Lab13
Composition and Aggregation

OBJECT ORIENTED PROGRAMMING

BSCS-SPRING-2022

Object Composition:

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **object composition**.

There are two basic subtypes of object composition:

1) Composition

2) Aggregation

Composition:

In Composition, parent owns child entity so child entity can't exist without parent entity. We can't directly or independently access child entity.

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body can not be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed.

Aggregation:

In C++, aggregation is a process in which one class (as an entity reference) defines another class. It provides another way to reuse the class. Parent and child entity maintain Has-A relationship but both can also exist independently.
HAS-A relation simply means dynamic or run-time binding.

To qualify as an aggregation, an object must define the following relationships:

1. The member must be a part of a class.

- A member can belong to or more classes at a time.
- The member does not have any existence managed by the object.
- The member is unknown about the object's existence.
- The relationship is uni-directional.

Collectively, aggregation is a part-whole relationship, where the parts (member) are contained within the entire uni-directional relationship. However, unlike composition, members can belong to one object at a time. The entire object doesn't need to be responsible for the existence and lifespan of the members. In simple and sober language, **aggregation is not responsible for creating or destroying the members or parts**.

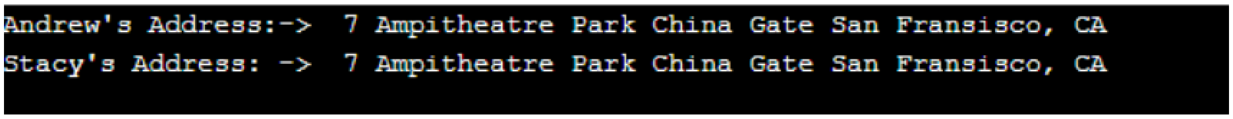
Example:

Consider a relationship between a person and their home address. However, the same address may belong to more than one person at a time in some significant order. Although, it is not managed by the person; the address existed even before the man and will tend to exist even after it. Additionally, a person knows where he/she lives, but the address does not have any information about what person lives.

```
#include<bits/stdc++.h>
 using namespace std;
 class Address
 {
     public:
         int houseNo;
         string colony,city, state;
         Address(int hno, string colony, string city, string state)
 {
         this->houseNo = hno;
         this->colony=colony;
         this->city = city;
         this->state = state;
   }
 };
 class Person
 {
     private:
         Address* address;
     public:
         string name;
         Person(string name, Address* address)
 {
         this->name = name;
```

```
        this->address = address;
    }
 void display()
 {
    cout<< name<< " "<< " "<< address->houseNo<<" "<<address-> colony<<" "
<<address->city<< " "<<address->state<<endl;
 }
 };
 int main(void)
 {
 Address add1= Address(007 ,"Ampitheatre Park","China
Gate","San        Fransisco, CA");
 Person p1 = Person("Andrew's Address:->",&add1);
 Person p2 = Person("Stacy's Address: ->",&add1);
 p1.display();
 p2.display();
 }
 return 0;
```

Output:



```
Andrew's Address:->  7 Ampitheatre Park China Gate San Fransisco, CA
Stacy's Address: ->  7 Ampitheatre Park China Gate San Fransisco, CA
```

Tasks

Problem 1:

A class named **Processor** has

- Two attributes i.e. **processName** and **price**
- A **parameterized constructor** to initialize attributes with user-defined values

Class MainMemory consists of

- Two attributes i.e. **size** and **price**
- A **parameterized constructor** to initialize attributes with user-defined values

Class MotherBoard has

- a data member named **compName** of type string
- a **no-argument** constructor to initialize with default name **intel**

Design a class named Computer that includes

- A data member named **proc** of type **Processor**
- A data member named **ram** of type **MainMemory**
- A data member named **mboard** of type **MotherBoard**
- A **parameterized constructor** that accept two arguments of type **Processor** and **MainMemory** to initialize members of these types. Moreover, within this constructor, instantiate object of **MotherBoard** to initialize **mboard** data field.

Write **main()** in a way that it clearly describes aggregation and composition relationships between objects of implemented classes.

Problem 2:

Consider six classes i.e. **Person, Professor, Researcher, Department, Laboratory, and University** having following specifications.

Class **University** has

- Two attributes of type string i.e. universityName and location
- An attribute named dept of type Department

Class **Department** has

- Two attributes i.e. **deptID, deptName**
- A **two-argument constructor** to initialize data fields with user-defined values
- A member function **display()** to show all attribute values

Class **Laboratory** contains

- Two attributes i.e. **labID** and **experimentNo**
- A **two-argument constructor** to initialize data member with user-defined values

Class **Person** has

- Two attributes i.e. **name** and **age**
- A **parameterized constructor** to initialize attributes with user-defined values
- A member function **display()** to show its attribute values

Class **Professor** is derived from class **Person** and has

- A data field named **profName** of type **string**
- A data field named **dept** of type **Department**
- A **two-argument constructor** to initialize both attributes of user-defined values

Class **Researcher** is derived from class **Professor** and has

- An additional attribute named **lab** of type **Laboratory**
- A constructor to initialize **lab** with **user-defined value**

Implement all these classes while illustrating the concept of aggregation and composition in terms of ownership and life-cycle.

Write following functions.

1) Write appropriate getter setter function for each Class.
2) Add/delete/update Department in University class