INHERITANCE

The general syntax of derivation of property from base class to derived class is

```
Class derived _class _name : visibility mode base _class _name
{
        ……..//
        ……..//    members of derived class
        …… //
};
```

visibility mode can be private, public or protected.

Visibility mode of derivation specifies what will be the visibility of the base class member in the derived class after derivation.

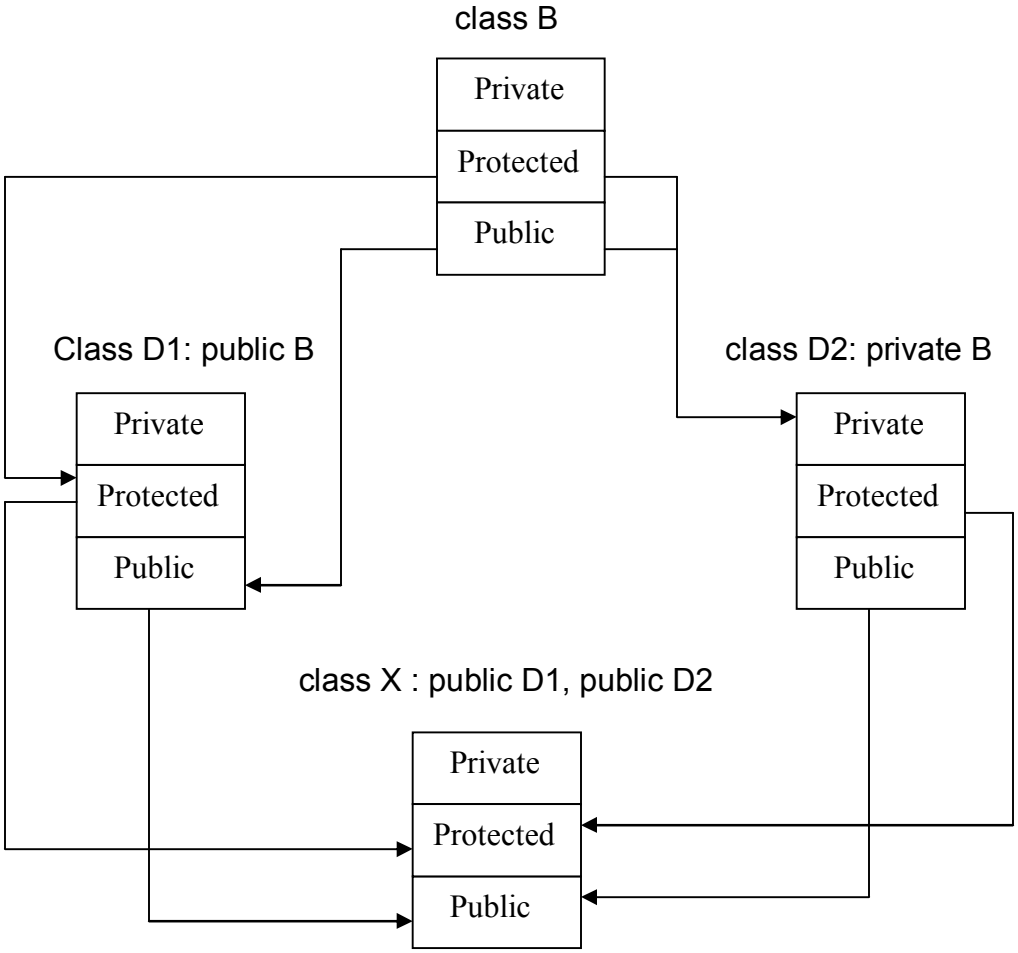
Any member of the base class can be derived privately, publicly or even with a protected way only to ensure further stage derivation with information hiding. However, private members of the base class can't be inherited.

If we consider the memory allocation for any object of derived class, we will found the derived allocation contains all the base class members irrespective of their visibility label.

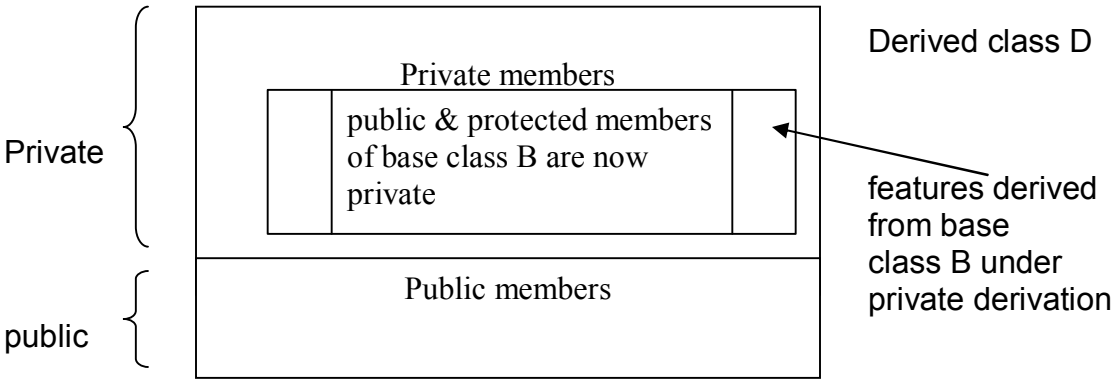
Visibility of the inherited members:

Base class Visibility	Private Derivation	Public Derivation	Protected Derivation
Private	not inherited	not inherited	not inherited
Public	private	public	protected
Protected	private	protected	protected

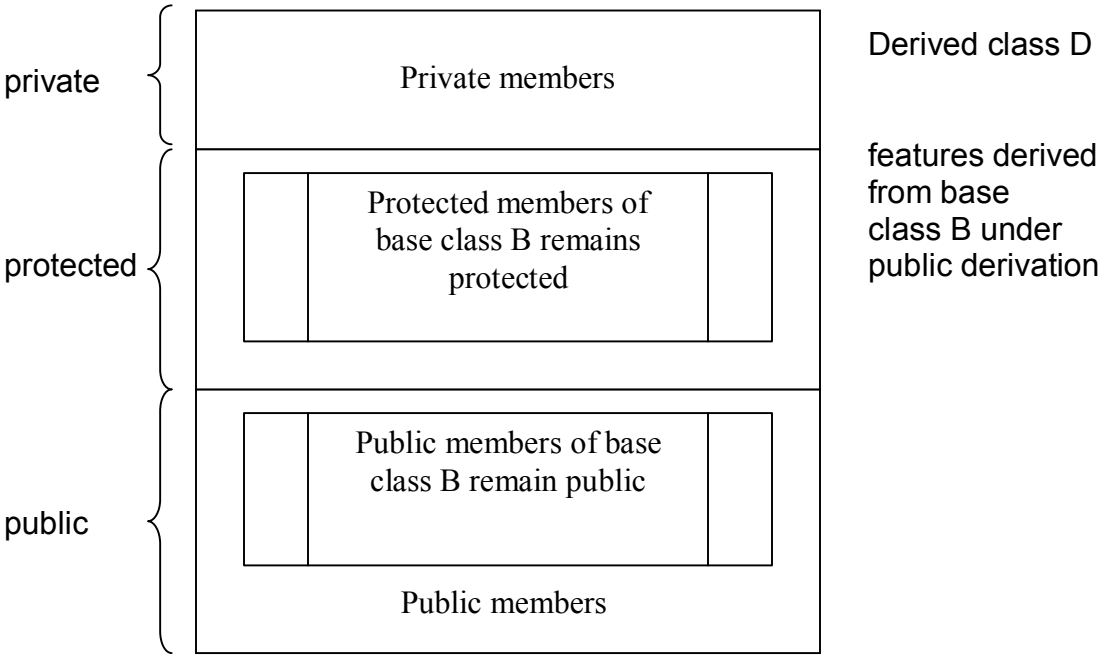
Following is the effect of inheritance on the visibility of the members-



When a base class is privately inherited by a derived class, *public members* of the base class become *private* in the derived class and therefore public members of the base class can then only be accessed by the member functions of the derived class. They are otherwise inaccessible i.e. no direct accessibility of them is allowed using derived class object



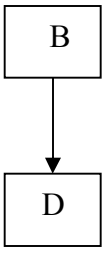
private members of the Base class are strictly non-inheritable under every derivation.
When the base class is publicly inherited , *public members* of the base class become '*public members'* of the derived class and therefore they are accessible to the objects of the derived class. In both the cases the private members are not inherited and the private members of a base class will never become the members of its derived class.



SINGLE INHERITANCE

When the single derived class inherits the feature of only one base class.

Derivation Diagram :



Derivation Syntax :

class D : derivation mode B

This derivation mode can be either private, public or protected.

Sample program :

When single derived class inherits the property of a single base class. E.g.

```
#include< iostream.h>

class B
{
                int     a;              // not inheritable
        public: int     b;              // inheritable
                void get _ab(void);
                int  get _a(void);
                void show _a(void);
};


class D: public B
{
                int     c;
        public: void mul(void);
                void display(void);
};

void B :: get _ab(void)
{
        a=10;
        b=20;
}

int B :: get _a(void)
{
        return(a);
}

void B ::show _a(void)
{
        cout<<"a="<<a;
}

void D:: mul(void)
{
        c = b * get _a( );
}

void D:: display(void)
{
        cout<<"a="<<get _a( )<<endl;
        cout<<"b="<<b<<endl;
        cout<<"c="<<c<<endl;
}


int main(void)
{
        D       d;
        d. get _ab( );
        d. show _a( );
        d. mul( );
        d. display( );
        d. b=30;
        d. mul( );
        d. display( );
        return ;
}
```

Overriding Base class member functions:

When the base class and a derived class have member functions with the same name, same number of arguments and same type of argument list , then when called by the object of the derived class, the function in the derived class overrides the base class function that is derived.

Consider the following example :

```
class base
{
        public: void  display( )
                {
                        cout<<"\n Base class function" ;
                }
};

class derived: public base
{
        public: void  display( )
                {
                        cout<<"\n Derived class function" ;
                }
};
```

Now by above the function display( ) should have two copies at derived. One is own member of derived, another is the derived from base whose visibility is also public. The question is, if we do

```
        int   main(void)
        {
                derived         dd;
                dd. display( );
        }
```

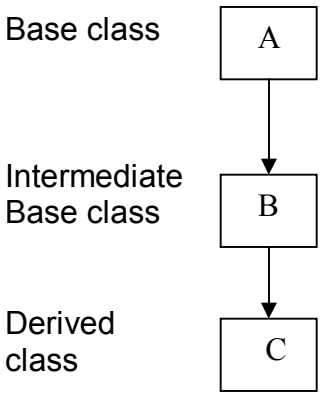
then what should be the output ?

Clearly the derived class function will override the base class function i.e. derived with public visibility as it is accessed by a derived class object. The derived class member function will have higher priority than the base class one.

So the output will be : Derived class function.

MULTILEVEL INHERITANCE

When the single derived class inherits the feature of only one base class.



A derived class with multilevel inheritance is declared as follows-

```
class A {  …….};              //  Base class
class B : public A {……};     //  B derived from A
class C : public B {……};     //  C derived from B
```

The following program will illustrate :

```
#include < iostream. h>

        class student
        {
                protected       : int     roll _no;
                public          : void get _roll( int);
                                  void show _roll( void);
        };

        void student :: get _roll ( int  a)
        {
                roll _no = a;
        }

        void student :: show _roll( void)
        {
                cout<<" \n Roll: "<<roll _no;
        }
```

```
class test : public student
{
        protected       :       float   sub1,sub2;
        public          :       void  get _marks ( float, float) ;
                                void   show _marks( void );
};

void test :: get _marks ( float  x, float  y)
{
        sub1 = x ;
        sub2 = y ;
}

void test :: show _marks ( float  x, float  y)
{
        cout<<" \n Marks in subject 1 :"<< sub1 ;
        cout<<" \n Marks in subject 2 :"<< sub2 ;
}

class result : public test
{
                        float   total;
        public :        void display ( void );
};
```

After two stage derivation of result from student through test, the second stage derived class result will have following visibility of members :

```
                    float  total;                   // own member
protected       :   float  sub1, sub2;              // both derived from test
                    int roll _no;           // derived from student through test
public          : void get _roll( int );            // derived from student
                  void show _roll( int );           // derived from student
                  void  get _marks( float , float);  //derived from test
                  void show _marks( void )          // derived from test
                  void  display ( void );               // own member
```

Thus display( ) of result may be defined as follows :

```
        void  result  :: display ( void )
        {
                total = sub1 +sub 2;
                show _roll( void );
                show _marks( void);
                cout<<" \n Total : " << total;
        }
```

Once all the classes of different level are declared the main( ) function can be declared as follows :

```
int  main( void )
{
        result  student1;
        student1.get _roll(50);
        student1.get _marks ( 75.0, 86.5);
        student1. display( );
        return 0;
}
```

OUTPUT

```
Roll : 50
Marks in Subject1: 75.0
Marks in Subject2: 86.5
Total : 161.5
```

MULTIPLE INHERITANCE

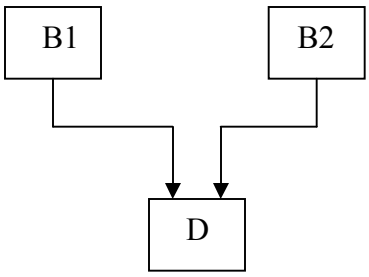
A class can inherit the attributes of two or more classes. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes.

The syntax of a derived class with multiple base classes is as follows :

```
Class D : visibility B1, visibility B2,…….
{
        …… ;
        Body of D;
        …….. ;
};
```

where visibility may be either public or private. The base classes are separated by commas.

We can represent it using following diagram-



Feature of more than one base class i.e. variety of features are incorporated in to derived class.

Following program can illustrate the concept explicitly-

```
class P : public M , public N
{
        public :        void display( void );
};
```

classes M and N have been specified as follows :

```
class M
{
        protected  :  int  m ;
        public  :   void get _m( int );
};

void M :: get _m( int  x )
{
        m = x ;
}

class N
{
        protected :    int  n ;
        public :       void get _n( int );
};

void  N :: get _n(int y)
{
        n= y;
}
```

The derived class P which is declared earlier as –

```
class P : public M , public N
{
        public :        void display( void );
};
```

will have the following visibility of its members :

```
class P
{
        protected :    int  m, n ;
        public      :  void get _m ( int );
                       void get _n( int );
                       void display ( void);
};
```

the member function display( ) can be defined as follows :

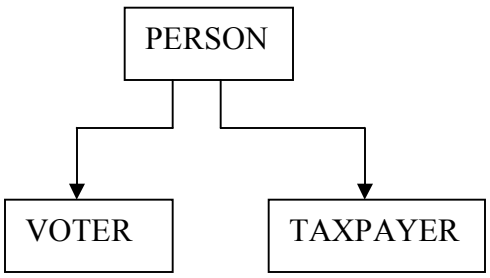
```
void P :: display (void)
{
        cout << " m= "<<m << endl ;
        cout << " n= "<<n << endl ;
        cout << " m*n= "<<m *n << endl ;
}
```

The main( ) function which provides the interface may be written as follows :

```
int  main( void)
{
        P  p;
        p. get _m( 10 );
        p. get _n( 20 );
        p. display( );
        return 0;
}
```

HIERARCHICAL INHERITANCE

When we derive more than one class from a single base class, it is known as hierarchical inheritance. This is represented by following diagram :



Derivation syntax : will be single inheritance type for each of the derived classes.

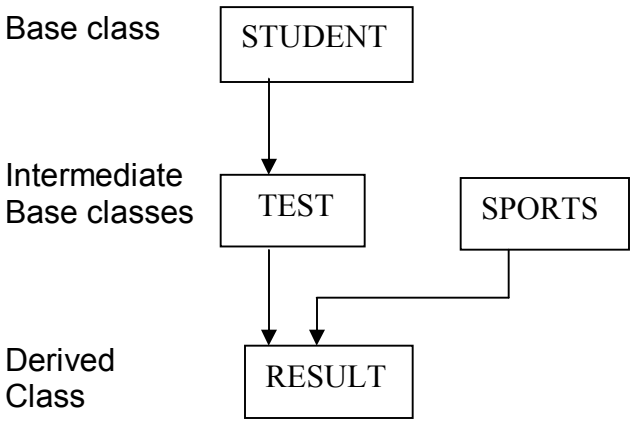
```
# include< iostream. h>
const int size=15;

class person
{
        char    name[size];
        int     age;
public: void get _data(void);
        void show(void);
};
```

```
void person : : get _data(void)
{
        cout << “\n Enter name :”;
        cin>>name;
        cout << “\n Enter age :”;
        cin>>age;
}

void person : : show(void)
{
        cout << endl<<name<<” \t ”<<age“<<” \t ”;
}

class voter  : public person
{
                        int     icard;
        public:         void    get( void)
                        {
                                get _data( );
                                cout<<”\n Enter voter id:” ;
                                cin>>icard;
                        }
                        void display( )
                        {
                                show( );
                                cout<<icard<<endl;
                        }
};

class taxpayer  : public person
{
                        int     pan;
        public:         void    get( void)
                        {
                                get _data( );
                                cout<<”\n Enter Pan number :” ;
                                cin>>pan;
                        }
                        void display( )
                        {
                                show( );
                                cout<<pan<<endl;
                        }
};
```

```
int main(void)
{
        voter           vobj;
        taxpayer        tobj;
        vobj. get( );
        tobj. get( );
        vobj. display( );
        tobj. display( );
        return 0;
}
```

HYBRID INHERITANCE

When the single derived class inherits the feature of only one base class.




```
#include< iostream. h>

class student
{
        protected :     int roll _no;
        public:         void get _no(int a)
                        {
                                roll _no = a;
                        }
                        void put _no(void)
                        {
                                cout<< " Roll No:"<<roll _no<<endl;
                        }
};

class test : public student
{
        protected :     float part1, part2;
```

```
        public:         void get _marks(float x, float y)
                        {
                                part1 = x ;
                                part2 = y ;
                        }
                        void put _marks(void)
                        {
                                cout<<" marks obtained :"<<endl;
                                cout<<"\n Part1 :"<<part1;
                                cout<<"\n Part2 :"<<part2;
                        }
};

class sports
{
        protected :     float    score;
        public :        void get _score(float s)
                        {
                                score = s;
                        }
                        void put _score(void)
                        {
                                cout<<" Sports wt: "<< score<<"\n\n";
                        }
};

class result : public test, public sports
{
                  float  total;
        public : void display( void);
};

void result :: display(void)
{
                total =part1 + part2 + score;
                put _no( );
                put _marks( );
                put _score( );
                cout<< "\n Total Score :"<<total;
}

int main(void)
{
        result stud;
        stud. get _no (1234);
        stud. get _marks ( 86.6, 78.8);
```

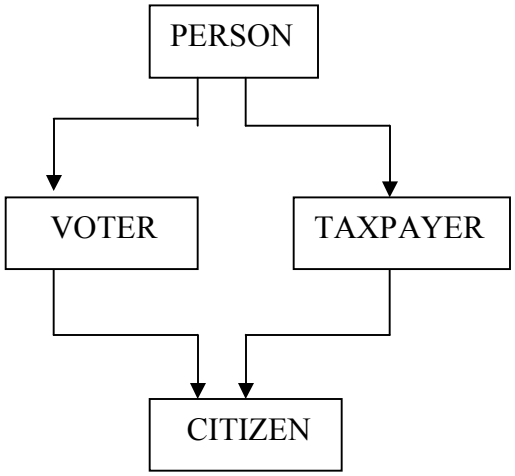
```
        stud. get _score( 70.0);
        stud. display( );
        return 0;
}
```

ABSTRACT CLASS

The class corresponding to which no object is created is referred to as Abstract Base class. It is designed only to act as a base class the property of which is derived to other classes. It is more a design concept based upon which other classes may be built.

VIRTUAL BASE CLASS

This is a method that is applied in a typical situation involving multilevel, multiple and hierarchical inheritance. Consider the following construction-



The class CITIZEN has two direct base classes VOTER and TAXPAYER which themselves have a common base class PERSON. The class CITIZEN inherits the traits of class PERSON via two separate paths. The problem here is that all the features of the base class inherited twice to the derived class via separate paths comprising two intermediate base classes as all the public and protected members are derivable with relevant derivability mode. Consequently the derived class will have duplicate sets of base class members in such case of ***Multi-path Inheritance.*** This introduces ambiguity.

Following is the mechanism to get rid of that duplication. The duplication of inherited members due to this multiple paths can be avoided by making the common base class (ancestor class) as *virtual base class* while declaring the direct or intermediate base class as follows-

```
class PERSON                                    // base class
{
        . . . . . ;
        . . . . . ;
};
class VOTER : virtual public PERSON             // intermediate base
{
        . . . . . ;
        . . . . . ;
};
class TAXPAYER : public virtual PERSON          // intermediate base
{
        . . . . . ;
        . . . . . ;
};
class CITIZEN : public VOTER, public TAXPAYER   // derived class
{
        . . . . . ;
        . . . . . ;
};
```

By virtual derivation of actual base class into the intermediate base classes will prevent the duplication of actual base class members into resultant derived class.

CONSTRUCTORS IN DERIVED CLASSES

The following points should be observed at the time of creation as derived class constructor-

- If base class constructors don't have arguments then derived class need not have a constructor function.
- If base class contains parameterized constructor then it is mandatory to have derived class constructor and C++ provides a special syntactical mechanism to pass the arguments to base class constructor through it.
- It is to be remembered that we normally create and use derived class objects in inheritance. So there is need to have derived class constructor that passes arguments to base class constructor.
- In multiple inheritance, the base classes are constructed in the *order they appear in the declaration of the derived class*. In multilevel inheritance, the constructor will be executed in the *order of inheritance*.

Suppose the classes A and B have two integer members each. Moreover assume that both of these classes be derived to the class D having one integer member of its own, in multiple inheritance.
Following is the syntax of the derived class constructor that passes arguments to the base class constructor

```
D (int a1, int a2, int b1, int b2, int  d1) : A(a1, a2) , B(b1, b2)
{                                                    // call to base constructor
        d =d1;                                       // executes its own body
}

                                /* Example program */
#include < iostream. h>
class alpha
{
                int  x;
        public:  alpha ( int  i )
                {
                        x =i ;
                        cout << "  alpha initialized "<< endl;
                }
                void show _x(void)
                {
                        cout<< " x = " << x<<endl ;
                }
};
class beta
{
                float  y ;
        public:  beta ( float j )
                {
                        y =j;
                        cout << "  beta initialized "<< endl;
                }
                void show _y(void)
                {
                        cout<< " y = " << y<< endl ;
                }
};
class gamma : public beta , public alpha
{
                 int  m ;
        public:  gamma ( int  a, float b, int  c ) : alpha(a), beta( b)
                {
                        m = c;
                        cout << "  gamma initialized "<< endl;
                }
                void show _m(void)
                {
                        cout<< " m = " << m<< endl ;
                }
};
```

```
int  main(void)
{
        gamma g( 10, 34.9, 100);
        cout<<endl;
        g. show _x( );
        g. show _y( );
        g. show _m( );
        getch( );
        return 0;
}
```

The output of the program -

```
beta initialized
alpha initialized
gamma initialized
 x = 10
 y = 34.9
m = 100
```

EXECUTION OF BASE CLASS CONSTRUCTORS

Method of Inheritance	Order of execution

class B : public A
{
};
 	
class A :public B, public C
{
};
 	
Class A :public B, virtual public C
{
};
	A( ); base constructor
	B( ); derived constructor
	
	B( ); base constructor (first)
	C( ); base constructor (second)
	A( ); derived constructor
	
	C( ); virtual base constructor
	B( ) ; base constructor
	A( ); derived constructor

C++ supports another method of initialization of class objects. This method is known as initialization list in the constructor function. The general form of which is :

```
Constructor(argument list) : initialization section
{
        assignment section
}
```

e.g. consider

```
class XYZ
{
        int  a, b;
public :  XYZ ( int  i, int  j) : a( i), b(2*j)
        { }
};
int main(void)
{
        XYZ  x(2, 3);
        return 0;
}
```

This program will initialize a to 2 and b to 6.