UNIT III Operator Overloading and Type Conversion & Inheritance: The Keyword Operator, Overloading Unary Operator, Operator Return Type, Overloading Assignment Operator (=), Rules for Overloading Operators, Inheritance, Reusability, Types of Inheritance, Virtual Base Classes, Object as a Class Member, Abstract Classes, Advantages of Inheritance, Disadvantages of Inheritance.

Introduction to Operator overloading

Operator overloading is one of the important and useful features of C++. We are familiar with function overloading in which multiple functions use the same name. The concept of operator overloading is somewhat similar to that of function overloading.

A symbol that is used to perform an operation is called an operator. It is used to perform an operation with constants and variables. A programmer cannot build an expression without an operator.

C++ frequently uses user-defined data types that are a combination of one or more basic data types. C++ has the facility to create/build user-defined data type. User-defined data types created from class or struct are nothing but a combination of one or more variables of basic data types. The compiler knows how to perform various operations using operators for the built-in types; however, for the objects those are instance of the class, the operation routine must be defined by the programmer.

For example, in traditional programming languages the operators such as +, −, <=, >=, etc. can be used only with basic data types such as int or float. The operator + (plus) can be used to perform addition of two variables, but the same is not applicable for objects. The compiler cannot perform addition of two objects. The compiler would throw an error if addition of two objects is carried out. The compiler must be made aware of the addition process of two objects. When an expression including operation with objects is encountered, a compiler searches for the definition of the operator, in which a code is written to perform an operation with two objects. Thus, to perform an operation with objects we need to redefine the definition of various operators. For example, for addition of objects A and B, we need to define operator + (plus). Redefining the operator plus does not change its natural meaning. It can be used for both variables of built-in data type and objects of user-defined data type.

Operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. Polymorphism permits to write multiple definitions for functions and operators. C++ has a number of standard data types such as int, float, char, etc. The operators +, −, *, and = are used to carry operations with these data types. Operator overloading helps the programmer to use these operators with the objects of classes. The outcome of operator overloading is that the objects can be used in as natural manner as the variables of basic data type. Operator overloading provides the capability to redefine the language in which the working operator can be changed.

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.

Operator that are not overloaded are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Restrictions on Operator Overloading in C++

- Following are some restrictions to be kept in mind while implementing operator overloading.
- Precedence and Associativity of an operator cannot be changed.
- Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Unary Operator overloading

- Operator overloading is a compile polymorphic technique where a single operator can perform multiple functionalities
- As a result, the operator that is overloaded is capable to provide special meaning to the user-defined data types as well.
- Here we can overload unary operators like + +,–, unary + and unary – to directly.

Syntax

```
return_type:: operator unary_operator_symbol(parameters)
{
        // function definition
}
```

Steps to perform unary operator ++ overloading

1. Declare the class with its variables and its Member function
2. Using the function get_data()to read two numbers
3. Define the function operator ++ to add the values
4. Define the function of the operator — to subtract values
5. Define the display function
6. Define the class object
7. Call the function get_data()
8. Call the function operator ++() by incrementing in the class object and call the function display()
9. Call the function operator –() by the decrementing in the class object and call the function display().

C++ program for Unary increment and decrement

```
#include<iostream>

using namespace std;

class incredecre

{

            public:

                        int a,b;

                        void  getinput();

                        void operator ++();

                        void operator --();

                        void  getoutput();

};

Void incredecre::getinput()

{

            cout<<"enter a value for a,b"<<endl;

            cin>>a>>b;

}

Void incredecre::operator ++()

{

            a=++a;

}

Void incredecre::operator --()

{

            b=--b;

}

Void incredecre::getoutput()

{
            cout<<a<<endl;

            cout<<b<<endl;

}

main()

{

            Incredecre obj;

            obj.getinput();

            ++obj;

            --obj;

            obj.getoutput();

}
```

C++ Program for unary minus

```
#include<iostream>

using namespace std;

class  unaryminus

{

        Private:

                int a;

        public:

                void details()

                {

                        a=10;

                }

                void operator -();

};

Void unaryminus::operator -()

{
        a=-a;

        cout<<a<<endl;

}

main()

{

        Unaryminus s1;

        unaryminus.details();

        -s1;

}
```

C++ program for unary plus

```
#include<iostream>

using namespace std;

class  unaryplus

{

        Private:

                int a;

        public:

                void getinput()

                {

                        cout<<"enter value";

                        cin>>a;

                }

                void operator +()

                {

                        a=+a;

                }

                void display()
                {
                        cout<<a;
                }
};
main()
{
        unaryplus o1;
        o1.getinput();
        +o1;
        o1.display();
}
```

Binary operator overloading

- Unary operators work on one operand while binary operators work on two operands.
- Similar to unary operator overloading, binary operators can also be overloaded.
- In binary operator overloading one operand is passed implicitly to operator function and one is passed explicitly to operator function.

C++ program for binary addition

```
#include<iostream>
using namespace std;
classnum
{
        private:
                int a,b;
        public:
                void input(void);
                void show(void);
                num operator +(num);
};
Void num::input()
```

```
{

        cout<<"\n Enter Values for a:";

        cin>>a>>b;

}

Void num::show()

{

        cout<<a<<b;

}

num num::operator +(num t)

{

        num tmp;

        tmp.a=a+t.a;

        tmp.b=b+t.b;

        return (tmp);

}

main()

{

        num x,y,z;

        cout<<"\n Object x";

        x.input();

        cout<<"\n Object y";

        y.input();

        z=X+Y;

        z.show();

        return 0;

}
```

C++ program for == operator using binary operator overloading

```
#include<iostream>

using namespace std;

class equalitycheck

{

        public:

                int a;

                void getinput()//input member function

                {

                        cin>>a;

                }

                void operator ==(equalitycheck t)

                {

                        if(a==t.a)

                        {

                                cout<<"objects are equal";

                        }

                        else

                        {

                                cout<<"objects are not equal";

                        }

                }

};

main()

{

        equalitycheck t1,t2;

        cout<<"enter a value for t1 object";
        t1.getinput();

        cout<<"enter a value for t2 object";

        t2.getinput();

        t1==t2;

}
```

Operator Return type

- In the last few examples we declared the operator() of void types, that is, it will not return any value.
- However, it is possible to return value and assign to it other object of the same type.
- The return value of operator is always of class type, because the operator overloading is only for objects.
- An operator cannot be overloaded for basic data type. Hence, if the operator returns any value, it will be always of class type.

Write a program to return values from operator() function.

```
#include<iostream.h>

#include<conio.h>

Class plusplus

{

        private:

                int num;

        public :

                plusplus()

                {

                        num=0;

                }

                int getnum()

                {

                        return num;

                }

                plusplus operator ++ (int)
                {
                        plusplus tmp;
                        num=num+1;
                        tmp.num=num;
                        return tmp;
                }
};
void main()
{
        plusplus p1, p2;
        cout<<“\n p1=”<<p1.getnum();
        cout<<“\n p2=”<<p2.getnum();
        p1=p2++;
        cout<<endl<<“ p1=”<<p1.getnum();
        cout<<endl<<“ p2=”<<p2.getnum();
        p1++;
        cout<<endl<<“ p1=”<<p1.getnum();
        cout<<endl<<“ p2=”<<p2.getnum();
}
```

C++ program for increment operator using return type.

```
#include<iostream>
using namespace std;
class num
{
        private:
                int a;
        public:
```

```
                    void input(void);

                    void show(void);

                    num operator+(num);

};

Void num::input()

{

          cout<<"\n Enter Values for a:";

          cin>>a;

}

void num::show()

{

          cout<<a;

}

num num::operator +(num t)

{

          num tmp;

          tmp.a=a+t.a;

          return (tmp);

}

main()

{

          num X,Y,Z;

          cout<<"\n Object X";

          X.input();

          cout<<"\n Object Y";

          Y.input();

          Z=X+Y;
```

```
        Z.show();

        return 0;

}
```

C++ Overloading Assignment Operator

- C++ Overloading assignment operator can be done in object oriented programming.
- By overloading assignment operator, all values of one object (i.e instance variables) can be copied to another object.
- Assignment operator must be overloaded by a non-static member function only.

C++ program for assignment operator using constructors.

```
#include <iostream>

using namespace std;

class assignment

{

        private:

                int a;

                int b;

        public:

                assignment()

                {

                     a = 0;

                     b = 0;

                }

                assignment(int f, int i)

                {

                        a = f;

                        b = i;

                }

                void operator = (assignment D1 )

                {
                        a = D1.a;
                        b = D1.b;
                }
                 // method to display distance
                Void displayvalues()
                {
                        cout<<a<<b<<endl;
                }
};
int main()
{
        assignment D1(11, 10),D2;
         /*cout<< "First value : ";
        D1.displayDistance();
        cout<< "Second Distance :";
        D2.displayDistance();
         // use assignment operator*/
        D2 = D1;
        cout<< "First Distance :";
        D1.displayvalues();
        return 0;
}
```

C++ program for assignment operator

```
#include<iostream>
using namespace std;
class num
{
	private:
		int x;
	public:
		num(int a);
		void operator = (num b);
		void show();
};
num::num(int a)
{
	x=a;
}
Void num::show()
{
	cout<<x<< " ";
}
Void num::operator = (num a2)
{
	x=a2.x;
}
int main()
{
	num a1(100),a2(200);
	cout<<"\n Before overloading assignment operator:";
	cout<<"\n A=";
	a1.show();
	cout<<"\n B=";
	a2.show();
```

```
        cout<<"\n After overloading assignment operator Explicitly:";
        a2.operator=(a1);
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        return 0;
}
```

Inheritance

- One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
- Inheritance is the process by which new classes called derived classes are created from existing classes called base classes.
- The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

Features or Advantages of Inheritance:

Reusability:

- Inheritance helps the code to be reused in many situations.
- The base class is defined and once it is compiled, it need not be reworked.
- Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort:

- The above concept of reusability achieved by inheritance saves the programmer time and effort. The main code written can be reused in various situations as needed.
- Increases Program Structure which results in greater reliability.
- It provides extensibility i.e., we add new features to the derived class.

General Format for implementing the concept of Inheritance:

```
        Class Derived_classname: access_specifier baseclassname
```

For example, if the base class is MyClass and the derived class is sample it is specified as:

class sample: public MyClass

The above makes sample have access to both public and protected variables of base class MyClass

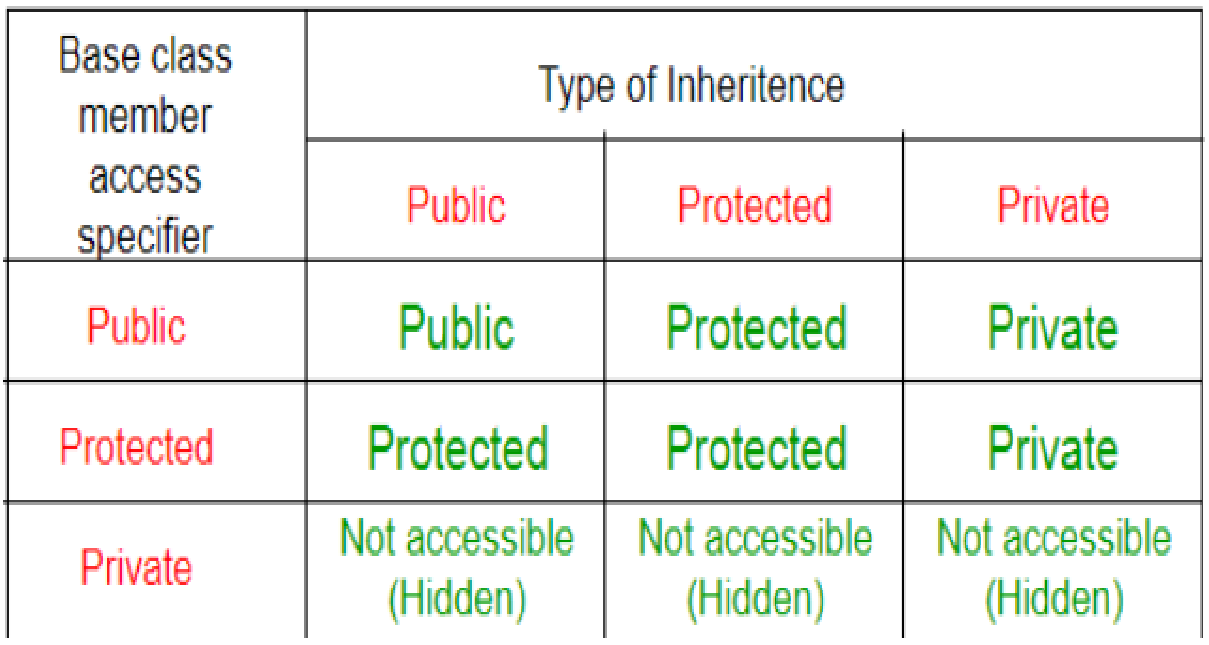
Access specifier's:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
- Public members and variables are accessible from outside the class.
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

1. Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

2. Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

3.Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class



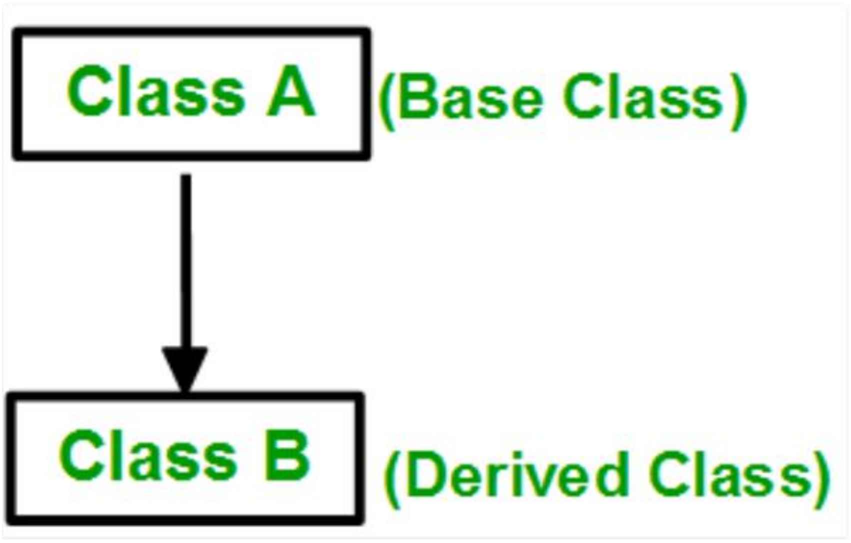
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of inheritance

1. Single level inheritance
2. Multilevel inheritance
3. Multiple inheritance

4. Hybrid inheritance
5. Hierarchical inheritance

1. **Single class Inheritance:**
   Single inheritance is the one where you have a single base class and a single derived class.



C++ program for single level inheritance using public access specifiers

```
#include<iostream>

using namespace std;

class studetails

{

        private:

                int rollnumber;

                string name;

                string branch;

                char section;

        public:

                void getdetails()

                {

                        cout<<"enter name and rollnumber and branch and section";

                        cin>>rollnumber>>name>>branch>>section;

                }

                Void displaydetails()
                {
                        cout<<"rollnumber:"<<rollnumber<<endl;
                        cout<<"name:"<<name<<endl;
                        cout<<"branch:"<<branch<<endl;
                        cout<<"section"<<section<<endl;
                }
};
Class acadetails:private studetails
{
        private:
                int sgpa;
                int cgpa;
        public:
                void getdetails1()
                {
                        getdetails();
                        cout<<"enter sgpa and cgpa";
                        cin>>sgpa>>cgpa;
                }
                void displaydetails1()
                {
                        displaydetails();
                        cout<<"sgpa:"<<sgpa<<endl;
                        cout<<"cgpa"<<cgpa<<endl;
                }
};
int main()
```

```
{
        acadetails o1;
        o1.getdetails();
        o1.getdetails1();
        o1.displaydetails();
        o1.displaydetails1();
}
```

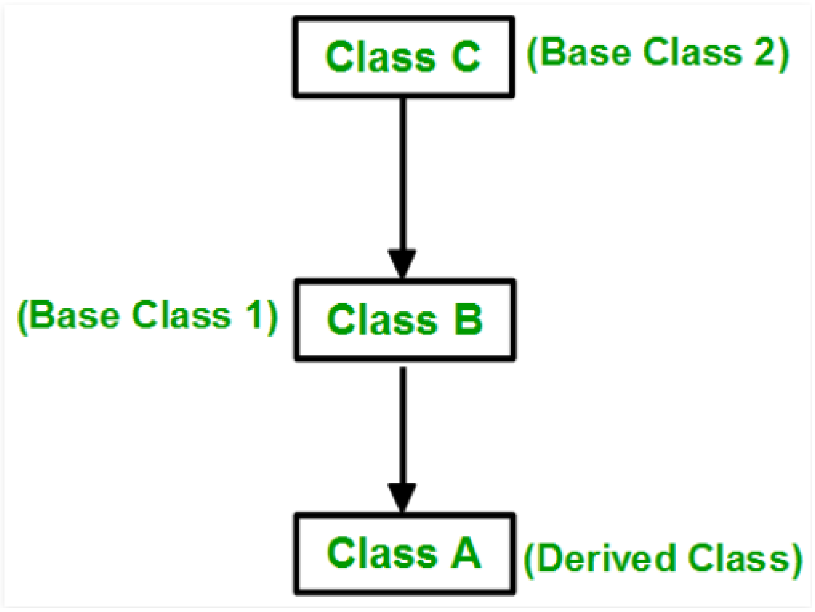
C++ program for single level inheritance using private access specifiers.

```
#include<iostream>
using namespace std;
class studetails
{
        private:
                int rollnumber;
                string name;
                string branch;
                char section;
        public:
                void getdetails()
                {
                        cout<<"enter name and rollnumber and branch and section";
                        cin>>rollnumber>>name>>branch>>section;
                }
                Void displaydetails()
                {
                        cout<<"rollnumber:"<<rollnumber<<endl;
                        cout<<"name:"<<name<<endl;
```

```
                    cout<<"branch:"<<branch<<endl;
                    cout<<"section"<<section<<endl;
                }
};
Class acadetails:private studetails
{
        private:
                int sgpa;
                int cgpa;
        public:
                void getdetails1()
                {
                        getdetails();
                        cout<<"enter sgpa and cgpa";
                        cin>>sgpa>>cgpa;
                }
                void displaydetails1()
                {
                        displaydetails();
                        cout<<"sgpa:"<<sgpa<<endl;
                        cout<<"cgpa"<<cgpa<<endl;
                }
};
int main()
{
        acadetails o1;
        o1.getdetails1();
        o1.displaydetails1();
}
```

2.Multilevel Inheritance:

In Multi level inheritance, a class inherits its properties from another derived class.



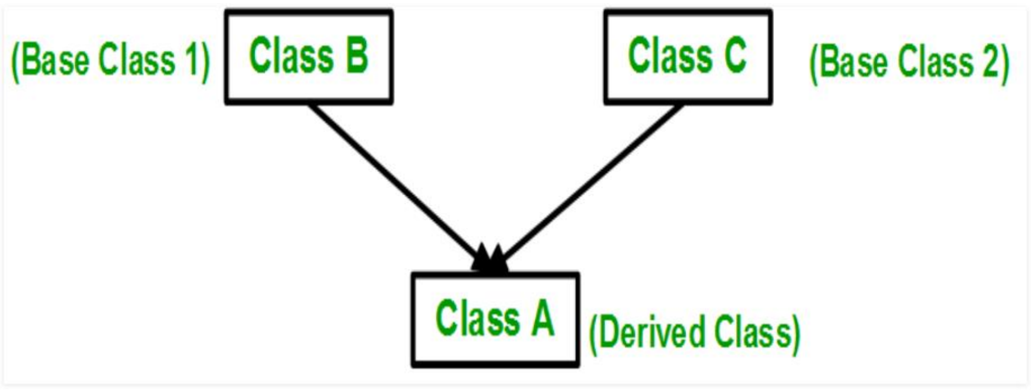
C++ program for printing total and avg of marks using multi level inheritance

```
#include<iostream>
using namespace std;
class student
{
        private:
                int id;
                string name;
        public:
        void getdetails()
        {
                cout<<"enter name and id";
                cin>>name>>id;
        }
        Void displaydetails()
        {
                cout<<name<<endl<<id<<endl;
        }
};
Class marks:public student
{
        protected:
                int m1,m2,m3,m4;
        public:
                void getmarks()
                {
                        cout<<"enter marks for m1,m2,m3,m4";
                        cin>>m1>>m2>>m3>>m4;
                }
                Void displaymarks()
                {
                        cout<<m1<<endl<<m2<<endl<<m3<<endl<<m4<<endl;
                }
};
Class totavg:public marks
{
        private:
                int total;
                float average;
        public:
```

```
        void show()
        {
                total=m1+m2+m3+m4;
                average=total/4;
                cout<<"the total of m1,m2,m3,m4 is:"<<total<<endl;
                cout<<"the average of m1,m2,m3,m4 is:"<<average<<endl;
        }

};
int main()
{
        totavg o1;
        o1.getdetails();
        o1.getmarks();
        o1.displaydetails();
        o1.displaymarks();
        o1.show();
}
```

3.Multiple Inheritances:

In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



C++ program for calculating total and average using multiple inheritance

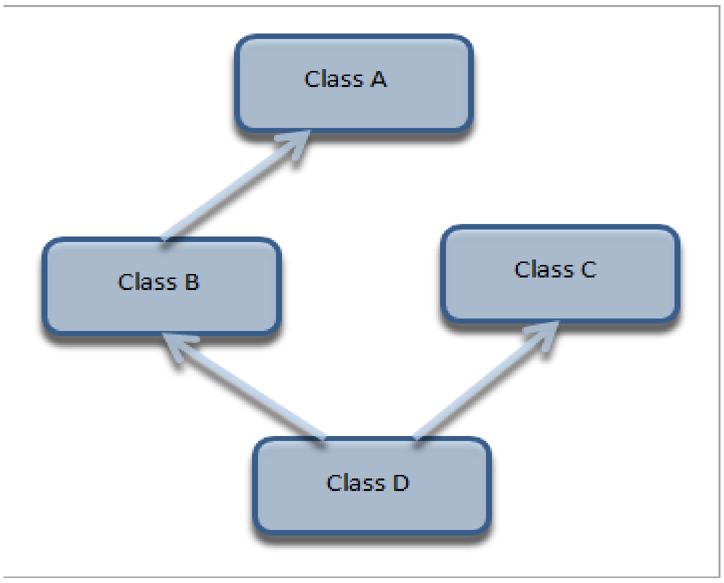
```
#include<iostream>

using namespace std;

class student//base class

{

	protected:

		int rollnumber;

		string name;

	public:

		void getdata()

		{

			cout<<"enter name:"<<name<<endl;

			cin>>name;

			cout<<"enter rollnumber:"<<rollnumber<<endl;

			cin>>rollnumber;

		}

		Void displaydata()

		{

			cout<<"name:"<<name<<endl;

			cout<<"rollnumber"<<rollnumber<<endl;

		}

};

class marks//base class

{

	protected:

		int co,python,ds,se;

	public:
```

```
                void getmarks()

                {

                        cout<<"enter marks for co,python,ds,se";

                        cin>>co>>python>>ds>>se;

                }

                Void displaymarks()

                {

                        cout<<"co marks:"<<co<<endl;

                        cout<<"python marks:"<<python<<endl;

                        cout<<"ds marks:"<<ds<<endl;

                        cout<<"se marks:"<<se<<endl;

                }

};

Class totavg:public student,public marks

{

        protected:

                int total;

                floatavg;

        public:

                void show()

                {

                        total=co+python+ds+se;

                        avg=total/4;

                        cout<<"total:"<<total<<endl;

                        cout<<"avg:"<<avg<<endl;

                }

};
```

```
int main()

{

        totavg obj1;

        obj1.getdata();

        obj1.getmarks();

        obj1.displaydata();

        obj1.displaymarks();

        obj1.show();

}
```

4. Hybrid inheritance

- Hybrid inheritance is usually a combination of more than one type of inheritance. In the above representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B and D) to get a hybrid inheritance.



C++ program for calculating eamcet marks using hybrid inheritance

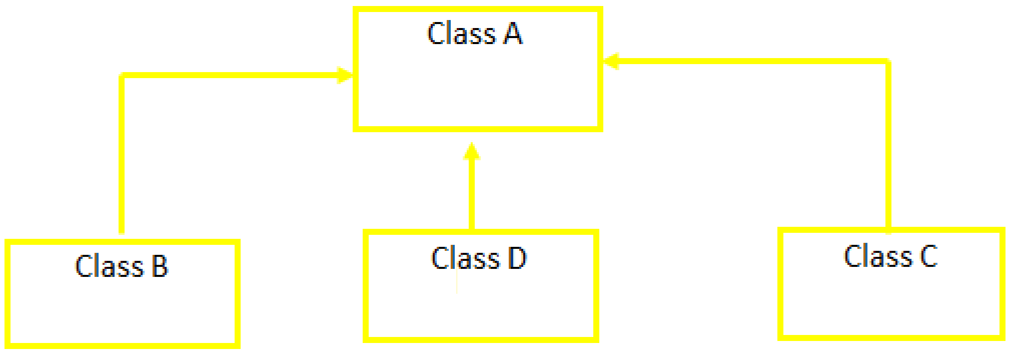
```
#include<iostream>

using namespace std;

class student//base class

{

        protected:
```

```
		string name;
		int number;
	public:
		void getdata()
		{
			cout<<"enter name";
			cin>>name;
			cout<<"enter number";
			cin>>number;
		}
		Void displaydata()
		{
			cout<<"name:"<<name<<endl;
			cout<<"number:"<<number<<endl;
		}
};
Class intermarks:public student//derived class
{
	protected:
		int maths,physics,chemistry;
	public:
		void getmarks()
		{
			cout<<"enter marks for maths,physics,chemistry";
			cin>>maths>>physics>>chemistry;
		}
		Void displaymarks()
```

```
                {
                        cout<<"maths:"<<maths<<endl;
                        cout<<"physics:"<<physics<<endl;
                        cout<<"chemistry:"<<chemistry<<endl;
                }
};
Class eamcetmarks//base class
{
        protected:
                int eammarks;
        public:
                void geteammarks()
                {
                        cout<<"enter eamcet marks";
                        cin>>eammarks;
                }
                Void displayeammarks()
                {
                        cout<<"eamcet marks:"<<eammarks<<endl;
                }
};
Class finalmarks:public intermarks,public eamcetmarks
{
        protected:
                int  finalmark;
        public:
                void displayfinalmark()
                {
                        int total=maths+physics+chemistry;
                        float avg=total/3;
                        float percent1=avg*0.25;
                        float percent2=eammarks*0.75;
                        int finalmark=percent1+percent2;
                        cout<<"final eamcet marks:"<<finalmark<<endl;
                }
};
int main()
{
        finalmarks o1;
        o1.getdata();
        o1.getmarks();
        o1.geteammarks();
        o1.displayeammarks();
        o1.displaydata();
        o1.displaymarks();
        o1.displayeammarks();
        o1.displayfinalmark();
}
```

5.Hierarchical Inheritance:

In hierarchical Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class. It's quite analogous to the File system in a unix based system



Example Program ForHierarchial Inheritance.

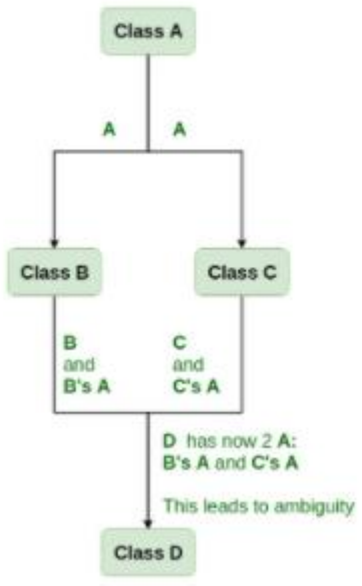
```
#include <iostream>

using namespace std;

class A //single base class

{

public:

        int x, y;

        void getdata()

        {

        cout<< "Enter value of x and y:\n";

        cin>> x >> y;

        }

};

class B : public A //B is derived from class base

{

public:

        void product()

        {

        cout<< "\nProduct= " << x * y;

        }

};
```

```
class C : public A //C is also derived from class base
{
        public:
                void sum()
                {
                        cout<< "\nSum= " << 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;
}
```

Virtual base class in C++

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.
- **Need for Virtual Base Classes:**
  Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.

};



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

```
#include <iostream>

using namespace std;

class A

{

public:

        void show()

        {

                cout<< "Hello form A \n";

        }

};

class B : public A

{

};

 class C : public A
```

```
{

};

class D : public B, public C

{

};

  int main()

{

    D object;

    object.show();

}
```

How to resolve this issue?
To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A

{

};
```

Syntax 2:

```
class C : public virtual A

{

 };
```

Note: virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.
Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

C++ program for virtual base class

```
#include<iostream>
```

```
using namespace std;

class a//base class

{

        public:

                void show()

                {

                        cout<<"this is an example program for virtual base class";

                }

};

class b:virtual public a

{

        public:

                void display()

                {

                        cout<<"this is a derived class for base class-a";

                }

};

class c:virtual public a

{

        public:

                void print()

                {

                        cout<<"this is derived class for base class-a";

                }

};

class d:public b,public c

{
        public:
        void output()
        {
                cout<<"this is derived from a and c classes";
        }
};
int main()
{
        d o1;
        o1.show();
}
```

Advantages of inheritance

- **Reusability:** Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes form the base class as needed while adding specific features to each derived class as needed.
- **Save time and effort:** the above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situation as needed.
- **Data hiding:** the base class can decide to keep some data private so that it cannot be altered by the derived class.
- **Extensibility:** it extensibility the base class logic as per business logic of the derived class.
- **Easy to understand:** It's easy to understand large program with use of inheritance.
- **Reliability:** increases program structure which result in greater reliability.
- **Maintainability:** It is easy to debug a program when divided in parts. Inheritance provides an opportunity to capture the program.

Disadvantages:-

- Inherited functions work slower than normal function as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.

- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.