**

Create a Car class that contains a model, manufacturer, and year.

Create a Garage class that contains a collection of Car objects.

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

class Car {
private:
    string model;
    string manufacturer;
    int year;
public:
    Car(string m, string manu, int y) : model(m), manufacturer(manu), year(y) {}
    void display() {
```

```
        cout << "Model: " << model << ", Manufacturer: " << manufacturer << ", Year: " << year << endl;
    }
};



class Garage {
private:
    vector<Car> cars;
public:
    void addCar(Car car) {
        cars.push_back(car);
    }
    void displayCars() {
        for (auto &car : cars) {
            car.display();
        }
    }
};
int main() {
    Car car1("Model S", "Tesla", 2020);
    Car car2("Mustang", "Ford", 2018);

    Garage garage;
    garage.addCar(car1);
    garage.addCar(car2);

    garage.displayCars();
    return 0;
}
```

## Choosing Between Composition and Aggregation

When designing a system, it is crucial to understand the relationship between objects to decide whether to use composition or aggregation:

### Use composition when:

The contained object is part of the container object and its existence is meaningless outside the container.
You want to enforce a strong lifecycle dependency where the destruction of the container object should also destroy the contained objects.

### Use aggregation when:

The contained objects can logically exist independently of the container.
You need to allow for the reuse of the contained objects in different contexts.

By carefully considering these factors, you can design your classes and their relationships in a way that best fits the logical and functional requirements of your system. This ensures that your code is both robust and maintainable, adhering to good object-oriented design principles.

## Practice Questions

### Question 1: Composition

Create a Book class that has title, author, and a Date object for the publication date. Implement a method to display the book details.

### Question 2: Aggregation

Create a Library class that can hold multiple Book objects. Implement methods to add books and display all books in the library.

### Question 3: Composition with push_back

Create a Playlist class that contains a collection of Song objects. Each Song object should have a title, artist, and duration. Implement methods to add songs and display the playlist.