2017

Object Oriented Programming

إعداد : م. واثق نجاح عبدالله

Non-structured programming

Non-structured programming is the historically earliest programming paradigm. It has been followed historically by procedural programming and then object-oriented programming, both of them considered as structured programming.

Unstructured programming has been heavily criticized (تعرضت للنقد) for producing hardly-readable ("spaghetti") code and is sometimes considered a bad approach for creating major projects, but had been praised(تمدح) for the freedom it offers to programmers.

There are both high and low level programming languages that use non-structured programming. These include early versions of BASIC (such as MSX BASIC and GW-BASIC), COBOL, and machine-level code.

A program in a non-structured language usually consists of sequentially ordered commands, or statements, usually one in each line. The lines are usually numbered or may have labels: this allows the flow of execution to jump to any line in the program.

Non-structured programming introduces basic control flow concepts such as loops, branches and jumps. Although there is no concept of procedures in the non-structured paradigm, subroutines are allowed. Unlike a procedure, a subroutine may have several entry and exit points, and a direct jump into or out of subroutine is (theoretically) allowed.

Procedural programming

It derived from structured programming, based upon the concept of the *procedure call*. Procedures, also known as routines, subroutines, methods, or functions simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself. Some good examples of procedural programs are the Linux kernel, and Apache HTTP Server.

Inputs are usually specified syntactically in the form of *arguments* and the outputs delivered as *return values*.

Scoping is another technique that helps keep procedures strongly modular. It prevents the procedure from accessing the variables of other procedures (and vice-versa), including previous instances of itself, without explicit authorization.

The focus of procedural programming is to break down a programming task into a collection of variables, data_structures, and subroutines.

Modular programming

Is a software design technique that increases the extent to which software is composed of separate, interchangeable components, called **modules** by breaking down program functions into modules, each of which accomplishes one function and contains everything necessary to accomplish this.

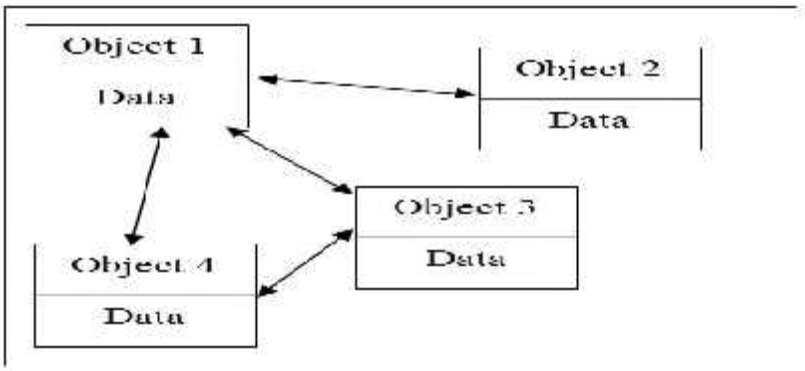
Languages that formally support the module concept include Ada, BlitzMax, Fortran, MATLAB, Python, and Ruby. Modular programming can be performed even where the programming language lacks explicit syntactic features to support named modules.

Theoretically, a modularized software project will be more easily assembled by large teams, since no team members are creating the whole system, or even need to know about the system as a whole.

Each module can have its own data. This allows each module to manage an internal state which is modeled by calls to procedures of this module.

Object- Oriented programming

In this technique, we have a web of interacting objects, each house keeping its own state.



Objects of the program interact by sending messages to each other. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.

OOP allow us to decompose a problem into a number of entities called object and then builds data and functions around these entities.

The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Some of features of OOP are:

• Emphasis is on data rather than procedure.
• Data is hidden and cannot be accessed by external functions.
• Programs are divided into what are known as objects.
• Objects may communicate with each other through functions.
• New data and functions can be easily added whenever necessary.

OOP can be defined as "OOP is an approach that provides a way to modularization programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand".

Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Basic Concepts of Object-Oriented Programming:

1- Objects. 2- Classes. 3- Data abstraction.
4- Data Encapsulation. 5- Inheritance. 6- Polymorphism.

Objects:

Objects are the basic run-time entities in an OO System. They may represent a person, place, a bank account, a table of data or any item that the program must handle.

- Each object contains data and code to manipulate the data.
- When a program is executed, the objects interact by sending messages.

Classes:

The entire set of data and code of an object can be made a user define data type that called a class.

Objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to that class. Classes are user defined data types and behave like the built-in types of programming language.

If a fruit has been defined as a class then the statement

fruit mango ;

will create an object (mango) belonging to the class (fruit).

Specifying a Class:

Defining class means creating a new abstract data type that can be treated like any other built in data type;

class specification has two parts:

1- Class declaration.
2- Class function definition.

The general form of a class declaration is:

```
class class-name
{
  private: variable declaration;
           Function declaration;
  public: variable declaration;
          Function declaration;
 };
```

Classes and Objects

Class: is a way to bind the data and its associated functions together, it allows the data and functions to be hidden if necessary from external use when defining a class it means creating a new abstracted data type.

Note that the key feature of OOP is data hiding using private declaration.

- The binding of data and functions together into a single class-type variable is referred to as **encapsulation**.

- Usually the data members are declared as private and the member functions as public.

Creating Objects:

Class variables are called objects; for example:

```
item X ;   //memory for X is created
```

To create more than one object use

```
item  x  ,  y  ,  z  ;
```

Accessing Class Members:

The private data cannot be used by **main** function they are accessed only through class functions.

A ***public*** member is accessible from anywhere within a program. It represents the interface to the outside world.

A ***private*** member can be accessed only by the member functions. A class that enforces information hiding declares its data members as private. The word ***private*** can be omitted.

We should try to limit or eliminate the use of public variables. Instead, we should make all data private and control access to it through public functions.

Another class section is called ***protected***. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessible by the functions outside these two classes.

The three concepts of OOP languages are:

1. **Encapsulation**: It is the mechanism that binds together code and data it manipulates and keeps both safe from outside interference and misuse.

   With encapsulation, we can accomplish *data hiding* in which an object can be used without knowing or caring how it works internally.

   C++ supports the properties of encapsulation and data hiding through the creating of classes.

2. **Inheritance**: it is a relationship among classes that allows one object to take on properties of another. Typically, OOP uses inheritance to construct new class from an existing class; the new class can be described as “a type of” the existing class.

The concept of *inheritance* provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. The new class will have the combined features of both the classes.

For example:

The class of a rectangle contains data members: (*length* and *width*). A new class *Square* will have similar data members with the special case (length = width). We also need a member function to calculate the area of a square. Instead of defining the class (*square*) from scratch, we can think of square as a special case of a rectangle. However, we can use the class of a rectangle by inheriting its behavior and redefining the area function to work for the class of (*square*).

3. **Polymorphism**: It is the quality that allows one name to be used for two or more related but technically different purpose. Such as redefining member functions to define a new behavior with different number and/or parameters. It simply means "one name, multiple forms".

Example1:

```
class item
{  int number;
   float cost;

    public:
     void getdata ( int a  , float  b)
         {  number = a  ;  cost = b  ;  }
     void putdata ( )
         { cout << number   << "\n"  <<  cost  ;  }
};

 item x ; // x is an object
```

To access the above functions from main function use:

Object-name . function-name (actual-arguments);

To access a member of a class use:

Object-name . member

The dot (.) operator is also called (*class member access operator*).

Therefore function call statement will be

```
x . getdata (6 , 75.5 ) ;
```

While the following statement

```
x . number =100 ;   // is illegal
```

Since ***number*** declared as private in the class therefore it could be accessed only through a member function and not by the **object** directly.
The objects communicate by sending and receiving messages. This is achieved through the member function, for example:

```
x.putdata( ) ; // Send message to the object x to display its data.
```

While the variables which are defined in public section of class could be accessed by the object directly.

Example 2:

```
# include < iostream.h >
class xyz
{  int   x ;        int   y  ;

  public:  int z;

  };

main( )
{  xyz   p ;

   p.x = 0 ;   // error x is private

   p.z = 10 ;  // ok, z is pubic
 }
```

Function Definition:

1- Inside Class.
2- Outside Class.

Inside Class:

Example 3: Class of Rectangle

```
# include < iostream.h >
class Rectangle
{ public: int length , width ;
            int area ( )
              { return length * width ; }
} ;
```

```
main ( )
{ Rectangle R ;
  R.length = 8 ;
  R.width =  5 ;
  cout << R.area( ) ; }
```

H.W. Write a program with a rectangle class whose its length and width are private (note: that you should write an additional function to enter length and width).

Outside Class:

The general form is:

Return type Class-name :: function-name(argument declaration)
{
Function body
}

Note: the operator (**::**) can be referred to as *Scope Resolution Operator*.

Example 4: Write a program to use a model of employee class (its data are: name, age, and department. Its functions are input( ) and display( ).
We can write the function definitions outside the class.

```
# include < itostream.h >
class employee
{ private:
    char name [30] ;
    int age ;
    char dept [10] ;
  public: void input ( ) ;
          void display ( ) ;
} ;

void employee :: input ( )
{ cin >> name ;
  cin >> age ;
  cin >> dept ; }

void employee :: display ( )
{  cout << name << endl << age  << endl << dept  ;  }
```

```
main ( )
{ employee E ;
  E . input ( ) ;
  E . display ( ) ;
 }
```

Note: if more than one object be created from the same class the process is called ***multiple objects***.

Example:

```
employee   A ,  B  ;
```

Example 5:

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

        void getdata ( )
         { cin >> name ;
           cin >> age ;
           cin >> salary ; }

        void putdata ( )
         { cout << name << endl ;
           cout << age << endl ;
           cout << salary << endl ; }
};

 void main ( )
{ employee doctor , nurse , worker ;

  doctor . getdata ( ) ;
  nurse . getdata ( ) ;
  worker . getdata ( ) ;

  doctor . putdata ( ) ;
  nurse . putdata ( ) ;
  worker . putdata ( ) ;     }
```

Example 6:

```
# include < iostream.h >
class set
{ private: int a , b ;
           void input ( )
           { cin >> a;
             cin >> b ; }
           void outp ( )
           { cout << a << endl ;
             cout << b << endl ;  }
 } ;
void main ( )
{  set   S  ;
   S . input ( ) ; // error because input is private function can not be accessed in main function

  S . outp ( ) ; // error because outp is private function can not be accessed in main function
}
```

* In order to avoid the above errors, the functions (*input* and *outp*) must be defined as public in class *set*. In such case, we can use these functions in main( ).

Memory Allocation of a Class

Once the member functions are defined as a part of a class, they are placed in the memory space.

Since all objects of the same class use the same member functions, no separate space is allocated for member functions when the objects are created. For each object, separate memory locations are allocated only for member data because member variables hold different data values for different objects.

Constructor

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. A constructor is a function that is executed automatically whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

Example 1:

```
class point
{ int m , n ;
  public:
        point ( )     // constructor
        {
          m = 0 ; n = 0 ;
        }
} ;
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example

```
point  X1 ;  //not only creates the object X1 of type point but also initializes it data members m and n to  zero
```

This is useful because the programmer may forget to initialize the object after creating it. It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it's created. The constructor does this.

A constructor with no parameters is called (*Default Constructor*) and a constructor that can take arguments is called (*Parameterized Constructor*).

The constructor functions have some special characteristics:

1- They should be declared in public section.
2- They are executed automatically when the objects are created.
3- They do not have return types, not even void and therefore, they cannot return values.

Parameterized Constructors

C++ permits to pass argument to the constructor function when the objects are created.

Example 2 : Rectangle Class with constructor

```
# include < iostream.h >
class Rectangle
{ int length , width ;
  public:
    Rectangle ( int a , int b )      // constructor
    {  length = a  ;
       Width = b  ; }
     int area ( )
     {  return length * width ;  }
} ;
main ( )
{ Rectangle R (10 , 4 ) ;    // implicit call
  cout << R . area ( ) ;
}
```

The initial values passed as arguments to the constructor function when an object is declared this can be done in two ways (types of constructor calling):

1- By calling the constructor explicitly.

2- By calling the constructor implicitly.

The following declaration illustrates the first method:

Rectangle R1 = Rectangle (9 , 2) ; //*explicit* call

This statement creates a Rectangle object R1 and passes the values 9 and 2 to it.

The second method is implemented as follows:

Rectangle R1 (9 , 2) ; // *implicit* call

This method, sometimes called the *shorthand* method, is used very often as it is shorter, looks better and is easy to implement.

Multiple Constructors in a Class

Example 3:

```
# include < iostream.h >
class AB
{ private: int x , y ;
  public:
        AB( )
        {  }
```

```
        AB ( int z , int r )
        {  x = z ;
           y = r ; }
} ;
main ( )
{ AB  m ;   // call constructor1
  AB  n ( 9 , 2 ) ; // call constructor2
}
```

When more than one constructor function is defined in class the process is called *constructor overloading*, as in the following example. Generally, the process of defining more than one function having the same name is called *function overloading*.

In the above example there is the empty constructor which is do nothing constructor (defined to satisfied (يرضي) the compiler), since there is the parameterized constructor, the empty constructor must be defined otherwise the compiler will show error message.

Example 4: shows *multiple constructors* and *scope resolution operator*.

```
# include < iostream.h>
class Rectangle
{ public: int length , width ;
          Rectangle ( )
           { length = 0 ;
             width = 0 ; }
          Rectangle ( int x , int y )
          {  length = x ;
             width = y ; }
          int area ( ) ;
          void show ( ) ;
 } ;
int Rectangle :: area ( )
{  return length  *  width  ; }

void Rectangle :: show ( )
 { cout << length ;
   cout << width ; }

main ( )
{ Rectangle R1 , R2 ( 20 , 8 ) ;
  R1 . length = 35 ;
  R1 . width  =  12 ;
```

```
R1 . show ( ) ;
R2 . show ( ) ;
cout << " area  of  R1 = " << R1 . area ( ) ;
cout << " area  of  R2 = " << R2 . area ( ) ;
}
```

H.W : Write a program that uses the above program but all data are private.

Destructor

A destructor is used to destroy the objects that have been created by a constructor.

The destructor is a member function whose name is the same as the class name but it preceded by a tilde (~) symbol.
For example the destructor for the class *Rectangle* can be defined as:

```
~ Rectangle ( )
  {     }
```

The destructor never takes any argument nor does it return any value.
It will be called implicitly by the compiler upon exit from the program to clean up the storage that is no longer accessible.
Unlike constructors, a class may have at most one destructor.

Example 1:

```
# include < iostream.h >
class ABC
{ public : int x , y ;
  public : ABC ( )
           { x = 2 ;
             y = 7 ; }

           ~ ABC ( )
             { }
 } ;
main ( )
{ ABC m ;
  cout <<  m . x << endl ;
  cout <<  m . y << endl ;
}
```

Output : 2
7

Example 2 : Employee class whose member data (name , age , salary , dept). Its member functions (input and show). Using constructor and destructor.

```
# include < iostream.h>
const  int  size  =  30 ;

class employee
{ public:  char  name [ size ] ;
           int  age , salary ;
           char  dept [ size ] ;

           employee ( )
             {      }

          ~ employee ( )
             {    }

           void input ( )
           { cin >> name ;
             cin >> age ;
             cin >> salary ;
             cin >> dept ;
           }

           void show ( )
           { cout << name << endl ;
             cout << age << endl ;
             cout << salary << endl ;
             cout << dept << endl ;
           }
   } ;

main ( )
{ employee emp  ;
  emp . input ( ) ;
  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*.

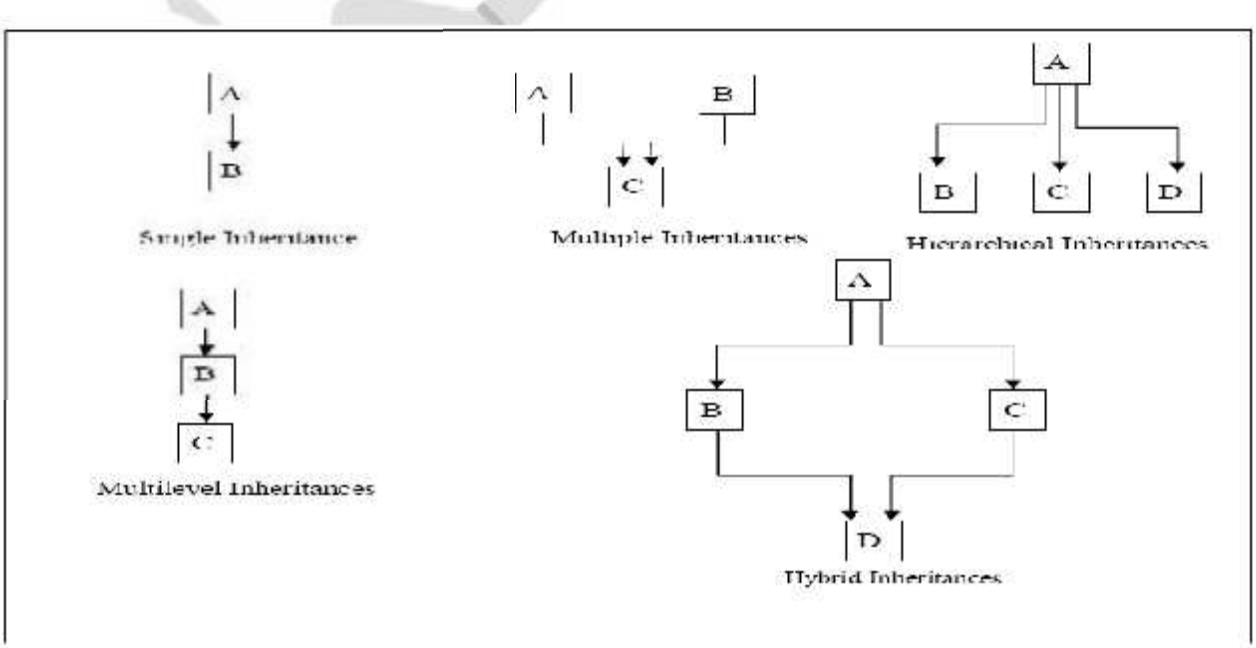
The old class is referred as the *base class* (*super class*) and the new one that inherits the properties of the base class is called the *derived class* (*subclass*).

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.

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:

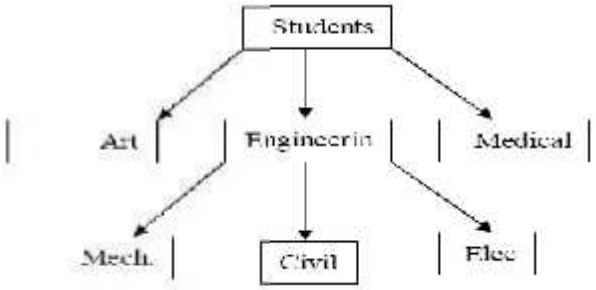
Example 2: // This program will not compile.

```
#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.

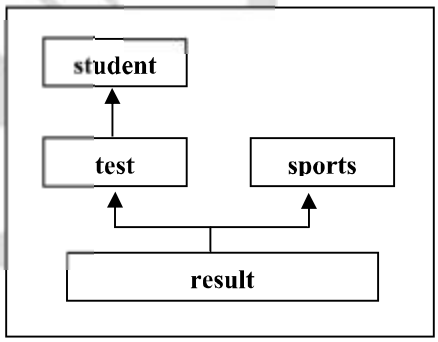
Hierarchical Inheritance:

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   std1  ;
   std1.get_number (111)  ;
   std1.get_marks (75.0 , 59.5 )  ;
   std1.getdata ( 63.5 )  ;
   std1.display ( )  ;
}
```

Friend Function

The private member cannot be accessed from the functions that are not member to the class. However, there could be a situation where two classes need to share a particular function.

In such situations C++ allows the common function to be made *friendly* with both the classes thereby allowing the function to have access to the private data of these classes.

Such a function need not be a member of any of these classes.

To make an outside function "*friend*" to a class it will be declared as in the following example:

Example 1:

```
class ABC
{    public:
          friend void xyz ( ) ;  //declaration or prototype
 }
```

The function declaration should be preceded by the keyword *friend.*

The function definition does not use either the keyword friend or the scope operator (::).

A friend function has certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class it can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

Example 2:

```
class sample
{  private:  int x , y  ;
   public:
        sample ( int a , int b) {  x = a  ;  y  =  b  ;  }
        friend float mean (sample) ;
} ;
float  mean ( sample s )      // note that there is no (::) operator
{  return ( s.x  +  s.y  )  /  2.0  ;  }    // it uses s.x  instead of x

 main ( )
{ sample  a ( 25 ,40 )  ;
  cout << "mean value = " << mean (a) << endl  ;
 }
```

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

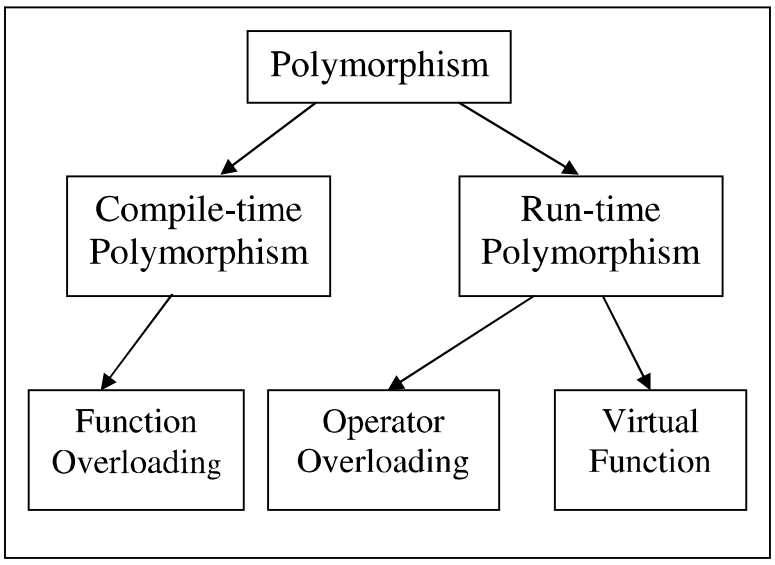
Example 3:

```
 class X
{    ....
     .....
     int  fun1 ( )  ;   //member function of X
 } ;

 class Y
 {   .......
      ......
     friend  int  X :: fun1( )  ; //The function fun1( ) is a member of class X and a friend of Y
      ......
  }  ;
```

We can also declare all the member function of one class as the friend functions of another class. In such case, the class is called a *friend class*.

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means "one name, multiple forms".



Function Overloading

Several different functions can be given the same name. Function overloading is one of the most powerful features of C++ programming language. It forms the basis of polymorphism (compile-time polymorphism).

The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called ***early binding*** or ***static binding*** or ***static linking***. Also known as ***compile time polymorphism***, early binding simply means that an object is bound to its function call at compile time.

Function Overloading: is the process of using two or more functions with the same name but differing in the signature (the number or type of arguments or both).

To avoid ambiguity, each definition of an overloaded function must have a unique signature.

All of you know that we cannot have two variables of the same name, but can we have two Functions having the same name. Such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

In overloaded functions, the function call determines which function definition will be executed.

The advantages of overloading are:

- It helps us to perform same operations on different data-types without having the need to use separate names for each version.

- The use of overloading may not have reduced the code complexity /size but has definitely made it easier to understand and avoided the necessity of remembering different names for each version function which perform identically the same task.

Example1: We have two functions with the same name: *calc*
We have different signatures: *(int) , (int , int)*
Overloading Functions differ in terms of *number of parameters*

```
#include <iostream.h>
class arith
{  public:
     void calc ( int num1)
     { cout << " Square of a given number: " << num1 * num1 << endl  ; }

    void calc ( int num1, int num2 )
     { cout << " Product of two numbers: " << num1 * num2  << endl ;  }
} ;
main ( )
{ arith a;
  a . calc ( 5 ) ;       // Based on the arguments we use when we call the calc function, the
  a . calc ( 6 , 7 ) ;  // compiler decides witch function to use at the moment we call the function
}
```

Example 2: Overloading Functions differ in terms of *type of parameters*

```
# include <iostream.h>
 int  func ( int  i ) ;
 double  func ( double  i ) ;

 main ( )
 {    cout <<  func ( 10 ) ;   //  func(int i)  is called
      cout <<  func (10.24 ) ;  //  func (double i) is called
```

```
}
int  func ( int  i )
{       return  i  + 5 ;      }

double  func ( double  i )
 {    return  i  * 8  ;      }
```

Example 3: Is the program below, valid ?

```
#include<iostream.h>
 int  func ( int  i )
 {      return i   ;       }

 double  func ( int  i )
 {
  return i;
 }

 void main ( )
 {
  cout<<func(10);

  cout<<func(10.201);
 }
```

No, because you can't overload functions if they differ **only** in terms of the data type they return.

H.W. Write a program to use three functions (sum) to find the summation of two numbers.

Operator Overloading

C++ tries to make user-defined data types behave in much the same way as the built-in types. For instance, C++ permits to **add** two variables of user defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with special meaning for a data type. The mechanism of giving such special meaning to an operator is known as ***"operator overloading".***

All C++ operators can be overloaded (given additional meaning) except the following:
- class member access operator (**.**)
- scope resolsution operator (::).

- size operator (sizeof)
- conditional operator (? :)

When an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Although the semantic of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associatively. For example: the multiplication operator is higher precedence than the addition operator.

Defining Operator Overloading:

The general form of an operator function is

```
Return type classname :: operator op ( arguments-list )
{
      Function body

}
```

Operator is reserved word and (op) is the operator being overloaded, ***operator*** op is the function name.

Note that operator functions can be either member function or friend function.

A basic difference between them is that a friend function take two argument for binary operators and one for unary operator while the member function take one argument for binary operator and no argument for unary operator.

This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with the friend function.

The process of overloading involves the following steps:

1- First, create a class that defines the data type that is to be used in the overloading operation.
2- Declare the operator function ***operator op( )*** in the public part of the class. It may be either a member function or a friend function.

3- Define the operator function to implement the required operations.

Overloading Unary Operators

Example: A minus operator, when used as a unary, takes just one operand

```
void  space :: operator - ( )
{  x  =  - x  ;
   y =  - y  ;
   z =  - z  ;
}
void  space :: getdata ( int a  , int b  ,  int c )
{ x =  a  ;
  y =  b  ;
  z  =  c  ;       }
void  space :: display ( )
{
 cout << x <<  " " << y << " " << z <<  "\n"  ;    }
main ( )
{ space  s  ;
  s . getdata ( 10 , -20 , 30 )  ;
  cout  <<  " s : "  ;
  s . display ( )  ;
  s . operator - ( )  ;     //   or   -s ;
  cout  <<  " s :  "    ;
  s . display ( ) ;
}
```

Note that statement like: s2 = - s1 ;
Will not work because the function operator – ( ) does not return any value. It can work if the function is modified to return object.

Overloading Unary Operators using Friends

To overload a unary minus operator using a friend function as follow:

```
void operator  - ( space & ss )
{ ss . x  = -  ss . x  ;
  ss . y  = -  ss . y  ;
  ss . z  = -  ss  . z  ;
}
```

Overloading Binary Operators:

The following example explain how to perform overloading for + operator

```
class complex
{  float  x  ;
   float   y  ;
   public:
         complex ( )
           {   }

         complex ( float real  ,  float image )
          {  x  =  real  ;
             y =  image  ;    }

         complex  operator + ( complex C ) ;

        void display( )
          { cout << x << " , " << y << "\n" ;  }
  } ;
complex  complex :: operator + ( complex C )
{  complex temp;
   temp . x = x  +  C . x  ;
   temp . y = y  +  C . y  ;
   return  temp ;
}
main ( )
{ complex  c1 ,  c2  ,  c3  ;
  c1 = complex ( 2.5 ,  3.5  )  ;
  c2 = complex ( 1.6 ,  2.7 )  ;
  c3 = c1 + c2  ;    //  or c =c1.operator + ( c2 ) ;
  cout << "c1= "  ;  c1 . display ( )  ;
  cout << "c2 = "  ;  c2 . display ( )  ;
  cout << "c3 = "  ;  c3 . display ( )  ;
}
```

Overloading Binary Operators using Friends

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

1- Replace the member function declaration by the friend function declaration
   friend complex operator + (complex , complex)
2- Redefine operator functions as follows:

```
complex operator + ( complex  a  , complex  b )
{ complex  temp  ;
  temp . x = a . x  + b . x  ;
```

```
    temp . y = b . y  +  b . y  ;
    return  temp  ;
  }
```

In this case the statement

```
c3  =  c1  +  c2  ;
```

Is equivalent to

```
c3  = operator + ( c1 , c2 )  ;
```

Note that the friend and member functions give the same result. Why then these alternatives? Consider a situation where one operand of binary operator is object and the second is build in data type as shown below:

```
A  =  B + 2  ;   ( or A = B * 2  )
```

This will work in member function and A , B object of the same class.

But the statement:

```
A = 2 + B  ;   (or A = 2  * B )
```

Will not work, this is because the left hand operand which is responsible for invoking the member function should be object of the same class. *Friend* function solves this problem.

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects.

Sometimes, we don't know, at the time we write the program, how many objects we want to create. When this is the case we can use ***new*** to create objects while the program is running.

Object pointers are useful in creating objects at run time. We can also use the object pointer to access the public members of an object.

```
Item x;
Item* ptr;
Item *ptr=&x;
```

Ptr->show () ; it is equivalent to (*ptr)->show();

We also can create the objects using pointers and new operator as follows:

```
Item*ptr=new Item;
```

```
ptr->show ( ) ;
Item*ptr  = new Item[10]  ;
ptr [0] -> show ( ) ;
```

When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator.

The next program illustrates how to access an object given a pointer to it:

Example 1:

```
class distance
{   private: int  feet  ;
                float inches ;

    public:  void  getdist ( )
                { cin >>  feet  ;
                  cin  >> inches  ;   }
                void  showdist ( )
                {  cout  << feet  <<  "\n"  <<  inches    ;     }
} ;
main ( )
{ distance  dist  ;
  distance * distptr  ;   // pointer to distance
  dist . getdist ( ) ;
  dist . showdist ( ) ;

  distptr  =  new  distance ;   // points to new distance object
  distptr  ->  getdist ( )  ;     // access object members with  -> operator
}
```

Note: that we can not refer to the member functions in the object pointed by distptr using the dot (.) membership-access operator, as in

```
distptr . getdist ( ) ;  // will not work,  distptr is not a variable
```

The dot operator requires the identifier on its left to be a variable. Since *distptr* is a pointer to a variable, we need another syntax.

One approach of dereference (get the contents of the variable pointed by) the pointer:

```
( *distptr ) . getdist ( ) ;   // ok but inelegant
```

The parentheses are necessary because the dot (.) has higher the indirection operator (*). An equivalent but more concise approach is by using the membership-access operator ->.

```
distptr ->  getdist ( ) ;  // better approach
```

Example 2: An Array of pointer to object

```
# include  <iostream.h>
class item
{  int  code  ;       float   price  ;
   public:
         void  getdata ( int a  , float  b )
          { code  = a  ;    price = b  ;  }
          void  show ( )
         { cout <<  code  <<  endl  ;
           cout <<  price  <<  endl  ;   }
} ;
const  int  size  =  2  ;
main ( )
{ item *p  =  new  item [size]  ;
  item *d  = p ;  // another pointer to first location in item in order not to loss the address of location
   int  x  ,  i  ;
   float  y  ;
   for ( i = 0  ;  i <  size  ;  i ++  )
   { cout << "input code and price for item"  <<  i+1  ;
     cin >>  x  >>  y  ;
     p -> getdata ( x , y ) ;
     p ++  ;
   }

  for ( i = 0  ;  i < size  ;  i ++ )
   { cout  <<  "item: "  <<  i+1  <<  endl  ;
     d ->  show ( )  ;
     d ++  ;
    }
}
```

Pointers to Derived Class

It is means that a single pointer variable can be made to point to objects belonging to different classes, for example, if B is a base class to the derived class D then

declaring a pointer pointing to the base class can also be used to point to the derived D.

```
B *ptr  ;
B b ;
D d ;
ptr = &b  ;
ptr = &d ; // This is perfectly valid because d is an object derived from the class B
```

Example 3:

```
# include <iostream.h>
class B
{  public :    void display ( )
                 { cout << " hello B "  <<  endl ;  }
} ;
class  D :  public  B
{ public  :
             void display()
            { cout  << " hello D " << endl  ;   }
} ;
main ( )
{ B a   ;    D b   ;
  B *p  ;
  p = &a  ;
  p -> display ( )  ;
  p =  &b  ;
  p ->  display ( )  ;   // call to display in base class not in derived
}
```

Note: There is a problem in using p to access the public members of the derived class. Using p, can access only the members inherited from the class B and not the functions in the derived class. In case the derived contain the function in the same name of the base class then any reference to that member by pointer p will always access the base class member.

Virtual Functions

An essential requirement of polymorphism is the ability to refer to objects without any regard to their classes. This necessitates (يوجب- يلزم) the use of a single pointer variable to refer to the objects of different classes.

Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call.

Virtual functions can be used when a member function is called through a pointer of the type *pointer to a base class*. The function called will be the function of that name in the *derived class*, even though the pointer is declared as a pointer to the base class declared to be *virtual*.

The function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration.
C++ will determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, then we could execute different versions of the virtual function.

Rules for virtual functions:

1- The virtual function must be member of some class.
2- They are accessed by using object pointer.
3- A virtual function can be friend of another class.
4- A virtual function in the base class must be defined even if it will not use.
5- The prototype of virtual function in the base class and derived class must be identical, if two functions with the same name has different prototype C++ considers them as overloaded functions not as virtual function (ignored).
6- While a base pointer can point to any type of derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access an object of the base type.
7- If the virtual function is defined in the base class. It need not be necessarily redefined in the derived class. Calls will invoke the base function.

Example 1: The following example illustrates the difference between a normal member function and virtual function.

```
# include <iostream.h>
class base
{ public:
      virtual  void  show ( ) { cout << "base class\n"; }
      void  ABC ( ) { cout << "ABC from base \n"; }
} ;

class derv1 : public base
{  public:
       void  show ( ) { cout << "derv1 class\n" ; }
```

```
        void  ABC ( ) { cout << "ABC from derv1 \n" ; }
} ;

class  derv2 : public  base
{  public:
        void  show ( ) { cout << "derv2 class\n" ; }
        void  ABC ( ) { cout << "ABC from derv2 \n" ; }
} ;
main ( )
{ derv1 D1 ;
  derv2 D2 ;
  base *ptr ;

  ptr = &D1 ; // pointer to derived
  ptr -> show ( ) ; // which show method do we get? ans: derv1's show
  ptr -> ABC ( ) ; // which ABC method do we get? ans: base's ABC

  ptr = &D2  ;
  ptr -> show ( ) ; // call derv2's draw
  ptr -> ABC () ; // call base's ABC

  base *list[10] ;  // declare an array of 10 pointers to base
  list[0] = new derv1 ;    // set list[0] to point to derv1
  list[1] = new derv2 ;   // and list[1] to point to derv2

  for (int i = 0 ; i < 2 ; i++)
    list[i] -> show ( ) ;   // call draw method of the appropriate derived class
}
```

Example 2:

```
# include <iostream.h>
class base
{  public:  virtual void show ( )          // virtual function
              {  cout << " base \n "  ; }
}  ;
class  Derv1 :  public  base
{ public :
            void  show ( )
             { cout <<  " Derv1  \n "  ; }
}  ;
class  Derv2 :  public  base
{ public :
            void  show ( )
```

```
            { cout <<  " Derv2  \n "  ; }
} ;

main ( )
{ Derv1  dv1  ;
  Derv2  dv2  ;
  Base   * ptr  ;   // pointer to base class

ptr =  & dv1  ;  // put address of dv1 in pointer
ptr -> show ( )  ;  //  call derv1 's show

ptr =  & dv2  ;  // put address of dv2 in pointer
ptr -> show ( )  ;  //  call derv2 's show
}
```

The same function call *ptr -> show* ( ) executes different functions, depending on the contents of *ptr*. The rule is that the compiler selects the function according to the *contents* of the pointer, not on the type of the pointer.

Note:

A class that declares or inherits a virtual function is called a polymorphic class.

Early Binding and Late Binding

In normal function (Non-Virtual), the compiler has no problem with the expression

ptr->show( ) ; // it always call to the *show* function in the base class.

But in ***virtual*** the compiler doesn't know what is the contents of ***ptr***. It could be the address of an object of the Derv1 class or of the Derv2 class.

At run-time, when it is known what class is pointed to by ***ptr*** the appropriate version of ***show( )*** will be called. This is called late binding or dynamic binding. (Choosing functions in the normal way, during compilation, is called early binding, or static binding). Late binding requires some overhead but provides increased power and flexibility.

Pure virtual function & Abstract Class

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function in the base class is seldom used for performing

any task. Such functions are called “do nothing” functions or called pure virtual function.

A “do nothing” function may be defined as follows:

```
virtual void display ( ) = 0 ;
```

The value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a function will be pure.

We can not create objects of abstract class.
Note that, although this is only a declaration you never need to write a definition of the base class.

A pure virtual function is a function declared in the base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.
Any class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called ***abstract base classes***.

The main objective of an abstract base class is to provide some traits (ميزات) to the derived classes and to create a base pointer required for achieving runtime polymorphism.(i.e.,: we can create pointers to an abstract class and take advantage of all its polymorphic abilities.

The main difference between an abstract base class and a regular polymorphic class is that we cannot create instances (objects) of it.