 Downloaded from: http://www.bsccsit.com/

Unit:3.3

Inheritance

Inheritance is like a child inheriting the features of its parents. It is a technique of organizing information in a hierarchical (tree) form.
Inheritance allows new classes to be built from older and less specialized classes instead of being written from the scratch. So classes are created by first inheriting all the variables and functions defined by some primitive class and then adding specialized variables and functions.

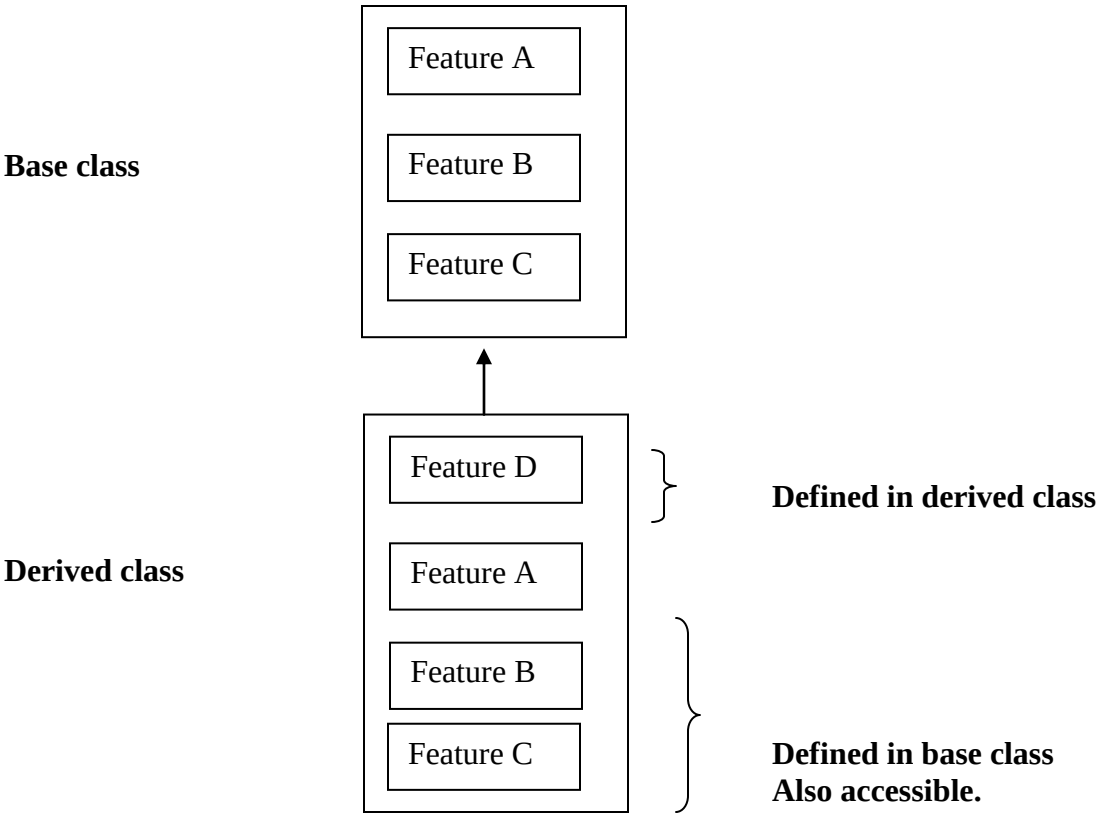
Base class and derived class:

Inheritance is a technique of building new classes from the existing classes. It is a technique of organizing information in a hierarchical from just like a child inheriting the features of its parents. Inheritance is a prime feature of oops

In the process of inheritance, the existing classes that are used to derive new classes are called base classes and the new classes derived from existing classes are called derived classes.

When a class is derived from a base class, the derived class inherits all the characteristics of base class and can add new features as refinements and improvements.

The great advantage of inheritance is reusability, which ensures ease of distributing class libraries. It improves the program reliability.
A base class is also called ancestor, parent or super class and derived class is also called as descendent, child or subclass.



Protected data member: Instead of private and public member there is another type protected member in a class. The purpose of making data member protected in a class is to make such member accessible from the function of the derived class. No other class than derived class function can access the protected data of a base class.

Derived class declaration: The derived class inherits all the features of its parent class and adds its own new features. The syntax of declaration of derived class is

```
class derivedclass  :  < private or public >  baseclass
{
        // member of derived class
        // other members.
};
```

Syntax:

```
class A                          class B : public A   // public derivation
{
    private :                    {
        // data members ;               //members of B
    protected :                  } ;
        //data members  ;
    public ;                             class  C : private A // private derivation
        //function members ;       {
                                             //members for c
} ;                              } ;
```

```
class D : A   // private derivation default
{
   //  member of D
} ;
```

```
// an example of inheritance
#include<iostream.h>
#include<conio.h>

class one //base class
{
      protected: //if private count will be inaccessible to derived class
            int count;
      public:
            one() {count=0;} //zero argument constructor
      void display()
      {
            cout<<"count="<<count<<endl;
      }

      void operator ++()//unary operator overloading
      {
            count++;
      }
```

```
};
class two: public one  //derived class
{
      public:
            void operator --()
            {
                  count--;
            }
};


void main()
{
      two i;
      ++i;
      ++i;
      i.display();
      --i;
      i.display();
      //one j; --j=? and ++j=? //not accessible
      getch();
}
output:
      count=2
      count=1
```

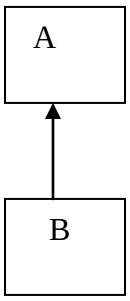
Analysis:

1. A protected member (i.e. int count) is introduced in base class. A private member can not be accessed by objects of a derived class. The protected members can be accessed by the member functions of derived class.
   e.g. `operator--( )` is defined in derived class two which can access the protected data 'count' of class one.
2. The derived class two inherits all the features of class members base class one.
3. In main( ), object of derived class is created(i.e. two i;) which can use both operator ++( ) defined in base class and operator --( ) in derived class. The derived class two which is child class gets member data count and member function operator ++( ) as inherit attribute of its parent
4. It is not possible to define an object of base class that use -- overloaded operator: since -- is member of derived class . The objects of base class can access the member of its own class.

Level of Inheritance

The level of inheritance refers to the length of its path from the root (top base class). A base class itself might have been derived from another classes in the class hierarchy. Inheritance is classified into the following forms based on the levels of inheritance and interrelationship among the classes in their hierarchy.

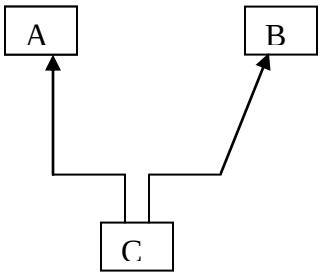
1. **Single Inheritance:** When a class is derived from only one base class, such derivation is called single inheritance. In single inheritance, base class and derived class exhibits one to one relationship. Following diagram exhibits the point



B is derived from class A.

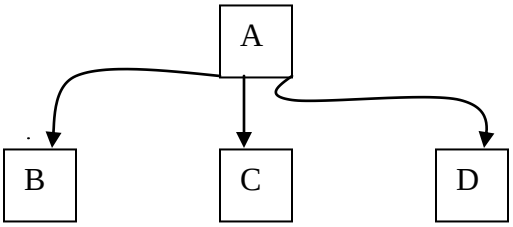
```
class  A            class B : A
{ ...               { ...
} ;                 } ;
```

2. **Multiple inheritance :** Derivation of a class from two or more base class is called multiple in heritance. In multiple inheritance, the derived class inherits some or all the features of base classes from which it is derived. Following figure shows the multiple inheritance.




```
Class  A        Class  B        Class C: A , B
{ ....          { ...           { ...
} ;             } ;             } ;
```

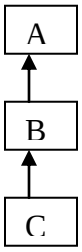
3 . **Hierarchical Inheritance :** When several (more than one) classes are derived from singled base class z-e. Feature of one class may be inherited by more than class, then it is called hierarchical inheritance. Following figure shows this type of inheritance.



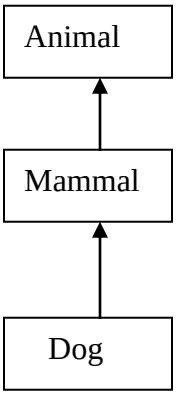

```
Class  B : A          Class  C : A
{ ....               { ....
} ;                  }
```

```
Class  D : A
{ ...
}...
```

4. **Multilevel inheritance :** The derivation of a class from another derived class called multilevel inheritance. In figure below, we can see multilevel inheritance

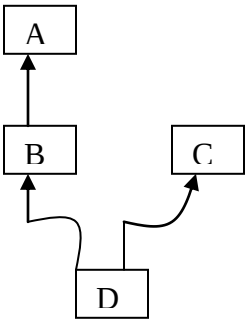


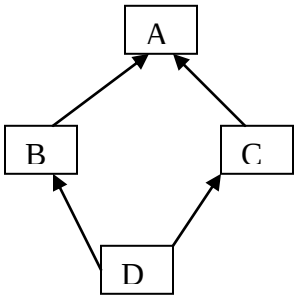
e.g

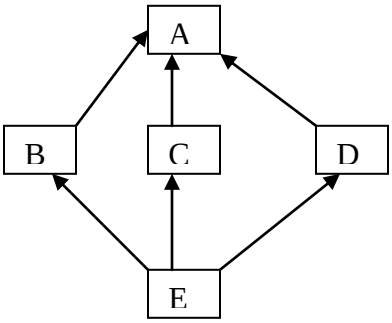


A serves as base class for derived class B, which is inturn serves as base class for class C. The class B is known as intermediate base class and the chain ABC is inheritance path.

5. **Hybrid inheritance :** Derivation of class involving more than one form of inheritance is know as hybrid in heritance see fly.



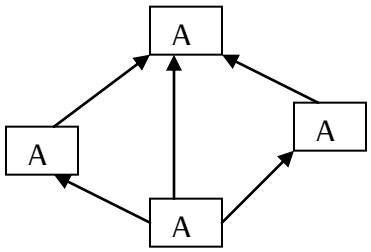




Hybrid (multilevel + multiple)

[multilevel , Hierarchical , multiple]
→ Hybrid

6. **Multipath inheritance :** Derivation of a class from other derived classes , which are derived from same base class is called multi-path inheritance. Fly.



Inheritance and member Accessibility:

- Private member of a base class can not be inherited directly to the derived class. The private member of a class is accessible only to the member function of its own.
- Making members of base class protected, they are accessible to the member functions of derived class.

- A protected member can be considered as a hybrid of a private and a public member. Like private members, protected are accessible only to its class member functions and they are invisible outside the class. Like public members, protected members are inherited by derived classes and also accessible to member function of derived class.
  Following points are to be considered when making members private, public or protected.

  1. A private member is accessible only to member of the class is which they are declared. They cannot be inherited.
  2. A private member of the base class can be accessed in the derived class through the member functions of the base class.
  3. A protected member is accessible to members of its own and any of the members in a derived class.
  4. The members that might be used in derived class should be declared as protected rather than private.
  5. A public member is accessible to members of its own class, member of derived class and even outside the class.
  6. Constructors of the base class and derived class are automatically invoked when the derived class is instantiated. If a base class has a constructor with arguments then their invocations must be explicitly specified in the derived class initialization section.

Public and private inheritance:

The visibility mode in the derivation of new class can be either public or private.

Public inheritance: When a class is derived publicly from its base class, the object of derived class can access public member of base class.

Private inheritance: When a class is derived privately, the objects of derived class cannot access public member functions of the base class. Since objects can never access private or protected members of a class, the result is that **no member of the base class is accessible to objects of the derived class.**

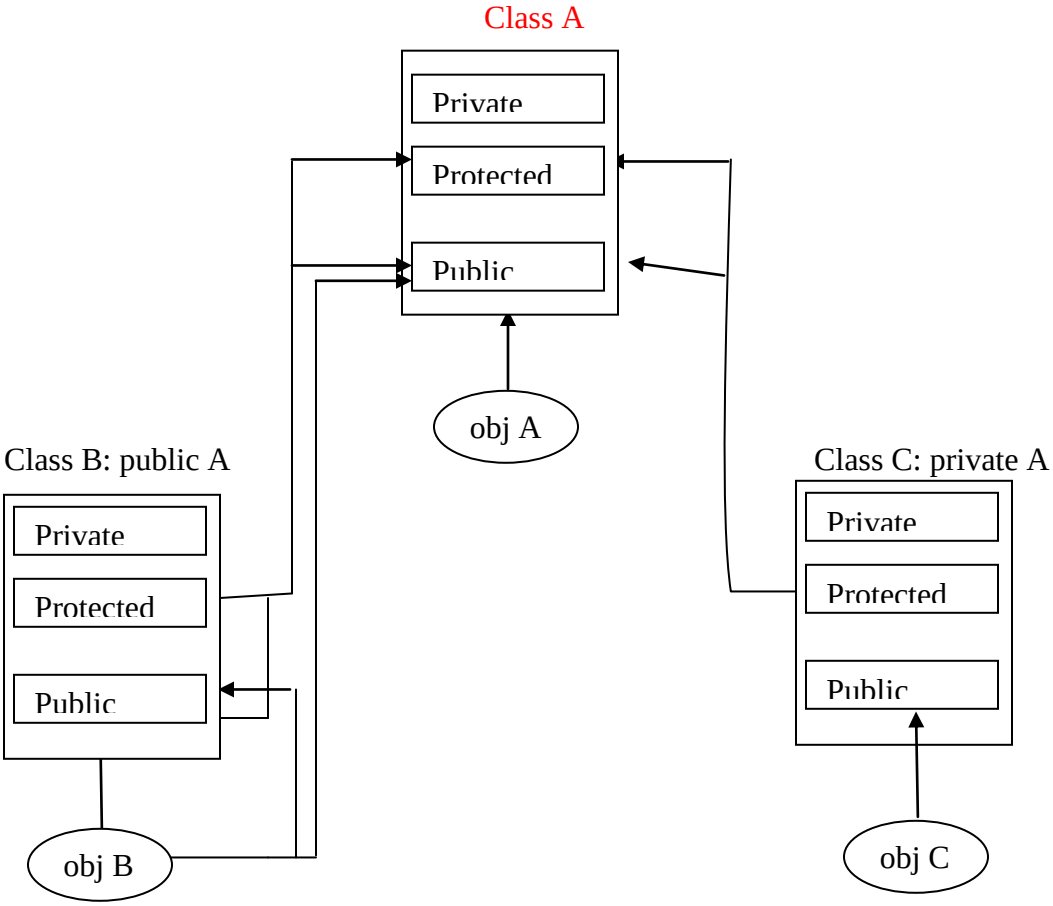
```
/*public or private inheritance*/
#include<iostream.h>
#include<conio.h>

class  A
{
       private :
            int pvtdataA ;
      protected :
            int protdataA ;
      public:
            int  pubdataA ;
} ;
```

```
class B  : public A // publicly derived
{
      public :
            void function ( )
            {
                  int a ;
                  //a = pvtdataA ; // error: not accessible.
                  a = protadataA ; // ok
                  a = pubdataA ;  // ok
            }
} ;

class C  : private A // privately derived.
{
      public :
            void funct ( )
            {
                  int  a ;
                  //a = pvtdataA ; // Error : not accessible
                  a = protdataA ; //  ok
                  a = pubdataA ; //   ok
            }
};

void main()
{
      int  a ;
      B objB ;
      //a = objB.pvtdataA ;  //  error : not accessible
      //a = objB.protdataA ; //  error : not accessible
      a = objB.pubdataA ; //   ok

      C objC ;
      //a = objC.pvtdataA;   // Error; not accessible
      //a = objC.protdataA;  // error; not accessible
      //a = objC.pubdataA;  //  error; not accessible A is private to C
      getch();
}
```

The program specifies a base class, A, with private, protected, and public data items. Two classes, B and C, are derived from A. B is publicly derived and C is privately derived. Functions in the derived classes can access private or protected members of the base class. Objects of the derived classes cannot access private or protected members of the base class.

Objects of the publicly derived class B can access public members of the base class A, while objects of the privately derived class C cannot. They can only access the public members of their own derived class.

Following figure shows the relationship and accessibility of members in private and public inheritance.



Member function accessibility:

Function type	Access directly to		
	private	protected	public
Class member	Yes	Yes	Yes
Derived class member func.	No	Yes	Yes
friend function	Yes	Yes	Yes
Friend class member	Yes	Yes	Yes

Derived Class Constructors:

The derived class need not have a constructor as long as the base class has a no-argument constructor. If base class has constructor with arguments (one or more) then derived class must have a constructor explicitly defined such that it passes arguments to the base class constructor.

In the application of inheritance, objects of derived class are usually created instead of the base class. So derived class should have constructor and pass arguments to the constructor of base class. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class is executed.

Constructor only in base class:

```
#include <iostream.h >
class B
{
      public:
      B( )
      {
            cout <<" No argument constructor of base class executed";
      }
};

class D : public B    // publicly derived
{
       public:
};


void main ( )
{
      D obj1 ;   // accesses base constrictor
}
```

```
Run : No argument constructor of base class executed.
```

Similarly, when constructor is present only derived class it is invoked at the object instantiation of derived class

Constructor in base and derived class:

```
class  B
{
      public:
      B(int a) { cout <<" One argument constructor in base class B";}
};

class D : public B
{
      D(int a)
       {cout <<" One constructor in derived class D";}
};

void main ( )
{
 D objd (3);
}
```

The compilation generates error like:
Cannot find "default" constructor to initialize base class 'B'

To overcome this error, explicit invocation of a constructor of base class in derived class constructor is needed.

```
class D : public B
{
       public:
                  D( int a ) : B (a)
                  {cout <<" One arg .Constrictor in derived class D ";}
};
```

Run : one – arg constructor in base class 'B'
one – arg constructor in derived class D.

Example:

```
//derived class constructor
#include<iostream.h>
#include<conio.h>

class one //base class
{
      protected:
            int count;
      public:
            one() {count=0;} //zero argument constructor
            one(int i) {count=i;}

   void display()
   {
      cout<<"count: "<<count<<endl;
   }
   void operator ++()//unary operator overloading
   {
      count++;
   }

};
class two: public one  //derived class
{
      public:
            two(): one()
            { }
            two(int i): one(i)
            { }
            void operator --()
            {
                  count--;
            }
};

void main()
{
      two i1;
```

```
   two i2(200);
   i1.display();  //displays 0
   i2.display(); //displays 200
   ++i1;
   ++i1;
   i1.display();   //displays 2
   --i2;
   i2.display();   //displays 199
   getch();
}
```

Constructor in multiple inherited classes with explicit invocation

```
#include<iostream.h >
class base1 // base class.
{
      public :
            base1()
            { cout << " No arg constructor in abase1 " ; }
};

class base2
{
      public :
            base2()
            {cout <<" No a rrg  Constructor in base2 "; }
};

class derived : public base1, public base2
{
      public :
      derived():base2(), base1()    //  explicit call
      { cout <<" No arg  constructor in derived class " ;}
};

void main ( )
{
      derived obj ;
}


Run :       No arg constructor in base1
            No arg constructor in base2
            No arg constructor in derived class
```

Here in above program, class derived is derived publicly from class base1 and base2 in order. Constructors of base classes are invoked explicitly in derived class as

derived ( ) : base2 ( ), base1 ( )

The order of execution of constructor is same as order as order of inheritance rather than order of explicit call.

Constructor invocation for data member initialization:

In multiple inheritances, the constructors of base classes are invoked first, in the order in which they appear in the declaration of the derived class.
In multilevel inheritance, constructors are invoked in the order of inheritance. For initialization of data members, the derived class object are created and values are supplied either by object of derived class or a constant value can be mentioned in the definition of the constructor.

The syntax of defining constructor in derived class

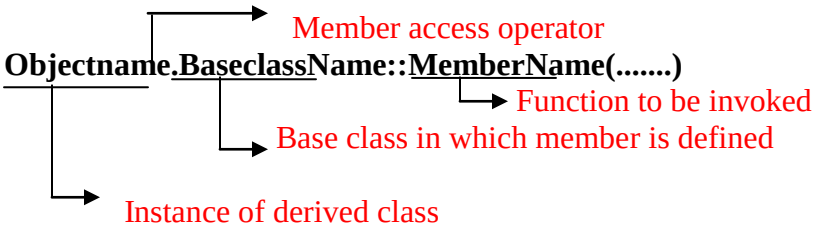
```
derived class (arg – list) : Base2 (arg – list2) , ..... , Base N (arg – list N)
{
        // body of derived class constructor

}
```

Ambiguity Resolution in Multiple inheritances:

Ambiguity is a problem that surfaces in certain situation involving multiple inheritances.

- Base classes having function with the same name.
- The class derived from these base classes is not having a function with the name as those of its base classes.
- Member of a derived class or its objects referring to a member, whose name is the same as those in base classes

The problem of ambiguity is resolved using the scope resolution operator as shown in figure.



Example: of Ambiguity

```
//amboguty in member access
#include<iostream.h>
#include<conio.h>

class  A
{
      public:
            void show() { cout<<" class  A "; }
};
class  B
{
      public:
            void show() { cout<<" class  B "; }
```

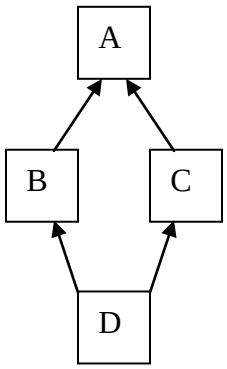
```
};
class C : public A , public B
{

};
void main()

{
      C objC;     // object of class C
      //objC.show() ;   //  ambiguous....error
      objC.A::show() ; // ok invokes show() in class A.
      objC.B::show() ;  // ok
      getch();
}
```

The statement objC.show(); is ambiguous as compiler has to choose A::show( ) or B:: show( ). It can be resolved using the scope resolution operator as follows

objc.A::show( );

refers to the version of show ( ) in the class A.

Another example:




```
class  A
{    public :
           void func ( ) ;
};
class B : public A.
{
     //......
};
class C :public A
{
     // .......
};
class D : public B, public C
{
     //……..
};
void main ( )
{
     D obj  D;
     obj D. fun ( ) ; // ambiguous :
}
```

Overriding Member Functions:

Defining a functions in derived class that have same name as those in base class is known as function (Member function) overriding. The member function in derived class overrides the function.

Overriding member functions, we can redefine a function in derived class that is previously defined in the base class, giving missing information. Following example shows the overriding of push ( ) and pop ( ) function in typical stack operation.

```
//implementation of stack
//overriding member functions
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define max 10

class stack
{
      protected:
            int s[max];
            int top;
      public:
             stack()   //constructor
            { top=-1;}

            void push(int x)//put number on stack
            {
                  s[++top]=x;
            }
            int pop()//take number from stack
            {
                  return s[top--];
            }
};

class stack2: public stack
{
      public:
            void push(int x)
            {
                  if(top==max)
                  {
                        cout<<"Stack Full";
                        getch();
                        exit(1);
                  }
                  stack::push(x);
            }

            int pop()
            {
                  if(top==-1)
                  {
                        cout<<"\nStack empty";
                        getch();
                        exit(1);
                  }
                  return stack::pop();
      }
};
void main()
```

```
{
      stack2 s;
      s.push(11);
      s.push(22);
      s.push(33);
      cout<<"\nNumber Popped"<<s.pop();
      cout<<"\nNumber Popped"<<s.pop();
      cout<<"\nNumber Popped"<<s.pop();
      cout<<"\nNumber Popped"<<s.pop();
      getch();
}
```

class stack2 is derived from class stack. Object of stack2 behave in exactly the same way as those of stack, except if attempt is made to push too many items on the stack, or to pop an item from an empty stack.

Another example:

```
#include<iostream.h>
#include<conio.h>
const int   SIZE = 25;
class teachers
{
   private:
         har empn[SIZE] ;
         nt numb ;
   public:
         void read()
         {     cout << " \nName : " ; cin >> empn ;
               cout << " \nNumber : " ; cin >> numb ;
         }
         void write ( )
         {
               cout << " \nName:"<< empn ;
               cout << " \nEmpNo:" << numb ;
         }
}; // end teachers.
class principal : public teachers
{
   private:
         char  desig [SIZE] ;
         float  deduct ;
   public :
         void read()
         {
               teachers :: read ( ) ;
               cout << " Designation: " ; cin >> desig ;
               cout << " Deduction : " ;  cin >> deduct ;
         }
         void write()
         {
               teachers :: write() ;
               cout << "Designation : " << desig ;
               cout << " Deduction : "  << deduct ;
         }
 } ;
```

```
class expert : public teachers
{
   private :
         int papers ;
   public :
         void read ( )
         {
               teachers :: read ( );
               cout << " How many papers " ;
               cin >> papers ;
         }
         void write ( )
         {
               teachers :: write ( ) ;
               cout << " in No . of papers : " << papers ;
         }
};
class    extra : public teachers
{  };
void main ( )
{
   principal p1, p2 ;
   expert e1 ;
   extra ex1 ;
   p1. read ( ) ;
   cout << " \n type in detail of VP : " ;
   p2 . read ( ) ;
   cout << " \n type in detail of expert: " ;
   e1. read ( ) ;
   cout << " \nPrinciple : " ; p1.write() ;
   cout << endl ;
   cout <<"\nV- principle:" ; p2. write() ; cout << endl;
   cout << " \nExpert : " ;      e1. write() ; cout  << endl ;
   cout << " \nExtra : " ; ex1. write() ;
   getch();
}
```

The base class has two member functions read ( ) and write ( )

- The first derived class principle declares its own private data desig and deduct and use : : operator to access read ( ) and write of base class.
- Similarly expert uses its own private data papers of type int and accesses read ( ) and write ( ) of base class
- The third derived class extra derived from teacher simply contains no members. Since it is derived from teacher it accesses read ( ) and write ( ) member function of base class.

Abstract Classes

The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. It is only designed to be inherited.

```
class One
```

```
{
        private:
                ……..
        public:
                ……..
};

class Two
{
        public:
                ……..
};

void main( )
{
        Two t1;
}
```

The class One serves as framework for building the derived class and it is treated as a member of the derived class Two. The instance of class One is not created in the function main( ), however it provides a framework for the class Two, so class One can be regarded as Abstract class.