```
  emp . show ( ) ;
}
```

Inheritance

It is an important aspect in OOP. One of the major advantages is the reusability of code. The reuse of a class that has already tested, debugged and used many times can save us the effort of developing and testing the same again. Shared properties are defined only once, and reused as often as desired. The mechanism of deriving a new class from an old is called *inheritance*.

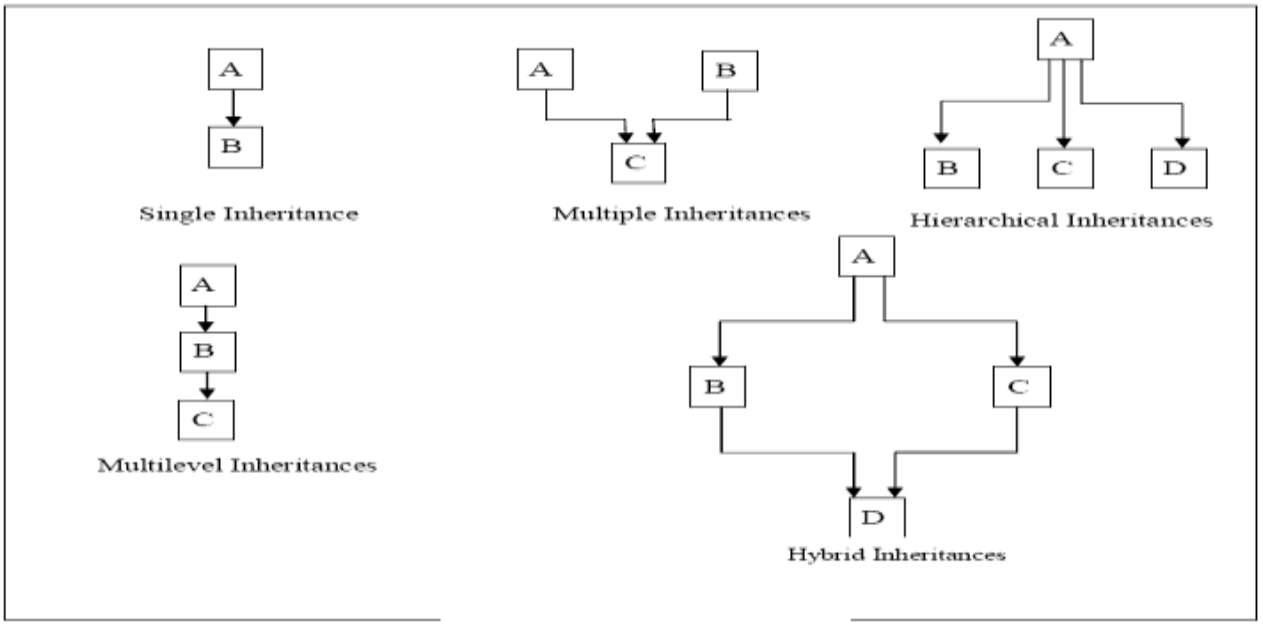
The old class is referred as the *base class* (*super class*) and the new one that inherits the properties of the base class is called the *derived class* (*subclass*).

The objects of a derived class contains all the members of the base except the private data and functions of the base class.

A derived class can share selected properties (functions as well as data members) of its base classes, but makes no changes to the definition of any of its base classes.

Types of Inheritance:

1- Single Inheritance.
2- Multiple Inheritance.
3- Multilevel Inheritance.
4- Hybrid Inheritance.



Single Inheritance

In this type, there is only one derived class and only one base class.

A derived class has direct access to both its own members and the public members of the base class.

The general form of defining a single inheritance is:

```
class derived class-name : access-specifier  base-class-name
{
    Member Data  ;
    Member Functions  ;
}
```

Where :

access-specifier: is either (*private* , *public* , or *protected*). The default is *private*, that is if no access specifier is present, the access is *private*.

Public Access-Specifier:

Using public means that all of the public members of the base class will become public members of the derived class and are available to the member functions of derived just as if they had been declared inside it.

However, derived's member functions do not have access to the private elements of base. This is an important point. Even though derived class inherits base class, it has access only to the public members of base. In this way, inheritance does not circumvent (يكسر او يتحايل) the principles of encapsulation necessary to OOP.

Base Class Section	Public derivation
Private	Not inherited
Protected	Protected
Public	Public

Example 1: This example shows a single inheritance of one base class and one derived class with public access specifier.

```
# include <iostream.h>
class Base
{ int i , j ;
  public:   void set ( int x , int y)
              { i  = x  ;
                j  = y ;}
             void view ( )
             { cout << i  <<  j  ; }
};
class Derived : public Base
{ int  z  ;
  public:  Derived ( int  a )
             {  z  =  a  ; }
             void viewD ( )
             { cout << z  ;  }
} ;
main ( )
{ Derived xd (4)  ;
  xd . set ( 2 , 6 )  ;   // set is known to Derived
  xd . view ( ) ;        // view is known to Derived
  xd . viewD ( ) ;
}
```

Example 2: Write a program to use a base class "institute" with data (name and emp_no) and functions for input and output. The base class has a derived class "Department" with data (dept_name and tel_no) and functions for input and output.

```
# include <iostream.h>
class institute
{ private:  char name [30] ;
             int emp_no  ;
  public:
            void institute_input ( )
            { cin >> name  ;
              cin >> emp_no  ; }
```

```
            void  show ( )
             { cout <<  name  << "\n"  ;
               cout << emp_no  << "\n"  ;  }
};

class  Department  : public institute
{  private : char dept_name  [20] ;
             int   tel_no   ;
   public :
             void  read ( )
              {  cin >> dept_name  ;
                 cin >> tel_no  ;       }
             void depShow ( )
             {  cout  <<  dept_name  << "\n" ;
                cout  <<  tel_no  << "\n"  ;  }
};
main ( )
{ Department   D   ;
  D . institute_input ( ) ;  // call to a function in class institute.
  D .  read ( )  ;
  D . show ( ) ;            // call to a function in class institute
  D . depShow ( )  ;
}
```

Inheritance and protected Members

When a member of a class is declared as **protected**, that member is not accessible by nonmember elements of the program. Access to a protected member is the same as access to a private member—it can be accessed only by the members of its class.
If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class (Making a private Member inheritable).

Example 3:

```
#include <iostream.h>
class base
{ protected:  int i, j;   // private to base, but accessible by derived
  public:
        void set ( int a , int b ) {  i = a  ;   j = b ; }
```

```
        void show ( ) { cout << i << " " << j << "\n"; }
};
class derived : public base
{  int k;
  public:   void setk( ) { k = i * j; } // derived may access base's i and j. If i and j declared as
                                        // private in base, then derived would not have access to
                                        // them and the program would not compile.
    void  showk( )  { cout << k << "\n"; }   } ;
main( )
{  derived  ob  ;
  ob.set (2 , 3 ) ;  // OK, known to derived
  ob.show( ) ;   // OK, known to derived
  ob.setk( ) ;
  ob.showk( ) ;
}
```

Example 4: Shows public inheritance and protected members. This example also use the same functions' names in both *base* and *derived* class.

```
# include <iostream.h>
const int size = 30 ;
class employee
{ protected:  char name [size] ;
              int age ;
              char department [size] ;
             void initialize ( )
             {  cin >> name   ;
                cin >> age  ;
                cin >> department  ;
             }
             void describe ()
            { cout << name << endl ;
              cout << age << endl ;
              cout << department  ;
            }
};
class manager : public employee
{ protected : int level ;
            void initialize ( )
             {  employee :: initialize ( )  ;  // call to function initialize( ) in employee class
                  cin >> level;
              }
            void describe ( )
```

```
        {   employee :: describe ( );  // call to function describe( ) in employee class
            cout << level  ;      }
};
main ( )
{ manager aa;
  aa . initialize ( ) ;  // error, manager:: initialize ( ) is not accessible. initialize( ) is  protected
   aa . describe ( ) ; // error, manager:: describe ( ) is not accessible. describe ( ) is  protected
}
```

Note: there are two methods to avoid the above errors:

1- Write *initialize( )* and *describe( )* in public section to call them in main( ).
2- Write *initialize( )* in *protected* section, Write *describe( )* in *public* section and call the function *initialize( )* in the *describe* function.

H.W: Write the above program with modification to avoid the errors.

Note: In the above example, the scope resolution operator (::) tells the compiler that this version of *initialize( )* and *describe( )* belong to the employee class (i.e., this *initialize()* and *describe( )* are in employee's scope).
In C++, several different classes can use the same function name. The compiler knows which function belongs to which class because of the scope resolution operator.

Private Access-Specifier:

When a access specifier is *private*, all *public* and *protected* members of the base class become *private* members of the derived class.
Therefore, the public members of the base class can only be accessed by the member functions of the derived class and cannot be accessed by parts of your program that are not members of either the base or derived class.

Base Class Section	Private derivation
Private	Not inherited
Protected	Private
Public	Private

Example 1:

```
# include <iostream.h>
class human
{  public :  char name [30]  ;
             int age  ;
```

```
            void getdata ( )
             { cin >>  name  ;
               cin >> age  ; }
            void putdata ( )
             { cout  <<  name  <<  endl  ;
               cout  <<  age  <<  endl  ;   }
} ;

class  worker : private  human
{  public :  int  grade  ;
            void input ( )
             {  cin >> grade  ;   }
            void view ( )
            {  cout  << grade  ;  }
} ;
main ( )
{ worker  W ;
  W. getdata ( ) ;   // error message('human::getdata' is not accessible )
  W. input ( )  ;
  W. putdata ( )  ; // error message('human::putdata' is not accessible )
  W. view ( ) ;
}
```

Protected Access Specifier:

It is possible to inherit a base class as **protected.** When this is done, all *public and protected* members of the base class become *protected* members of the derived class.

Base Class Section	Protected derivation
Private	Not inherited
Protected	Protected
Public	Protected

Example 1:

```
#include <iostream.h>
class base
{ protected:  int i , j ;  // private to base, but accessible by derived
  public:
        void setij (int a , int b) { i = a  ;  j = b  ; }
        void showij ( ) { cout << i << "  " << j << "\n"  ;  }
 };
class derived : protected base  // Inherit base as protected.
{ int k;
```

```
  public:
      void setk( ) { setij (10, 12) ; k = i * j ; } // ok, access to setij , i ,and j
      void showall( ) { cout << k << "  "  ;   showij ( ); }  // ok
};
main( )
{ derived ob;
  ob.setij(2 , 3 ) ;  // illegal, setij() is protected member of derived
  ob.setk( ) ;     // OK, public member of derived
  ob.showall( ) ; // OK, public member of derived
  ob.showij( ) ;  // illegal, showij() is protected member of derived
}
```

As you can see by reading the comments, even though **setij()** and **showij()** are public members of **base,** they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main( )**. The following table summarizes the visibilities of members and modifications on them when they are inherited.

Base Class Section	**Derived Class Visibility (derivation)**		
	public	**private**	**protected**
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	private	Protected
Public	public	private	Protected

Note: In all cases, the private members are not inherited; therefore they never become members of the derived class.

ملخص هام:

- اذا ورث الـ class الابن أعضاء الـ class الاب كـ public فالاستدعاء **من الممكن** ان يكون داخل احدى دوال الـ class الابن **أو** داخل دالة الـ main().
- أما اذا تمت وراثة الاعضاء من ال class الاب كـ protected او private **فيجب** ان يكون الاستدعاء لتلك الاعضاء **داخل الابن فقط** لان ال main() يجب الا يوجد فيه سوى ال public .

Multiple Inheritances

In multiple inheritances, a class inherits the properties of two or more classes. They allow us to combine the features of several existing classes as a starting point for defining new classes.

The general form of a derived class with multiple base classes is as follow:

```
Class Derived-name : access-specifier  Base-class1-name ,
                     access-specifier  Base-class2-name
{
      Member Data  ;
      Member Functions ;
```

Example:

```
# include <iostream.h>
class Parent1
{ public :  int x  ;
            void display_x ( )
            { cout  <<  x  ; }
} ;
class Parent2
{  public : int y ;
           void  display_ y ( )
           {  cout  <<  y  ;  }
} ;
class  Derived : public Parent1 , public Parent2
{  public :  void fillAB (int A , int B )
              { x  =  A  ;
                y  =  B  ; }
} ;

main ( )
{   Derived  N  ;
    N . fillAB ( 3 , 10 )  ;
    N . display_x ( )  ;    // from Parent1 (Base)
    N . display_y ( )  ;    // from Parent1 (Base)
}
```

Multilevel Inheritance

The mechanism of deriving a class from another "derived class" is known as *multilevel inheritance*.

If the class ***(A)*** serves as a base class for the derived class *(B)* which in turn serves as a base class for the derived class *(C)*, the class B is known as *Intermediate Base* class since it provides a link for the derived class C.

The chain ABC is known as *inheritance path*.

The general form of multilevel inheritance is as follow:

```
class Base-class-name
{
    .......
} ;
class Derived-class-name1 : access-specifier Base-class-name
 {
    .......
 } ;
class Derived-class-name2 : access-specifier Derived-class-name1
 {
    .......
 } ;
```

Example 1:

```
# include <iostream.h>
class Base
{ protected : int x , y ;
  public:     void input ( int a , int b )
                { x = a ;
                  y = b ; }
              void display ( )
                { cout << x << endl << y ; }
} ;
class Derived1 : public Base
{ private : z ;
  public :  void inputz ( )
              { z = x + y ; }   //ok, use of x and y because they are inherited as protected
            void displayz ( )
              { cout << "z = " ; }
} ;
class Derived2 : public Derived1
{ int k ;
  public : void inputk ( )
             { k = x * y ; }  //ok , x and y are inherited as protected
           void displayk ( )
```

```
            {  cout  <<  " k =  " <<  k  ; }
} ;
main ( )
{  Derived1  ob1  ;
   Derived2  ob2  ;
   ob1 . input ( 3  ,  7 ) ;
   ob1 . display ( )  ;
   ob1 . inputz ( )  ;
   ob1 . displayz ( ) ;
   ob2 . input ( 2  , 10 ) ;
   ob2 . display ( )  ;
   ob2 . inputz ( )  ;
   ob2 . displayz ( ) ;
   ob2 . inputk ( )  ;
   ob2 . displayk ( ) ;    }
```

Note that ***ob1*** calls its functions and functions of ***Base.***

Note that ***ob2*** calls its functions, functions of ***ob1***, and functions of ***Base.***

Note: When a derived class is used as a base class for another derived class, any *protected* member of the initial base class that is inherited (as public) by the first derived class may also be inherited as *protected* again by a second derived class. For example, the program above is correct, and ***derived2*** does indeed have access to ***x*** and ***y***.

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **x** and **y** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

Example 2:

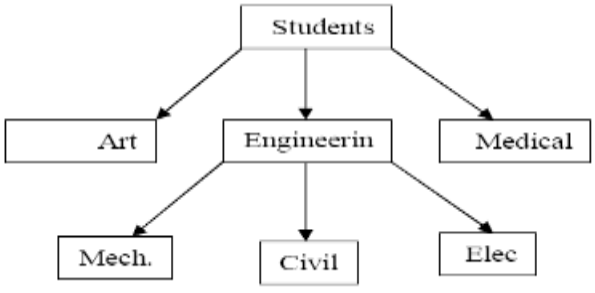
```
// This program will not compile.
#include <iostream.h>

class base
{    protected:  int x , y ;
     public:
          void set ( int a , int b )   { x = a  ;   y = b  ; }
          void show ( ) { cout << x << " " << y << "\n"; }
};

class derived1 : private base     // Now, all elements of base are private in derived1
{    int k ;
     public:
       void setk( ) { k = x * y ; }   // OK, this is legal because x and y are private to derived1
       void showk( ) { cout << k << "\n"; }
```

```
};

// Access to x, y, set( ), and show( ) not inherited.
class derived2 : public derived1
{   int m;
    public:
        void setm( ) { m = x – y ; } // Error, illegal because x and y are private to derived1
        void showm( ) { cout << m << "\n"; }
};

main()
{   derived1   ob1;
    derived2   ob2;
    ob1.set(1, 2);   // error, can't use set()
    ob1.show();   // error, can't use show()
    ob2.set(3, 4);   // error, can't use set()
    ob2.show();   // error, can't use show()
}
```

Note: Even though **base** is inherited as **private** by **derived1, derived1** still has access to **base**'s **public** and **protected** elements.

Hierarchical Inheritance:

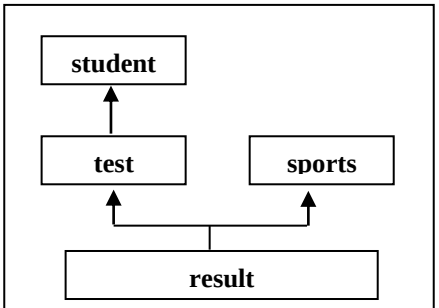
Many programming problems can be cast into a hierarchical where certain features of one level are shared by many others below that level. The following figure shows a hierarchical classification of students in a university.



Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program.

```
#include <iostream.h>
class student
```



```
{  protected:  int  r  ;
   public:
       void get_number( int a )
       {  r  =  a  ;    }
       void put_number( )
       { cout  << " r  = " <<  r  <<  endl  ;   }
};
class test : public student
{  protected :  float sub1;
                float sub2;
   public:
        void get_marks( float  x ,  float  y )
        { sub1 = x  ;   sub2 = y   ;   }
        void put_marks( )
        { cout  << " Marks in sub1  = " << sub1  <<  endl  ;
          cout  << " Marks in sub2  = " << sub2  <<  endl;    }
};
class sports
{  protected:  float score  ;
   public:
          void getdata ( float s )
             {  score  = s  ;  }
          void putscore ( )
             {  cout << " sports " <<  score <<  endl  ;    }
};
class result  : public test  ,  public sports
{  float   total  ;
   public:
        void display ( )
         { total  =  sub1  +  sub2  +  score  ;
           put_number ( ) ;
           put_marks ( ) ;
           putscore ( )  ;
           cout  <<  " total  = "  << total <<  endl  ;   }
} ;

main ( )
{  result   std  ;
   std.get_number (111)  ;
   std.get_marks (75.0 , 59.5 )  ;
   std.getdata ( 63.5 )  ;
   std.display ( )  ;
}
```