Lecture review

INHERITANCE

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).
Base Class: A base class is the class from which features are to be inherited into another class.
Derived Class: A derived class is the one which inherits features from the base class. It can have additional properties and methods that are not present in the parent class that distinguishes it and provides additional functionality.
Real World Example: A real world example of inheritance constitutes the concept that children inherit certain features and traits from their parents. In addition, children also have their unique features and traits that distinguishes them from their parents.
Basic syntax for Inheritance:

```
class derived-class-name : access base-class-name {
//  body of class
};
```

TYPES OF INHERITANCE BASED ON BASE CLASS ACCESS CONTROL

There are three types of inheritance with respect to base class access control:

- Public
- Private
- Protected

Public Inheritance

With public inheritance, every object of a derived class is also an object of that derived class's base class. However, base class objects are not objects of their derived classes.

Is – A Relationship

Inheritance is represented by an is-a relationship which means that an object of a derived class also can be treated as an object of its base class for example, a Car is a Vehicle, so any attributes and behaviors of a Vehicle are also attributes and behaviors of a Car.

```
Class (name of the derived class) : public (name of the base class)
                            class Car : public Vehicle
```

Base Class Access Control for Public, Private and Protected:

Visibility of Base Class Members	**Types of Inheritance**		
	Public Inheritance	***Private Inheritance***	***Protected Inheritance***
Public	Public in derived class	Private in derived class	Protected in derived class
Private	Hidden in derived class	Hidden in derived class	Hidden in derived class
Protected	Protected in derived class	Hidden in derived class	Protected in derived class

TYPES OF INHERITANCE BASED ON DERIVED CLASSES

Inheritance based on derived classes can be categorized as follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

In this type of inheritance there is one base class and one derived class. As shown in the figure below, in single inheritance only one class can be derived from the base class. Based on the visibility mode used or access specifier used while deriving, the properties of the base class are derived.

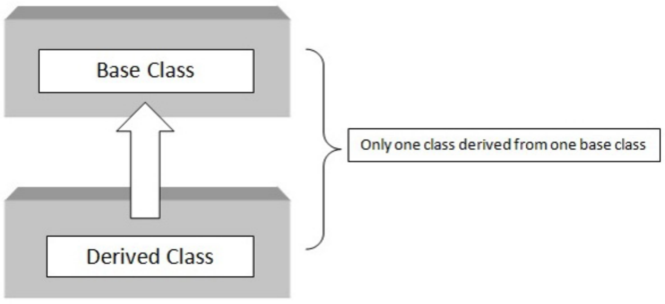


Figure 1: Single Inheritance

Syntax for single Inheritance:

```
 class A   {    // base class

           // body of the class
        };
class B : acess_specifier A   {// derived

           // body of the class
           };
```

Coding Example:

```
#include <iostream>
using namespace std;
class base    {//single base class

   public:
     int x;
   void getdata() {
     cout << "Enter the value of x = "; cin >> x;
   }
 };
class derive : public base {  //single derived class

   private:
    int y;
   public:
     void readdata() {
     cout << "Enter the value of y = "; cin >> y;
   }
     void product() {
     cout << "Product = " << x * y;
   }
 };

 int main() {
    derive a;     //object of derived class
    a.getdata();
    a.readdata();
    a.product();
    return 0;
 }
```

```
Sample Run
Enter the value of x = 3
Enter the value of y = 4
Product = 12
```

Multiple Inheritance

In multiple inheritance, a class is derived from two or more base classes. In multiple inheritance a derived class has more than one base class. As shown in the figure below, class C is derived from two base classes A and B.

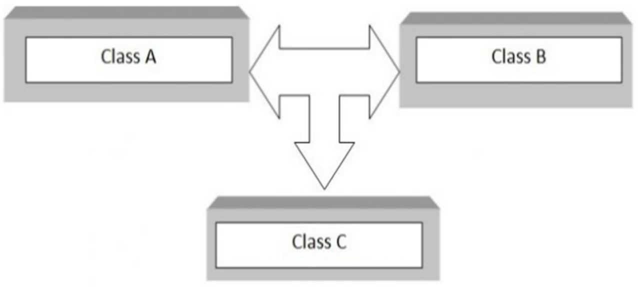


Figure 2: Multiple inheritance

Syntax for multiple inheritance:

```
      class A  { // base class

                // body of the class
                };
      class B  {// base class

                // body of the class
            };
      class C : acess_specifier A,  acess_specifier B  {   // derived class

                // body of the class
            };
```

Example code for multiple Inheritance:

```
#include<iostream>
using namespace std;
class A // base class
{
    public:
    int x;
    void getx()
    {
        cout << "enter value of x: "; cin >> x;
    }
};
class B // base class
{
    public:
    int y;
    void gety()
    {
        cout << "enter value of y: "; cin >> y;
    }
};
class C : public A, public B   //C is derived from class A and class B
{
    public:
    void sum()
    {
        cout << "Sum = " << x + y;
    }
};
int main()
{
     C obj1; //object of derived class C
     obj1.getx();
     obj1.gety();
     obj1.sum();
     return 0;
}       //end of program
```

```
Sample Run
enter value of x: 5
enter value of y: 4
Sum = 9
```

Multilevel Inheritance

If a class is derived from another derived class then it is called multilevel inheritance, so in multilevel inheritance, a class has more than one parent class. As shown in the figure below, class C has class B and class A as parent classes. As in other inheritance, based on the visibility mode used or access specifier used. while deriving, the properties of the base class are derived. Access specifier can be private, protected or public.

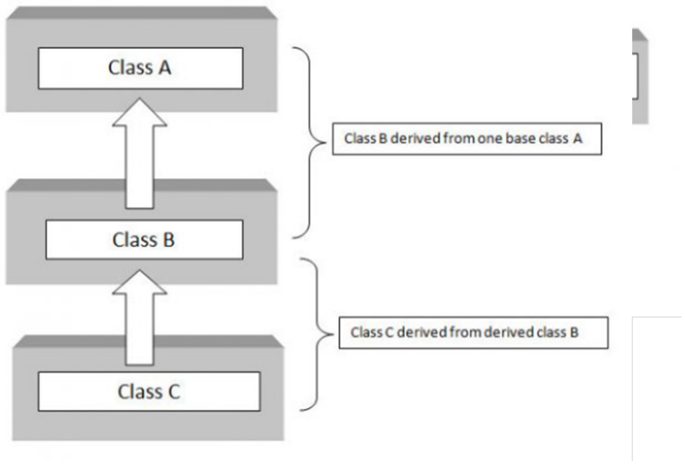

Class A
Class B
Class C
Class B derived from one base class A
Class C derived from derived class B

Figure 3: Multilevel inheritance

Syntax for multilevel Inheritance:

```
     class A   { // base class

                 // body of the class
                 };
     class B : acess_specifier A // derived class
    {
     // body of the class
    };
    class C : acess_specifier B     // derived from class B
    {
          // body of the class
    };
```

Example code for multilevel Inheritance:

```
#include <iostream>
using namespace std;
class base //single base class
{
    public:
    int x;
    void getdata()
    {
        cout << "Enter value of x= "; cin >> x;
    }
};
class derive1 : public base // derived class from base class
{
    public:
```

```
    int y;
    void readdata()
    {
        cout << "\nEnter value of y= "; cin >> y;
    }
};
class derive2 : public derive1   // derived from class derive1
{
    private:
    int z;
    public:
    void indata()
    {
        cout << "\nEnter value of z= "; cin >> z;
    }
    void product()
    {
        cout << "\nProduct= " << x * y * z;
    }
};
int main()
{
     derive2 a;      //object of derived class
     a.getdata();
     a.readdata();
     a.indata();
     a.product();
     return 0;
}                //end of program
```

```
Sample Run
Enter value of x= 2
Enter value of y= 3
Enter value of z= 3
Product= 18
```

Hierarchical Inheritance

When several classes are derived from a common base class it is called as hierarchical inheritance. In C++ hierarchical inheritance, the feature of the base class is inherited onto more than one sub-class. For example, a car is a common class from which Audi, Ferrari, Maruti etc can be derived. As shown in the figure below, in C++ hierarchical inheritance all the derived classes have a common base class. The base class includes all the features that are common to derived classes

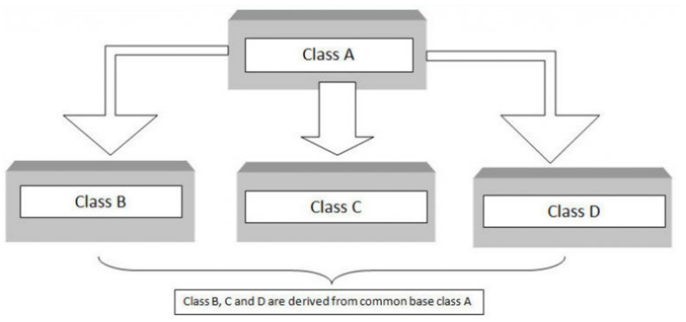


Figure 4: Hierarchical Inheritance

Syntax for hierarchical Inheritance:

```
   class A   { // base class

                // body of the class
                };
   class B : acess_specifier A { // derived class from A

                // body of the class
                };
   class C : acess_specifier A  {   // derived class from A

                // body of the class
            };
   class D : acess_specifier A  {   // derived class from A

                // body of the class
            };
```

Example code for hierarchical Inheritance:

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

class A //single base class
{
    public:
    int x, y;
    void getdata()
    {
        cout << "\n Enter value of x and y:\n"; cin >> x >> y;
    }
};
class B : public A //B is derived from class base
{
    public:
    void product()
    {
        cout << "\n Product= " << x * y;
    }
};
class C : public A //C is also derived from class base
{
    public:
    void sum()
    {
        cout << "\n Sum= " << x + y;
    }
};
int main()
{
    B obj1;          //object of derived class B
    C obj2;          //object of derived class C
    obj1.getdata();
    obj1.product();
    obj2.getdata();
    obj2.sum();
    return 0;
}  //end of program
```

```
Sample Run
Enter value of x and y:
2
3
Product= 6
Enter value of x and y:
2
3
Sum= 5
```

Hybrid Inheritance

The inheritance in which the derivation of a class involves more than one form of any inheritance is called hybrid inheritance. Basically C++ hybrid inheritance is combination of two or more types of inheritance. It can also be called multi path inheritance. The figure below shows the hybrid combination of single inheritance and multiple inheritance. Hybrid inheritance is used in a situation where we need to apply more than one inheritance in a program.

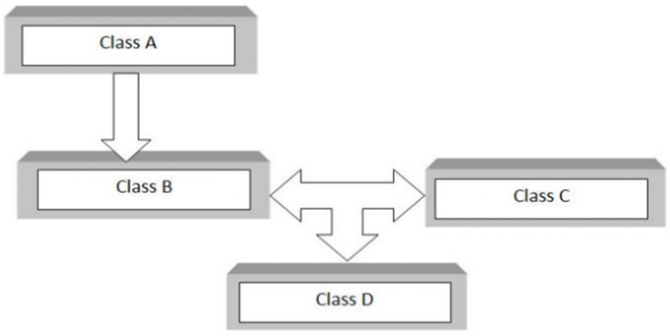


Figure 5: Hybrid Inheritance

Syntax for hybrid Inheritance:

```
class A   // base class
        {
        // body of the class
        };
class B : public A
    {
       // body of the class
        };
class C
    {
          // body of the class
    };
class D : public B, public C
    {
          // body of the class
    };
```

Example code for hybrid Inheritance:

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

class A
{
    public:
    int x;
};
class B : public A
{
    public:
    B()      //constructor to initialize x in base class A
    {
       x = 10;
    }
};
class C
 {
    public:
    int y;
    C()   //constructor to initialize y
    {
        y = 4;
        }
};
class D : public B, public C   //D is derived from class B and class C
{
    public:
    void sum()
    {
        cout << "Sum= " << x + y;
    }
};

int main()
{
       D obj1;          //object of derived class D
    obj1.sum();
    return 0;
}                   //end of program
```

```
Sample Run
Sum= 14
```

Lab exercises

Inheritance

Exercise 1 ....

A school library wants to organize its library system by categorizing books according to their genre. They need an automated system that will allow them to input the details of the books that are in their library. To do this, you need to implement a program that contains a base class called Books that will contain a data member to store the genre of the book. Derive two other classes from the base class and name them accordingly. Each of these two classes will hold details about a book from a specific genre of your choice such as Novel, Narrative, Mystery and so on. The derived class will contain data members to store the title and the author of the book. Display the details of each book along with their genre.

Exercise 2 ....

A vehicle company is deciding to hire a programmer to develop a system that will allow the company to enter the details of the vehicles sold by them. As a programmer, you need to implement a program that contains a base class called **Vehicles** that contains a data member to store the price of the vehicles. Derive two other classes named as **Car** and **Motorcycle**.

- The Car class will contain data members to store details that include seating capacity, number of doors and fuel type (diesel or petrol).
- The Motorcycle class will contain data members to store details such as the number of cylinders, the number of gears and the number of wheels.

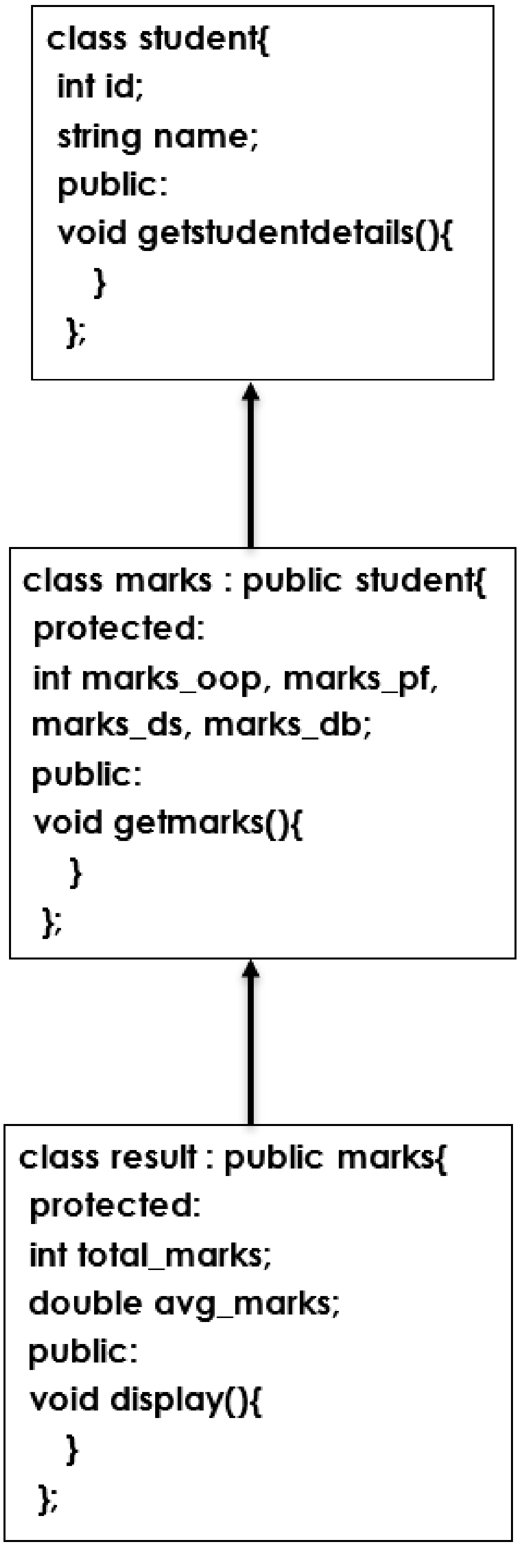
Derive another subclass named as Audi of Car and Yamaha of Motorcycle.

- The Audi class will contain a data member to store the model type.
- The Yamaha class will contain a data member to store the make – type. Display the details of an Audi car (price, seating capacity, number of doors, fuel type, model type) and the details of the Yamaha motorcycle (price, number of cylinders, number of gears, number of wheels, make – type).

Exercise 3 ....

A university is deciding to upgrade its system. In order to upgrade, you need to implement the following scenario as shown in the figure: Note the following:

- The class student has a function that displays all the information about the student.
- Class marks is derived from class student and has a function that displays all the marks obtained in the courses by the students.
- Class result is derived from class marks. This class has a function that calculates the total marks and then calculates the average marks. It then displays both the total and the average marks.
- In the main function you are required to do the following:
- Create an object of the result class.
- Then display the student details, the marks obtained in each courses and the total and the average marks.



```
class student{
 int id;
 string name;
 public:
 void getstudentdetails(){
    }
 };
```

```
class marks : public student{
 protected:
 int marks_oop, marks_pf,
 marks_ds, marks_db;
 public:
 void getmarks(){
    }
 };
```

```
class result : public marks{
 protected:
 int total_marks;
 double avg_marks;
 public:
 void display(){
    }
 };
```

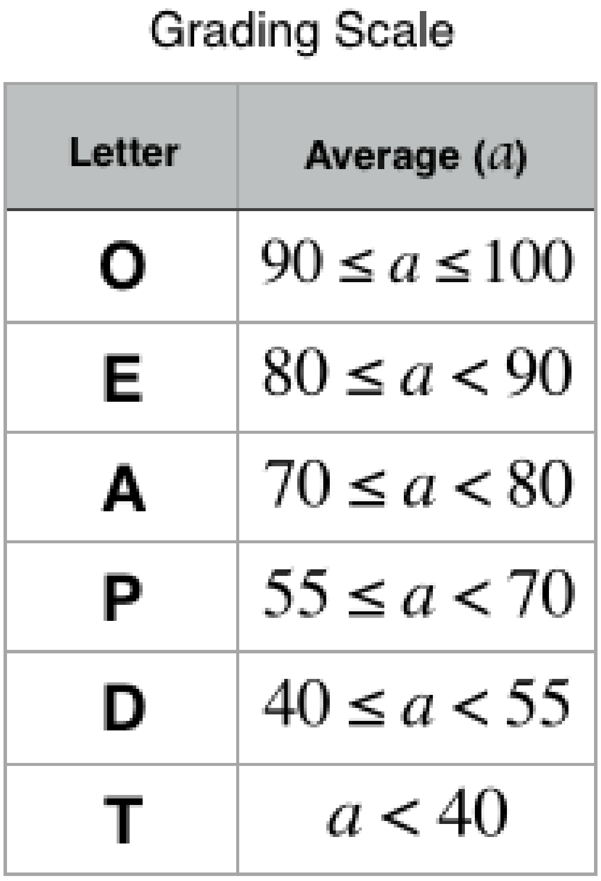
Figure 6: Sample code

Exercise 4 . . . .

You are given two classes, Person and Student, where Person is the base class and Student is the derived class. Completed code for Person and a declaration for Student are provided for you in the editor. Observe that Student inherits all the properties of Person. Complete the Student class by writing the following:
A Student class constructor, which has 4 parameters:

1. A string, **firstName**.
2. A string, **lastName**.
3. An integer, **idNumber**.
4. An integer array (or vector) of test scores,**Scores** .

A char calculate() method that calculates a Student object's average and returns the grade character representative of their calculated average:



Grading Scale

Letter	Average (a)
O	$90 \leq a \leq 100$
E	$80 \leq a < 90$
A	$70 \leq a < 80$
P	$55 \leq a < 70$
D	$40 \leq a < 55$
T	$a < 40$

Figure 7: Grading Scale

```
#include <iostream>
#include <vector>

using namespace std;
```

```
class Person{
    protected:
        string firstName;
        string lastName;
        int id;
    public:
        Person(string firstName, string lastName, int identification){
            this->firstName = firstName;
            this->lastName = lastName;
            this->id = identification;
        }
        void printPerson(){
            cout<< "Name: "<< lastName << ", "<< firstName <<"\nID: "<< id << "\n";
        }

};

class Student :  public Person{
    private:
        vector<int> testScores;
    public:
        /*
        *   Class Constructor
        *
        *   Parameters:
        *   firstName - A string denoting the Person's first name.
        *   lastName - A string denoting the Person's last name.
        *   id - An integer denoting the Person's ID number.
        *   scores - An array of integers denoting the Person's test scores.
        */
        // Write your constructor here

        /*
        *   Function Name: calculate
        *   Return: A character denoting the grade.
        */
        // Write your function here
};

int main() {
    string firstName;
    string lastName;
    int id;
    int numScores;
    cin >> firstName >> lastName >> id >> numScores;
    vector<int> scores;
    for(int i = 0; i < numScores; i++){
        int tmpScore;
        cin >> tmpScore;
        scores.push_back(tmpScore);
    }
    Student* s = new Student(firstName, lastName, id, scores);
    s->printPerson();
    cout << "Grade: " << s->calculate() << "\n";
    return 0;
}
```

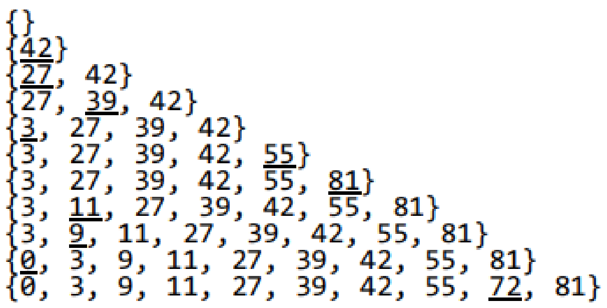
Exercise 5 . . . .

Imagine there is an existing class named **ArrayStack** that has been implemented to represent a stack of integers using an array as its internal storage. The **ArrayStack** class provides essential functionalities such as pushing, popping, peeking, checking if it's empty, and handling resizing when necessary.
Now, let's define a new class named **SortedStack** that inherits from **ArrayStack**. This new class ensures that the stack is always stored in a sorted, non-decreasing order, regardless of the order in which items are pushed onto the stack. The **SortedStack** class should provide the same member functions as the **ArrayStack** superclass. Importantly, your code for **SortedStack** must work seamlessly with the existing **ArrayStack** without any modifications.

For instance, if we add the following elements to an empty **SortedStack**:

```
42, 27, 39, 3, 55, 81, 11, 9, 0, 72
```

The stack's state before and after adding each element, in bottom-to-top (left-to-right) order, should be appropriately managed by the **SortedStack** class as shown in following figure.



```
{}
{42}
{27, 42}
{27, 39, 42}
{3, 27, 39, 42}
{3, 27, 39, 42, 55}
{3, 27, 39, 42, 55, 81}
{3, 11, 27, 39, 42, 55, 81}
{3, 9, 11, 27, 39, 42, 55, 81}
{0, 3, 9, 11, 27, 39, 42, 55, 81}
{0, 3, 9, 11, 27, 39, 42, 55, 72, 81}
```

Figure 8: Grading Scale

Exercise 6 . . . .

Consider a company named "**PakExports International**," engaged in the export trade of clothing from Pakistan to three continents: Africa, Asia, and Europe. The company employs three types of transportation methods, namely Land transport, Sea transport, and Air transport, to facilitate the shipment of clothing items. It utilizes Land and Sea transport services for shipments from Pakistan to South East Asia, the Middle East, Far East, and Africa. Additionally, Sea and Air transport services are employed for deliveries to European countries. The company also offers rental services for fitting rooms and exhibition halls to accommodate small and large orders from customers.

Tasks to be performed include:

- Identify the primary objects (entities) in the described system.
- Determine the essential attributes and functions associated with each object, limiting the mention to three attributes and one function for each class.
- Establish relationships between these objects.
- Implement this scenario after completing aforementioned points. The objective is to represent the system by showcasing the identified objects, their interconnections, along with their respective attributes and functions.
  As the user for [User Input] for attributes and then perform the identified functions accordingly.
  Hint:
  FittingRoomRental **Class** (inherits from FacilityRental):

  Additional Attributes: roomSize (e.g., small, medium, or large) **User Input**
  numberofCustomers (number of customers the fitting room can accommodate) **User Input**"

```
/* Sample code for FittingRoomRental */
class FacilityRental {
public:
    string facilityType;
    string availabilityStatus;
    int rentalDuration;

    FacilityRental(string type, string status, int duration)
        : facilityType(type), availabilityStatus(status),
          rentalDuration(duration) {}

    void reserveFacility() {
        cout << "Facility reserved for " << rentalDuration << " days." << endl;
    }
};

class FittingRoomRental : public FacilityRental {
public:
    string roomSize;
    int numberOfCustomers;

    FittingRoomRental(string type, string status, int duration, string size,
        int customers)
        : FacilityRental(type, status, duration), roomSize(size),
          numberOfCustomers(customers) {}

    // Additional function specific to FittingRoomRental
    void provideFittingExperience() {
        cout << "Enjoy your fitting experience in a " << roomSize << " fitting
            room." << endl;
    }
};
```