**Lab#11**

**Association(Aggregation & Composition) in C++**

**Learning Outcomes**

In this lab you are expected to learn the following:

* Basic Concept and Implementation of Aggregation and Composition

**Types of Relationships in Object-Oriented Programming (OOP)**

One of the key advantages of Object-Oriented Programming is **code reusability**, which is achieved through **relationships between classes**. OOP generally supports four types of relationships:

* **Inheritance**
* **Association**
* **Aggregation**
* **Composition**

**Overview of Relationships**

| **Relationship Type** | **Description** | **Example Relation** |
| --- | --- | --- |
| Inheritance | "is-a" | Dog *is-a* Animal |
| Association | General relationship | Student *uses* Library |
| Aggregation | "has-a" (weak) | Team *has* Players |
| Composition | "part-of" (strong) | Human *has* Heart |

**6.5.1 Association, Aggregation, and Composition**

**Association**

* Refers to how multiple objects are related and interact.
* It can be:
  + One-to-one
  + One-to-many
  + Many-to-one
  + Many-to-many
* Both **aggregation** and **composition** are forms of association.

**Aggregation**

* **Weak association** where both objects can exist independently.
* **Example**: A Team object contains Player objects, but a player can exist without a team.

**Example Code:**

#include <iostream>

using namespace std;

class Address {

public:

string addressLine, city, province;

Address(string addressLine, string city, string province) {

this->addressLine = addressLine;

this->city = city;

this->province = province;

}

};

class Employee {

private:

Address\* address; // Aggregation: Employee HAS-A Address

public:

int id;

string name;

Employee(int id, string name, Address\* address) {

this->id = id;

this->name = name;

this->address = address;

}

void display() {

cout << id << " " << name << " "<< address->addressLine << " "<< address->city << " "<< address->province << endl; }

};

int main() {

Address a1("C-146, Sec-15", "Swabi", "KPK");

Employee e1(101, "Ali", &a1);

e1.display();

return 0;

}

**Output:**

101 Ali C-146, Sec-15 Swabi KPK

**Composition**

* **Strong association** where the lifetime of the owned object is managed by the owner.
* If the container object is destroyed, the contained objects are also destroyed.
* **Example**: A Human has a Heart, and a heart cannot exist independently of a human.

**Conceptual Example:**

public class Car {

private final Engine engine;

public Car() {

engine = new Engine(); // Engine is created within Car

}

}

class Engine {

// Engine cannot exist without Car

}

**C++ Example:**

#include <iostream>

using namespace std;

class Id {

public:

Id(double i) : id(i) {}

void printId() {

cout << "your Id is : " << id << endl;

}

private:

double id;

};

class Money {

public:

Money(int m) : money(m) {}

void printMoney() {

cout << "you have " << money << "$ " << endl;

}

private:

int money;

};

class Country {

public:

Country(string c) : country(c) {}

void printCountry() {

cout << country << endl;

}

private:

string country;

};

class Person {

public:

Person(string n, Id ii, Money mm, Country cc)

: name(n), identification(ii), cash(mm), state(cc) {}

void printInfoPerson() {

cout << name << endl;

state.printCountry();

identification.printId();

cash.printMoney();

}

private:

string name;

Id identification;

Money cash;

Country state;

};

int main() {

Country state("Italy");

Id identification(0.001);

Money cash(55);

Person p("David", identification, cash, state);

p.printInfoPerson();

return 0;

}

**Output:**

David

Italy

your Id is : 0.001

you have 55$

**An Overview of these concepts is given in the picture below:**

**For detailed explanation refer to this link:**

[**https://www.urbanpro.com/btech-tuition/in-this-article-we-will-try-to-understand**](https://www.urbanpro.com/btech-tuition/in-this-article-we-will-try-to-understand)

[**https://blog.devgenius.io/association-composition-and-aggregation-in-c-925465987061**](https://blog.devgenius.io/association-composition-and-aggregation-in-c-925465987061)

**TASK#11**

**Question #1:**

**Problem Scenario: Online Learning Platform using Aggregation**

You are designing a simplified **Online Learning Platform** (similar to Coursera or edX) where students can enroll in courses taught by instructors. In this platform, multiple **Courses** are offered. Each **Course** can be associated with many **Students** and many **Instructors**. However, **Students** and **Instructors** are *independent entities* they can exist even if the course is deleted or not yet created.

Use fixed size of arrays:

const int MAX\_COURSES = 5;

const int MAX\_STUDENTS = 10;

const int MAX\_INSTRUCTORS = 5;

1. **Student**
   * Has name, ID, and email
   * Can enroll in multiple courses
2. **Instructor**
   * Has name, ID, and expertise
   * Can teach multiple courses
3. **Course**
   * Has title, course code, and description
   * Has a list of enrolled **Students**
   * Has a list of assigned **Instructors**

**Aggregation Requirements:**

* A **Course** aggregates multiple **Students** and **Instructors**.
* If a **Course** is removed from the system:
  + The **Student** and **Instructor** objects should remain in memory (not deleted).
* Students and Instructors may participate in many Courses or none at all.

This structure reflects an **Aggregation** relationship.

Implement the following functionality:

* Register Students and Instructors independently
* Create Courses
* Enroll Students in Courses
* Assign Instructors to Courses
* Display:
  + Course details with enrolled Students and assigned Instructors
  + Student details including list of courses they are enrolled in

**Expected Output:**

Course: "Intro to Machine Learning"

Instructor(s): Dr. Khan, Dr. Ayesha

Enrolled Student(s): Ali, Sara, John

Student: Sara

Enrolled Courses: "Intro to Machine Learning", "Python for Beginners".

**Problem #02 Scenario: Document Editor using Composition**

You're tasked with designing a simplified **Document Editor System** (similar to Microsoft Word), focusing on how different parts of a document are structured and managed. In this system, you will model the hierarchical structure of a document, which is made up of **Pages**, **Paragraphs**, and **Sentences**. Each level is completely dependent on its parent object they cannot exist independently. We can use **arrays** to store the pages, paragraphs, and sentences. Track how many pages have been added in a document, and also track how many paragraphs have been added and also track how many sentences have been added.

1. **Document**
   * Contains multiple **Pages**
2. **Page**
   * Contains multiple **Paragraphs**
3. **Paragraph**
   * Contains multiple **Sentences**
4. **Sentence**
   * Stores actual text

**Composition Requirements:**

* A **Document** *owns* its **Pages**.
* A **Page** *owns* its **Paragraphs**.
* A **Paragraph** *owns* its **Sentences**.
* If the **Document** is deleted, all associated **Pages**, **Paragraphs**, and **Sentences** must be destroyed automatically.

This strict “part-of” relationship reflects **Composition** in object-oriented design.

Implement the system with the following:

* Use **composition** to embed:
  + Pages inside Document
  + Paragraphs inside Page
  + Sentences inside Paragraph
* Provide functions to:
  + Add a Page to a Document
  + Add a Paragraph to a Page
  + Add a Sentence to a Paragraph
  + Display the entire Document’s content hierarchically

**Expected Output:**

Document:

Page 1:

Paragraph 1:

Sentence: "Object-oriented programming is powerful."

Sentence: "Composition helps manage complexity."

Paragraph 2:

Sentence: "Each part has a clear role."

**Problem #03**

Write a class Point that has the following data members.

**X\_Coordinate: x coordinate of type integer**

**Y\_Cooridnate: y coordinate of type integer**

The Point class has following **member functions.**

1. A **default constructor** that initializes the data members to zero.

**Point()**

2. A **parameterized constructor** that accepts the parameters for each member variable.

**Point(int , int)**

3. A **copy constructor** that takes a previously constructed object as an argument.

**Point(const Point &p)**

4. Write **accessors** for each data member.

**int getX\_Coordinate() const**

**int getY\_Coordinate() const**

Write a class Line that represents a line segment between two Points hence it composes Point class.

The Line class has the following data members.

**Point\_1: a point P1 of type Point**

**Point\_2: a point P2 of type Point**

The Line class has the following **member functions.**

**Note: Use member initializer list for all constructors.**

1. A **default constructor** that initializes the **coordinates of two points to 4,6 and 2, 4.**

**Line()**

2. A **parameterized constructor.**

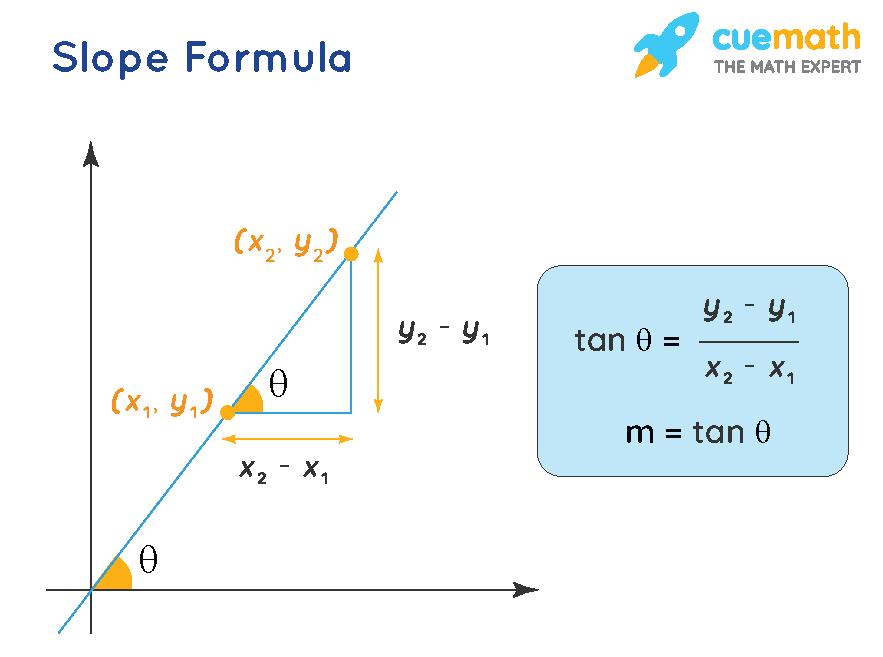
**Line(int x1, int y1, int x2, int y2)**

3. A **copy constructor** that takes two previously constructed Point objects as argument.

Line(const Point &p1, const Point &p2)

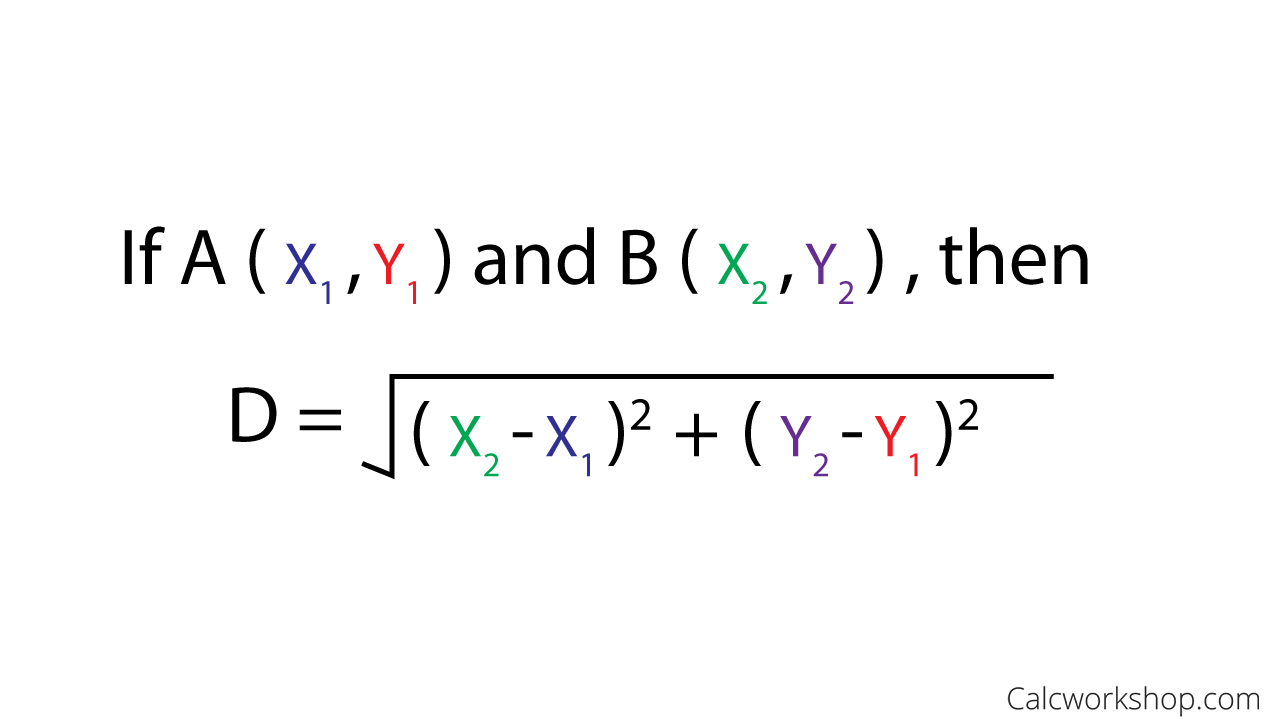
4. A member function **findSlope** that returns the slope of the length.

**float findSlope()**



5. A member function **findLength** that returns the length of the line segment using distance formula.

**float findLength()**



6. A member function **findMidPoint** that returns the midpoint of the line segment.

In this function you have to create a pointer of type Point, dynamically allocate memory

to the pointer, set midpoints and return it.

**Point& findMidPoint()**

