Lab 8 – Inheritance

Lab Outcomes:

- **Understanding Inheritance Types**: Gain a clear understanding of different types of inheritance (single, hierarchical, multi-level, multiple, and hybrid inheritance) and their applications in creating reusable, organized, and efficient class hierarchies.
- **Exploring Access Specifiers with Inheritance**: Learn how public, protected, and private inheritance affect the accessibility of base class members in derived classes, and understand the role of the `protected` specifier for managing access within inheritance hierarchies.
- **Constructor Execution Order in Inheritance**: Observe the order of constructor execution in single and multiple inheritance scenarios, and gain hands-on experience with explicitly calling base class constructors using initializer lists in derived classes.
- **Applying Function Overloading and Overriding**: Develop skills in implementing function overloading within a single class and function overriding between base and derived classes, recognizing the importance of function signatures in distinguishing overloaded and overridden functions.
- **Constructor Initializer Lists in Derived Classes**: Practice using constructor initializer lists to efficiently initialize base class members in derived classes, especially when working with parameterized constructors, ensuring correct constructor invocation and reducing redundancy.

Inheritance

Inheritance is one of the most important building blocks of OOP. The concept of reusable classes is helpful to manage objects because occasionally it happens that a class may have some features that it can derive from already existing class (es). Hence, with inheritance, it is sometimes crucial to build multi-level class hierarchies or to derive class (es) from multiple existing classes.

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

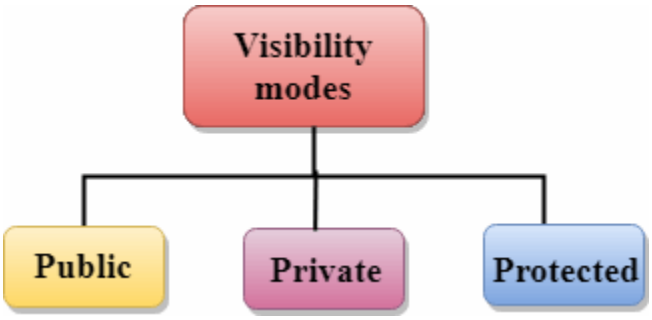
Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

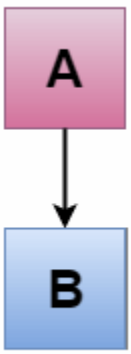
Visibility modes can be classified into three categories:



- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Type of Inheritance

1. Single Inheritance: a single child class derived from a single base class



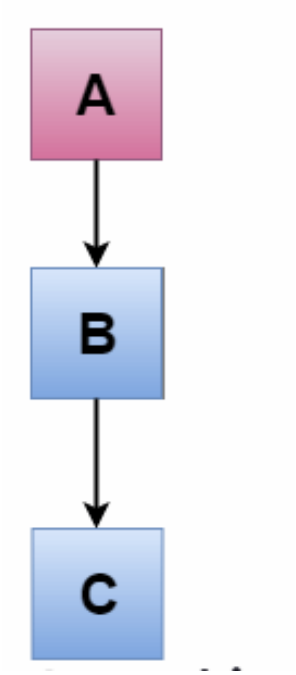
Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

2. Multi-level Inheritance: a single derived class inherited from another derived class



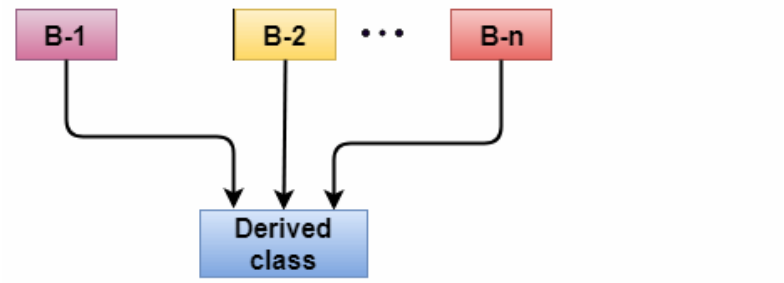
C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
class Animal {
  public:
void eat() {
   cout<<"Eating..."<<endl;
}
  };
  class Dog: public Animal
  {
     public:
    void bark(){
   cout<<"Barking..."<<endl;
    }
  };
  class BabyDog: public Dog
  {
     public:
    void weep() {
   cout<<"Weeping...";
    }
  };
int main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
```

3. Multiple Inheritance: a single derived class inherited from more than one base classes



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
   // Body of the class;
}
```

Example

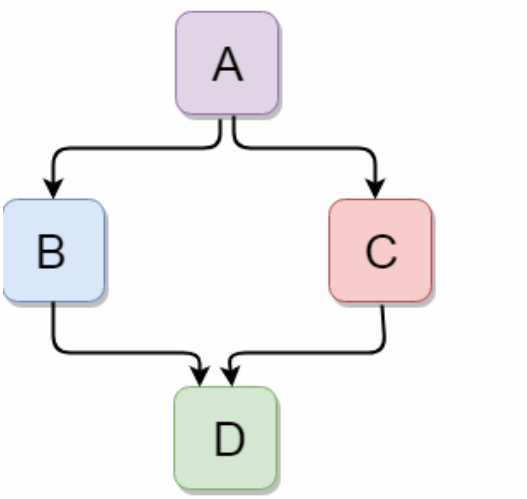
```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};

class B
{
   protected:
   int b;
   public:
   void get_b(int n)
   {
      b = n;
   }
};
class C : public A,public B
{
  public:
   void display()
   {
      std::cout << "The value of a is : " <<a<< std::endl;
      std::cout << "The value of b is : " <<b<< std::endl;
      cout<<"Addition of a and b is : "<<a+b;
   }
};
```

```
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

4. Hybrid Inheritance: Hybrid inheritance is a combination of more than one type of inheritance.



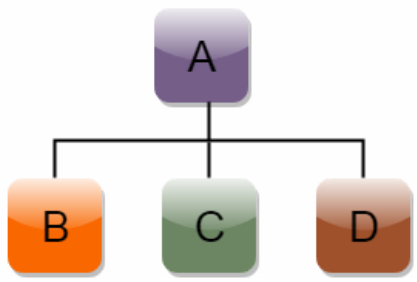

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
      std::cout << "Enter the value of 'a' : " << std::endl;
      cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
```

```
    {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

5. Hierarchical Inheritance: multiple child classes derived from a single base class



Syntax of Hierarchical inheritance:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

To grasp the concept and need of multi-level and multiple inheritance, consider the following class hierarchy.

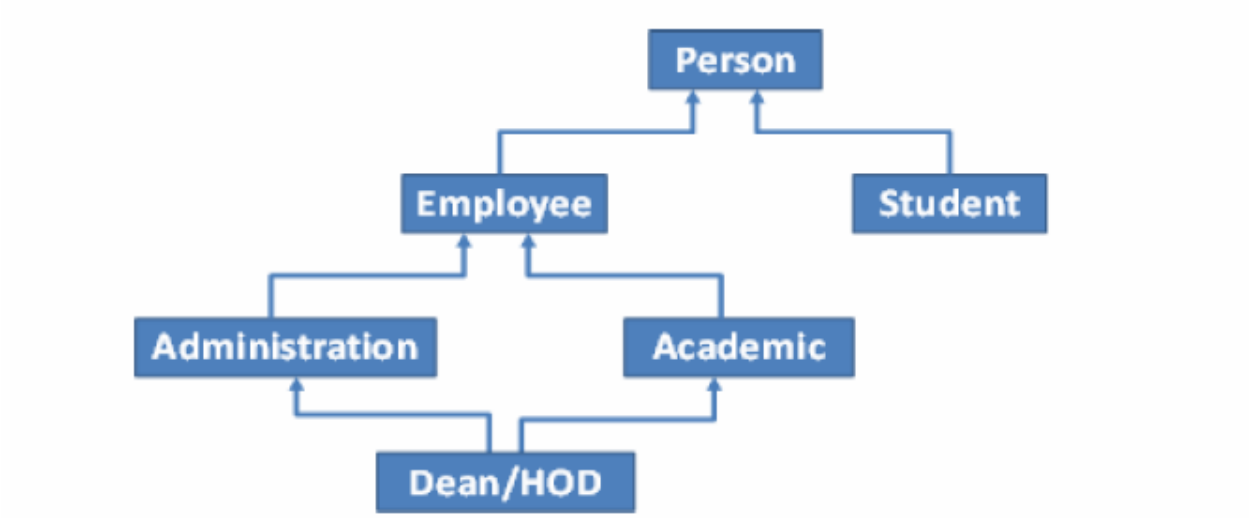


Figure 1: Class Hierarchies showing Multi-level and Multiple Inheritance

A person can be an employee or a student. An employee may have rights of admin officer or of academic officer. These class hierarchies represent multi-level inheritance. However, a Dean or Head of Department (HOD) may have rights to modify the status already defined by an admin or academic officer. This type of relationship between classes is an example of multiple inheritance. C++ facilitates users while supporting the implementation of both multi-level and multiple inheritance. There is no limitation on the number of levels in multi-level inheritance and there is no limitation on the number of parent classes from which a child can be derived. Nevertheless, as a good practice, it is recommended to restrict levels and number of base classes ***to two in multi-level and multiple inheritance.***

Constructors in Derived Classes

In C++, constructors in derived classes are called after the constructors of their base classes have completed execution. This means that when an object of a derived class is created, the constructors of all its base classes are invoked first, followed by the constructor of the derived class itself.

```
#include <iostream>
Using namespace std;
class Base {
public:
   Base() {
      cout << "Base constructor\n";
   }
};
class Derived : public Base {
public:
   Derived() {
      cout << "Derived constructor\n";
   }
};
int main() {
   Derived d;
   return 0;
}
```

Output

```
Base constructor
Derived constructor
```

In this example, when an object **d** of the **Derived** class is created in the **main** function, it invokes the constructor of **Derived**. However, since **Derived** is derived from **Base**, the constructor of **Base** is called first, followed by the constructor of **Derived**.

Case1:

```
class B: public A{
//order of execution of constructor: first A() then B()
}
```

Case2:

class A: public B,public C{

//order of execution of constructor: first B() then C() and then A()

}

Case3:

class A: public B,virtual public C{

//order of execution of constructor: first C() then B() and then A()

}

Syntax for Explicitly calling Base class constructors from Derived class:

If you define parameterized constructors in the base classes, and if the derived class doesn't provide default values for all the parameters of the base class constructors, it becomes necessary to explicitly call the appropriate base class constructors in the initializer list of the derived class constructor.

Derived-constructor(datatype arg1, datatype arg2,.. datatype argn):Base(arg1):Base2(arg2){

//Derived-constructor's body

}

As long as you provide explicit instructions for constructing the base classes in the derived class constructor's initializer list, the compiler won't generate an error for the non-invoked constructors. It's up to you, as the programmer, to ensure that the appropriate constructors are called based on your design and requirements.

Constructor Initializer List

A constructor initializer list is a feature in many programming languages, including C++, that allows you to initialize the member variables of an object when it is created. It's a comma-separated list of initializations that follows the constructor's parameter list and is enclosed in parentheses.

Syntax:

Constructor (argument-list) : initialization-section

{

// Constructor Body

}

Example:

```
// Constructor initializer list
   MyClass(int x, int y) : a(x), b(y) {
      cout<<"a="<<a<<endl<<"b="<<b<<endl;
   }
```

Function Overloading and Function Overriding

Function overloading is the availability of various functions within a class that differ from each other in function signature i.e. various functions share same name with different parameter types or number of parameters. Inheritance is not necessarily required to overload functions within a program. On the other hand, in function overriding, there must exists inheritance relationship between classes (i.e. base and derived) and functions' signature are also required to be same in both parent and child class(es). However, derived class(es) redefine function(s) having signature similar to base class's function(s).
Following code provides an example where both concepts (function overloading and overriding) have been elaborated:

```
class Base

{
  protected:
        void myFunc()
        {
        cout<<"Base Class' Function";
        }
};
class Derived: public Base
{
 public:
        void myFunc()
        {
        cout<<"Derived Class' Function";
        }
        void myFunc(int a)
        {
        cout<<"Derived Class' Function with Parameter Value" <<a; }
};
```

- At first, the class *Derived* shows function overloading while containing two functions differ from each other in function signature (i.e. functions *myFunc()* and *myFunc(int)*).
- Secondly, it also shows function overriding as it holds two function implementations with same signature (i.e. besides *myFunc()* of its own, it also contains *myFunc()* inherited from the Base class.

```cpp
#include<iostream>
using namespace std;
class A{
    public:
        void f1(){
        }
        void f2(){
        cout<<"Inside A";
        }
};
class B:public A{
    public:
        void f1(){ //Method Overriding
        }
        void f2(int x){ // Method Hiding
        cout<<"Inside B";
        }
};
int main(){
    B obj;
//  obj.f2(); //error'
    obj.f2(4);// B
return 0;
}
```

Lab Tasks

Task#01

Design a class named ***Staff*** that includes

- A data member named ***staffID***
- A ***parameterized constructor*** to initialize ***staffID***
- A ***getter function*** to get the value of ***staffID***

Derive a class named ***Professor*** inherited from ***Staff*** class and contains

- Two additional data members i.e. ***departmentID*** and ***departmentName***
- A ***parameterized constructor*** to initialize its own data fields along with the inherited data field
- Two ***getter functions*** that return the ***departmentID*** and ***departmentName***, respectively

Derive a class named ***VisitingProfessor*** inherited from class ***Professor*** and has

- A data field named ***no_of_courses***
- A data field named ***salary_per_course***
- A function named ***totalSalary*** that returns total payment for all courses (i.e. ***no_of_courses * salary_per_course)***
- A member function named ***display*** to show total salary of visiting professor

Task#02

- **Create a base class named "Shape" and derive two child classes named "Circle" & "Rectangle", create functions: area() & Display() with same signature in each class (Method overriding), and the required attributes.**
- **In main function create obj of derived classes and call their respective functions function.**

Task#03

Create a class **Ride** with methods startRide() and endRide().
Create subclasses:

- **CarRide**
- **BikeRide**

Override endRide() so BikeRide **cannot end an active ride unless helmet verification is true**, while CarRide ends normally.

Lab #08 Marks distribution

	ER1	ER6	ER8

Task	3points	3 points	4 points

Lab # 08 Rubric Evaluation Guideline:

#	**Qualities & Criteria**	**0 < Poor <=0.5**	**0.5 < Satisfactory <= 1.5**	**1.5 < Excellent <=2**
ER1	**Task Completion**	Minimal or no program functionality was achieved.	Some tasks were completed, but the program has errors or incomplete functionalities.	All tasks were completed, and the program runs without errors.
#	**Qualities & Criteria**	**0 < Poor <=0.5**	**0.5 < Satisfactory <= 1.5**	**1.5 < Excellent <=2**
ER6	Program Output	Output is inaccurate or poorly presented.	Output is mostly accurate but may lack labels, captions, or formatting.	Output is clear, accurate, and well presented with labels, captions, and proper formatting.
#	**Qualities & Criteria**	**0 < Poor <= 1.5**	**1.5< Satisfactory <= 3**	**3< Excellent <= 4**
ER8	Question & Answer	Answers some questions but not confidently or based on lab task knowledge.	Answers most questions confidently and based on lab task knowledge.	Answers all questions confidently and demonstrates a deep understanding of the given lab task.