**Academic Year: 2020**

**Semester: 2nd**

**Course Code: CS-241L**

**Course Title: Object Oriented Programming**

**CS-241L Object Oriented Programming Lab 07**

**Type of Lab: Open Ended Weightage: 10%**

**CLO 2:** Implement abstraction and encapsulation to develop reusable classes for objects of real world problems.

|  |  |  |  |
| --- | --- | --- | --- |
| State understand the concept of inheritance and Aggregation | **Cognitive/Understanding** | CLO 2 | Rubric A |

**Rubric A: Cognitive Domain**

**Evaluation Method: GA shall evaluate the students for Question according to following rubrics.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CLO** | **0** | **1** | **2** | **3** | **4** | **5** |
| CLO2 | Unable to understand and implement | Student understand inheritance concepts | Student understand function overloading | Student implement inheritance and function overloading | Understand and implemented half problem sets | Understand and implemented complete problem sets |

# 

**Lab 07**

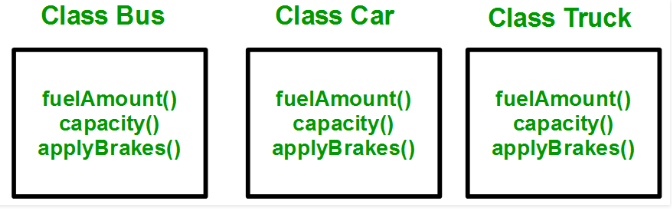
**BS-Computer Science**

**Object Oriented Programming**

**Target:** Inheritance, Access Specifiers and Aggregation

**Inheritance:** The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object Oriented Programming.  
Derived Class: The class that inherits properties from another class is called Sub class or Derived Class.  
Base Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Example:



# Implementing Inheritance

# 

# Example:

using namespace std;

//Base class

class Parent

{

    public:

      int id\_p;

};

// Sub class inheriting from Base Class(Parent)

class Child : public Parent

{

    public:

      int id\_c;

};

//main function

int main()

   {

        Child obj1;

        // An object of class child has all data members

        // and member functions of class parent

        obj1.id\_c = 7;

        obj1.id\_p = 91;

        cout << "Child id is " <<  obj1.id\_c << endl;

        cout << "Parent id is " <<  obj1.id\_p << endl;

        return 0;

# }

# Access Modifiers:

1. Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Example:

class A

{

public:

    int x;

protected:

    int y;

private:

    int z;

};

class B : public A

{

    // x is public

    // y is protected

    // z is not accessible from B

};

class C : protected A

{

    // x is protected

    // y is protected

    // z is not accessible from C

};

class D : private A    // 'private' is default for classes

{

    // x is private

    // y is private

    // z is not accessible from D

**Aggregation:**

In the previous lesson on [Composition](https://www.learncpp.com/cpp-tutorial/102-composition/), we noted that object composition is the process of creating complex objects from simpler ones. We also talked about one type of object composition, called composition. In a composition relationship, the whole object is responsible for the existence of the part.

In this lesson, we’ll take a look at the other subtype of object composition, called aggregation

Aggregation in C++ (commonly called as a has-a relationship), is a process in which one class defines a second class as an entity reference. It is a method of reusability of classes. In the simplest possible terms, it is when a class has an object of the other class.

**To qualify as an aggregation,** a whole object and its parts must have the following relationship:

* The part (member) is part of the object (class)
* The part (member) can belong to more than one object (class) at a time
* The part (member) does not have its existence managed by the object (class)
* The part (member) does not know about the existence of the object (class)

Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

For example, consider the relationship between a person and their home address. In this example, for simplicity, we’ll say every person has an address. However, that address can belong to more than one person at a time: for example, to both you and your roommate or significant other. However, that address isn’t managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don’t know what people live there. Therefore, this is an aggregate relationship.

Alternatively, consider a car and an engine. A car engine is part of the car. And although the engine belongs to the car, it can belong to other things as well, like the person who owns the car. The car is not responsible for the creation or destruction of the engine. And while the car knows it has an engine (it has to in order to get anywhere) the engine doesn’t know it’s part of the car.

When it comes to modeling physical objects, the use of the term “destroyed” can be a little dicey. One might argue, “If a meteor fell out of the sky and crushed the car, wouldn’t the car parts all be destroyed too?” Yes, of course. But that’s the fault of the meteor. The important point is that the car is not responsible for destruction of its parts (but an external force might be).

We can say that aggregation models “has-a” relationships (a department has teachers, the car has an engine).

Because aggregations are similar to compositions in that they are both part-whole relationships, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our parts to the composition using normal member variables (or pointers where the allocation and deallocation process is handled by the composition class).

In an aggregation, we also add parts as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregation usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these parts exist outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed (but not deleted). Consequently, the parts themselves will still exist.

Let’s take a look at a Teacher and Department example in more detail. In this example, we’re going to make a couple of simplifications: First, the department will only hold one teacher. Second, the teacher will be unaware of what department they’re part of.

**Example 1:**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Teacher**

**{**

**private:**

**const string m\_name{};**

**public:**

**Teacher(const string& name): m\_name{name}**

**{**

**}**

**const string& getName() const {return m\_name;}**

**};**

**class Department**

**{**

**private:**

**const Teacher& m\_teacher;** // This dept holds only one teacher for simplicity, but it could hold many teachers

**public:**

**Department(const Teacher& teacher): m\_teacher{teacher}**

**{**

**}**

**};**

**int main()**

**{**

// Create a teacher outside the scope of the Department

**Teacher t1("Ali");** // create a teacher

**{**

// Create a department and use the constructor parameter to pass the teacher to it.

**Department department (t1);**

**}**

// department goes out of scope here and is destroyed

// Ali still exists here, but the department doesn't

**cout << t1.getName() << " still exists!\n";**

**return 0;**

**}**

In this case, Ali is created independently of department, and then passed into department‘s constructor. When department is destroyed, the m\_teacher reference is destroyed, but the teacher itself is not destroyed, so it still exists until it is independently destroyed later in **main().**

**Example 2:**

**Consider the class** ‘address,’ along with a group of classes of professions, like ‘student,’ ‘scientist,’ ‘programmer’ etc. Each of the objects of the classes would have an attribute for their address. Hence, each class would contain an object of the ‘address’ class. This, given below, is the basic concept’s usage.

**class address{**

**public:**

**string city, state, locality;**

**address( city\_c, state\_c, locality\_c)**

**{ city = city\_c; state = state\_c; locality = locality\_c; }**

**};**

**class student{**

**address\*adrs;**

**public:**

**int id;**

**string name;**

**student( int i, string nm, address\*ad)**

**{ id = i; name = nm; adrs = ad; }**

**void display()**

**{ cout<<id<< " "<<name<< " "<<adrs->locality<< " "<<adrs->city<< " "<<adrs->state;}**

**};**

**void main()**

**{**

**address a1= address("C-146, Sec-15","CNT","LHR");**

**student s1 = student(101,"Ali",&a1);**

**s1.display();**

**}**

**Output:**

101 Ali C-146, Sec-15 CNT LHR

This example shows you how it is possible to relate two classes using the aggregation method.  Characteristics of the embedded class can also be accessed using pointers, as is shown in the example.

**Summarizing composition and aggregation**

**Compositions:**

* Typically use normal member variables
* Can use pointer members if the class handles object allocation/deallocation itself
* Responsible for creation/destruction of parts

**Aggregations:**

* Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
* Not responsible for creating/destroying parts

It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some parts but not others. For example, our Department class could have a name and a Teacher. The name would probably be added to the Department by composition, and would be created and destroyed with the Department. On the other hand, the Teacher would be added to the department by aggregation, and created/destroyed independently.

While aggregations can be extremely useful, they are also potentially more dangerous, because aggregations do not handle deallocation of their parts. Deallocations are left to an external party to do. If the external party no longer has a pointer or reference to the abandoned parts, or if it simply forgets to do the cleanup (assuming the class will handle that), then memory will be leaked.

For this reason, compositions should be favored over aggregations.

**Tasks**

Use all examples of aggregation and implement in c++. Write an appropriate and meaningful code for the below examples.

1. Team and Players
   1. If team is disbanded then player still exist
2. SmartPhone , Battery and IMEI (composition and aggregation)
   1. If smartPhone is destroyed IMEI will destroy but not battery.
3. Book and Chapters
   1. If book is destroyed then chapters will also.

**End**