INHERITANCE

Unit Structure

9.1 INTRODUCTION :

C++ supports the concept of reusability once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called ***inheritance***.

The old class is referred to as base class or super class. And the new one is called as derived class or subclass.

Inheritance:

The Inheritance is the mechanism of deriving one class from the other class. Let us assume that the class def is derived from the existing base class abc. In that case abc is called as base class and def is called as derived class. The feature that is facility of the base class will be inherited to the derived class. The derived class will have its own features. The features of the derived class will never be inherited to the base class.

Need of Inheritance:

In object oriented programming there are applications for which the classes are develop, every classes will have its own features .Consider a situation where we require all the features of a particular existing class and we also need some additional features. The above requirement can be satisfied with three different methods.

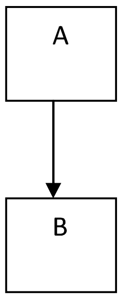
1) Modify the original existing class by acting new features into it. The disadvantage of this method is that the users of the original class will get some extra facility and they can misuse them

2) Rewrite the new class with all the required features i.e. the features of the existing class plus the new features. The disadvantage of this method is that it will be consuming more time and space.

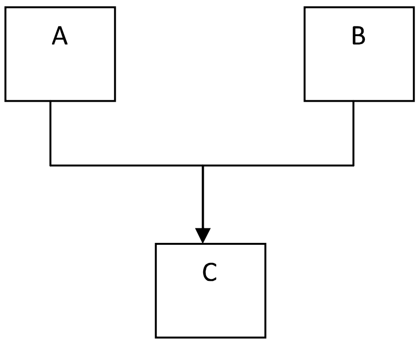
3) Derive the new class from the existing class and only write the new features in the derived class. The features of the original class will be inherited to the new class. This method consumes less space and time. Moreover the original class is not modified. Hence user of original class will not get extra facilities.

9.2 TYPES OF INHERITANCE: -

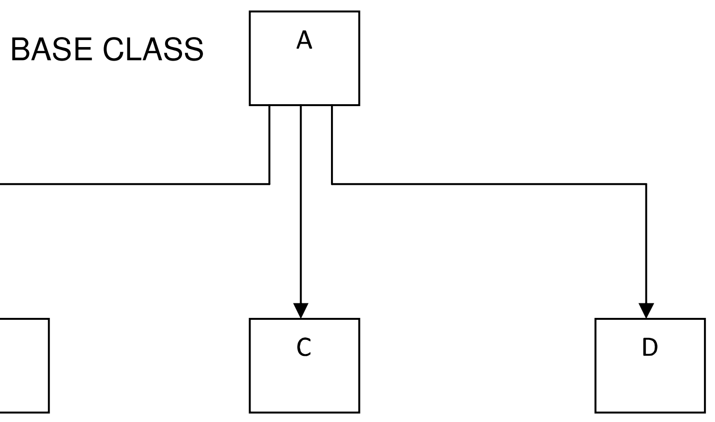
1. **Single Inheritance** : A derived class with only one base class, is called Single Inheritance.



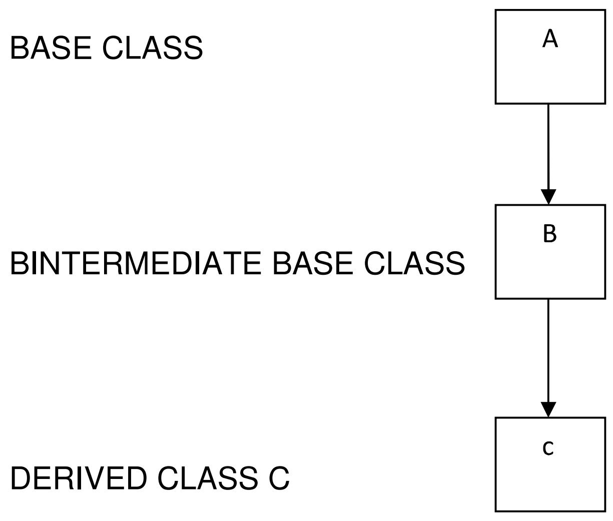
2. **Multiple Inheritance** : A derived class with several base classes, is called multiple Inheritance.



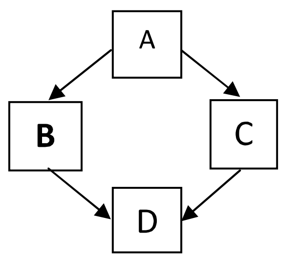
3. **Hierarchical Inheritance** : The properties of one class may be inherited by more than one class is called hierarchical Inheritance.



4. **Multilevel Inheritance** : The mechanism of deriving a class from another derived class is known as multilevel Inheritance.



5. **Hybrid Inheritance :** The hybrid Inheritance is a combination of all types of
Iheritance.



9.3 DERIVED AND BASE CLASS

1. **Base class: -**

A base class can be defined as a normal C++ class, which may consist of some data and functions.

Ex .

```
class Base
{
        : : : : :
}
```

2. Derived Class: -

i). A class which acquires properties from base class is called derived class.
ii). A derived class can be defined by class in addition to its own detail.

The general form is .

```
        class derived-class-name: visibility-mode
base-class-name
        {
                : : : : :
        }
```

The colon indicates that the derived class name is derived from the base-classname.

The visibility-mode is optional and if present it is either private or public.

9.4 PUBLIC INHERITANCE

i) When the visibility-mode is public the base class is publicly inherited.
ii) In public inheritance, the public members of the base class become public members of the derived class and therefore they are accessible to the objects of the derived class.
iii) Syntax is .

```
class ABC : public PQR
{
        members of ABC
};
```

Where PQR is a base class illustrate public inheritance.

iv) Let us consider a simple example to illustrate public inheritance.

```
# include<iostream.h>
class base
{
        int no1;
        public:
        int no2;
        void getdata();
        int getno1();
        void showno1();
};
class derived: public Base // public derivation.
{
        int no3;
        public:
        void add();
        void display();
};
void Base :: getdata()
{
        no1 = 10;
        no2 = 20;
}
int Base :: getno1()
{
        return no1;
}
void Base :: showno1()
{
        cout <<.Number1 =.<<no1 <<.\n.;
}
void Derived :: add()
{
        no3 = no2 + getno1(); // no1 is private
}
void Derived :: display()
{
        cout << .Number1 = .<<getno1() <<.\n.;
        cout << .Number2 = .<<no2<<\n.;
        cout << .Sum . <<no3 << .\n.;
}
main ( )
{
        derived d;
        d.getdata ( );
        d.add ( );
        d.showno1 ( );
        d.display ( ) ;
        d.b = 100;
        d.add ( ) ;
        d.display ( ) ;
        return 0;
}
```

The output of the program is

```
Number1 = 10
Number1 = 10
Number2 = 20
Sum = 30
Number1 = 10
Number2 = 100
Sum = 110
```

v) In the above program class Derived is public derivation of the base class Base.

Thus a public member of the base class Base is also public member of the derived class Derived.

9.5 PRIVATE INHERITANCE

i) When the visibility mode is private, the base class is privately inherited.

ii) When a base class is privately inherited by a derived class, public members of the base class becomes private members of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

iii) Public member of a class can be accessed by its own objects using the dot operator. So in private Inheritance, no member of the base class is accessible to the objects of the derived class.

iv) Syntax is .

```
class ABC: private PQR
{
member of ABC
}
```

v) Let us consider a simple example to illustrate private inheritance.

```
#include <iostream.h>
class base
{
        int x;
        public :
        int y;
        void getxy() ;
        int get_x(void) ;
        void show_x(void) ;
};
void base ::getxy(void)
{
        cout<<.Enter Values for x and y : . ;
        cin >> x >> y ;
}
int base : : get_x()
{
        return x;
}
void base :: show_x()
{
        cout <<.x = . << x << .\n.;
}
void Derived : : mul( )
{
        getxy();
        z = y * get_x ( ) ;
}
void Derived ::display()
```

```
{
        show_x ( ) ;
        cout <<.y = . <<y<<.\n.;
        cout<<.z = . <<z<<.\n.;
}
main ( )
{
        Derived d;
        d. mul();
        d.display() ;
        //d.y = 4a; won.t work y has become
        private.
        d.mul();
        d.display();
}
```

The output is Output: -

```
Enter values for x and y: 510
X = 5
Y = 10
Z = 50

Enter values for x and y:1220
X = 12
Y = 20
Z = 240
```

Derived Class Constructor

When a base class has a constructor, the derived class must explicitly call it to initialize the base class portion of the object. A derived class can call a constructor defined by its base class by using an expanded form of the derived class' constructor declaration.

The general form of this expanded declaration is shown here:

```
derived-constructor(arg-list) : base-cons(arg-list);
{
        body of derived constructor
}
```

Here, base-cons is the name of the base class inherited by the derived class. Notice that a colon separates the constructor declaration of the derived class from the base class constructor. (If a class inherits more than one base class, then the base class constructors are separated from each other by commas.)

The following program shows how to pass arguments to a base class constructor. It defines a constructor for TwoDShape that initializes the width and height properties.

```
// Add a constructor to TwoDshape.

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

// A class for two-dimensional objects.
class TwoDshape
{
        //These are private
        double width;
        double height;
public:
        //Constructor for TwoDshape.
        TwoDshape(double w, double h)
        {
                width=w;
                height=h;
        }
        void showdim()
        {
                cout<<"Width and height are"<< width<<
"and" <<height<<"\n";
        }

        //accessor functions
        double getWidth() {return width;}
        double getHeight(){return height;}
        void setWidth(double w)P{width=w;}
        void setHeight(double h){height=h;}
};
```

```
//Triangle is derived from TwoDshape.
class Triangle : public TwoDshape
{
        char style[20]; // now private
        public:
        // Constructor for triagle.
        Triagle(char    *str,    double    w,    double    h)    :
TwoDshape(w, h)
        {
                strcpy(style, str);
        }

        double area()
        {
                return getWidth() *getHeight()/2;
        }

        void showStyle()
        {
                cout<<”Triangle is”<<style<<”\n”;
        }
};

int main()
{
        Triangle t1(“isosceles”,4.0,4.0);
        Triangle t2(“right”,8.0, 12.0);
        cout<<”Info for t1:\n”;
        t1.showStyle();
        t1.showDim();
        cout<<”Area is” << t1.area()<<”\n”;

        cout<<”\n”;
        cout<<”Info for t2:\n”;
        t2.shwoStyle();
        t2.showDim();
        cout<<”Area is”<<t2.area()<<”\n”;

        return0;
}
```

Here, Triangle( ) calls TwoDShape with the parameters w and h, which initializes width and height using these values. Triangle no longer initializes these values itself. It need only initialize the value unique to it: style. This leaves TwoDShape free to construct its subobject in any manner that it so chooses. Furthermore, TwoDShape can add functionality about which existing derived classes have no knowledge, thus preventing existing code from breaking.

Any form of constructor defined by the base class can be called by the derived class' constructor. The constructor executed will be the one that matches the arguments

Summary of access privileges

1. If the designer of the base class wants no one, not even a derived class to access a member, then that member should be made private.
2. If the designer wants any derived class function to have access to it, then that member must be protected.
3. if the designer wants to let everyone, including the instances, have access to that member , then that member should be made public.

Summary of derivations

1. Regardless of the type of derivation, private members are inherited by the derived class, but cannot be accessed by the new member function of the derived class , and certainly not by the instances of the derived class .

2. In a private derivation, the derived class inherits public and protected members as private. a new members function can access these members, but instances of the derived class may not. Also any members of subsequently derived classes may not gain access to these members because of the first rule.

3. In public derivation, the derived class inherits public members as public , and protected as protected . a new member function of the derived class may access the public and protected members of the base class ,but instances of the derived class may access only the public members.

4. In a protected derivation, the derived class inherits public and protected members as protected .a new members function of the derived class may access the public and protected members of the base class, both instances of the derived class may access only the public members .

Table of Derivation and access specifiers

Derivation Type	Base Class Member	Access in Derived Class
Private	Private	(inaccessible)
	Public	Private
	Protected	Private
Public	Private	(inaccessible)
	Public	Public
	Protected	Protected
Protected	Private	(inaccessible)
	Public	Protected
	Protected	Protected

We can summarize the different access types according to whom can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
Not-members	yes	no	no

where "not-members" represent any reference from outside the class, such as from main(), from another class or from any function, either global or local.