Classes and Objects

C++ structures and classes are identical in terms of their functionality. That is, al the OOP concepts such as data abstraction, encapsulation, inheritance and polymorphism can be implemented in both. Thus, they can be used interchangeably with some modifications. However, to follow the conventions of object oriented programming, classes are generally used to contain data and functions both and structure are generally used to contain the data only.

A class serves as a template that provides a layout common to all of its instances known as Object. That is, a class is only logical abstraction that specifies what data and functions its objects will have, whereas the object are the physical entities through which those data and functions can be used in a program. Thus, objects are considered as the building blocks of object oriented programming.

Declaration of Class

A class is a user defined data type that binds data and functions that operate on the data together in a single unit. Like other user defined data types, it also needs to be defined before using its objects in the program. A class definition specifies a new data type that can be treated as a build in data type.

Syntax,

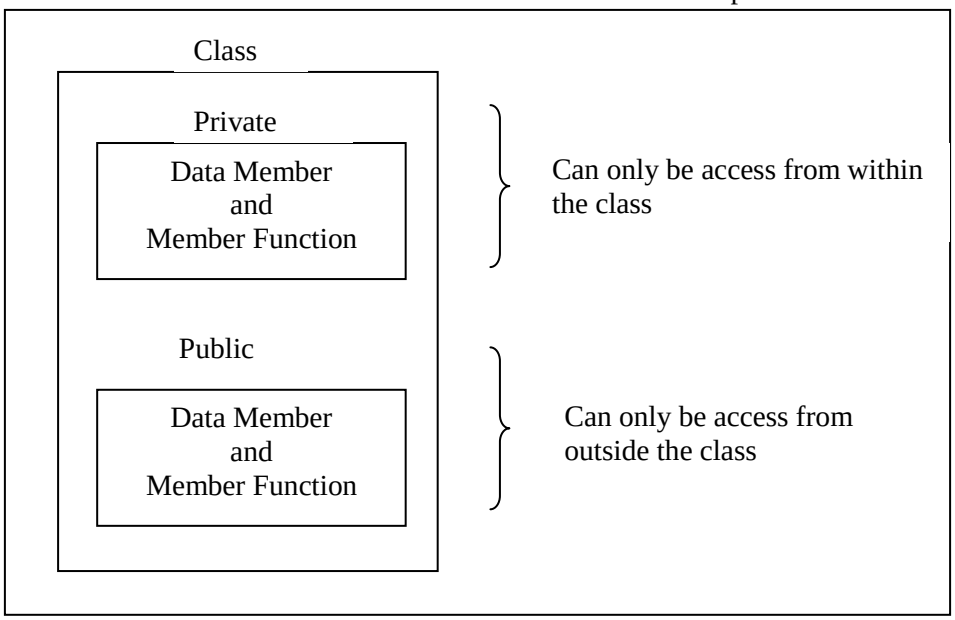
```
class class_name
{
        private:
                variables;
                functions;
        public:
                variables;
                functions;
        protected:
                variables;
                functions;
};
```

Where,

class, private, public, protected	= C++ keywords
class_name	= the name of class
variables	= variables (data) of the class
functions	= functions of the class

The variables and functions declared within the curly braces are collectively known as ***members*** of the class. The variables declared in the class are known as ***data members***, while the functions declared in the class are known as ***member function.***

The keywords ***private, public*** and ***protected*** are known as ***access specifiers***, also known as ***visibility mode.*** Each members of a class is associated with an access specifier.



❖ **Defining Member Functions**

Member functions of a class can be defined either outside the class definition or inside the class definition. In both cases, the function body remains the same; however, the function header is different.

Outside the class

Defining a member function outside a class requires the function declaration (function prototype) to be inside the class definition. The member function is declared inside the class like a normal function. this declaration informs the compiler that the function is a member of the class and that it has been defined outside the class.

The definition of member function outside the class differs from formal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (::). The scope resolution operator informs the compiler, to which class the member belongs.

Syntax,

```
return_type class_name :: function_name (parameter_list)
{
        // body of the member function
}
```

Inside the class

A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions. Thus, the member function that are small in size and frequently used are best suited to be defined inside the class.

Syntax,

```
return_type function_name (parameter_list)
{
        // body of the member function
}
```

Defining the object of a class

Once a class is defined, it can be used to create variables of its type. These variables are known as ***object.*** The relation between an object and a class is the same as that of a variable and its data type.

The Syntax,

Class_name object_list;

Where,

class_name = the name of the class

object_list = a comma separated list of object.

For example,

```
class book
{
        //body of the class
}
main()
{
        book b1, b2, b3;
        return 0;
}
```

❖ **Accessing member of a class**

The member of class can be directly accessed insider the class using their names. However, accessing a member outside the class depends on its access specifier. The access specifier not only determines the part of the program where the member is accessible, but also how it is accessible in the program.

Accessing public members

The public members of a class can be accessed outside the class directly using the object

name and dot operator ' . '. The dot operator associates a member with the specific object of the class.
Syntax,

Accessing a public data member

object_name . member_name

calling a public member function

object_name . function_name (parameter_list);

Accessing private member

The private members of a class are not accessible outside the class not even with the object name. however, they can be accessed indirectly through the public member function of that class.

❖ **Object and Arrays**

In addition to built in data types, a class can have array variables as its data members. A user can also create an array of objects which can have objects of a class as its individual elements.

Arrays of objects

Like array of other user defined data types, an array of type class can be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built in data type.
Syntax,

class_name array_name [size];

❖ **Pointer and classes**

Pointers can also be used with a class. In a class, pointer to the class members or to the objects of a class can be created. Moreover, a pointer to an object of a class can also point to an object of its derived class.

Pointers to class members

A class contains data as well as functions as its members and these members are accessed through the object of a class using the dot operator ' . '. In addition, the members of the class can be accessed with the help of pointers. A pointer that points to a class member is called a ***pointer to class member*** or simply a ***pointer-to-memer.***

C++ provides a set of three pointer operator to use pointers with the class members, which are listed

Operator	Description	Purpose
::*	Pointer-to-member declaratory	Declares a pointer to a class member.
.*	Pointer-to-member access operator	Accesses a member using object name and a pointer to that member.
->*	Pointer-to-member access operator	Accesses a member using pointer to object and a pointer to that member.

Pointer to data member

Before using a pointer to data member of the class, it needs to be declared.
The syntax,

*data_type class_naem ::*pointer_name;*

A pointer to data member should also be initialized with the address of the data member it points to.
The syntax,

pointer_name = &class_name :: data_name;

Pointer to member functions

A pointer to the member function of a class can also be declared and used. The member function pointers allow to call one of the set of an object's member functions indirectly. While specifying a member function pointer, the prototype of the function that it points to is first defined and then the pointer is initialized with the address of an appropriate member function of that class.
The syntax,

*return_type (class_name ::*pointer_name) (parameter_list); //declaration*
pointer_name = &class_name :: member_function; //initialization

Classes within classes (Nested Class)

A class that is defined inside another class is known as a ***nested class.*** A nested class (inner class) has a local scope inside the class that contains its definition (enclosing class).

Syntax,

```
class enclosing class_name
{
        .
        .
        class nested class_name
        {
                .
                .
        };
};
```

The syntax for defining a member function of a nested class outside the enclosing class is:

return_type enclosingclass_name :: nestedclass_name :: function_name (parameter_list);

Local Class

Classes that have been used earlier are defined outside the *main( )* function. Hence, these classes have global scope and can be accessed any where in the program. However, classes can also be defined inside a particular function. Such classes are known as ***local classes.*** These local classes are recognized only in the function in which they are defined and thus, cannot be accessed outside that function.

```
return_type function_name (parameter_list)
{
        class local_class
        {
                // body of the class
        };
                //body of the function
}
```

Limitation due to which use of local classes in C++

1) A local class cannot have static data member.
2) It cannot access local variables of a function in which it is defined.
3) An enclosing function can only access the public members of the class

Static Data Member

Each object of the class contains its own copy of data members. However, if a class data member is declared as static, only one copy of that data member is created, regardless of the number of objects. All the objects of a class share the single copy of the static data member.

To declare a static data member inside a class, the ***static*** keyword is prefixed to the data type of the data member.

Syntax,

```
static data_type member_name;
```

When the static data member is defined outside the class, the keyword ***static*** is not required.

Syntax,

```
data_type class_name :: member_name=value ;
```

Static Member Function

The member functions declared as static can access only other static data members and static member functions of the same class. static member functions are defined by prefixing the keyword ***static*** to their return type in the header of their definition inside the class.

Syntax,

```
static return_type function_name (parameter_list)
{
        //body of the member function
}
```

Special Member Function

In C++, memory is allocated to the data members of a class when an object of the class is declared. Once the object of class is declared, it need to be initialized with valid values before it is used in the program. To initialize the object of a class, a member function of the same class is to be defined that assigns valid values to the data members of the class.

Initialization of objects at the time of declaration is done with the help of a special member function of a class called ***constructor***. Similarly, when an object is no longer needed, the memory allocated to the object needs to be released. The de-allocation of the memory is done by another special member function called the ***destructor.***

Constructors

A constructor is a special member function of a class that initializes the objects of that class at the time of their declaration. It makes the object functional by converting an object with the unused memory into a usable object. It is special as it has the same name as that of the class and is automatically invoked whenever an object of the class is created.

Unlike other member functions, a constructor does not have any return type. This is because the constructor is invoked automatically by the system.

A constructor can be defined either inside or outside the class definition.

Syntax to define a constructor inside the class;

```
class class_name
{
        .
        .
Public:
        class_name(parameter_list)          //header of constructor
        {
                // body of the constructor
        }
};
```

Syntax, to define the constructor outside the class;

```
class class_name
{
        .
        .
Public:
        class_name(parameter_list)          // constructor prototype
        .
        .
};
class_name :: class_name (parameter_list)   //constructor definition
```

Calling Constructors

Once a constructor is defined, it can be called implicitly as well as explicitly. If the name of the constructor is not used in the object declaration, the call is known as an ***implicit call*** to the constructor. On the other hand, if the name of the constructor is used in the object declaration, the call is known as an ***explicit call*** to the constructor.

Types of Constructor

Constructor are classified into three types, namely, default constructor, parameterized constructor and copy constructor.

1) Default Constructor:

A default constructor has an empty parameter list and is used to initialize all the objects of a class with the same values. There can be only one default constructor in a class. The default constructor provided by the compiler initializes all the data member with garbage values.

2) Parameterized Constructor:

A parameterized constructor is a constructor that accept one or more parameters at the time of declaration of objects and initializes the data members of the objects with these parameters.

3) Copy Constructor:

The objects of a class can be initialized either with same values using default constructor or with different values using parameterized constructor. However, a new object of a class can also be initialized with an existing object of the same class. for this the compiler calls another constructor, known as the ***copy constructor***. A copy constructor initializes a new object of a class with the values of an existing object of the same class.

Destructors

Once an object is declared, memory space is allocated to the data members. During the execution of a program, an object may use other resources like files, etc. these resources and the memory allocated to object must be released with the object is destroyed. This is accomplished by another special member function called destructor that is automatically invoked to release all the resources and memory that an object acquires during the lifetime.

Like a constructor, a destructor is also special as it has the same name as that of the class of which it is a member but with a **tilde (~)** prefixed to its name. A tilde (~) is a C++ complement operator, which reminds that a destructor is a complement of the constructor.

Inline Function

Whenever a function is invoked, a set of operations is performed which includes passing the control from the calling function to the called function, managing stack for arguments and return values, managing registers, etc. all these operations take much of compiler time and slow down the execution process.

C++ provides a better way to make function calls execute faster and also perform tyhpe checking, which is making the function inline.

An inline function is a function whose code is copied in place of each function call. Inline functions can be declared by prefixing the keyword ***inline*** to the return type in the function prototype. An inline function 'request' the compiler to replace its each and every call by the code in its body.

The syntax for inline function,

inline return_type function_name (parameter_list);

Friend Function

A non member function that is made a 'friend' of a class is known as a ***friend function.*** A friend function has access to all the private and public members of a class of which it is a 'friend'. Friend functions are declared with the *friend* keyword inside the class. However, the keyword friend is not specified while defining the friend function. moreover, a friend function has to be defined outside the class definition as an ordinary function.

Syntax,

```
class class_name
{
        friend return_type function_name (Parameter_list);
};
```

Even though a friend function is declared inside a class, it is not a member of the class. As a result, the access specifiers of a class do not apply to friend functions and hence, can be directly accessed outside the class.

Dynamic Memory Allocation

In C++, the pointer support dynamic memory allocation (allocation of memory during runtime).

While studying arrays we declared the array size approximately. In this case if the array is less than the amount of data we cannot increase it at runtime. So, if we wish to allocate memory as and when required **new** operator helps in this context.

The syntax of the new operator is given below :

pinter_variable = **new** data_type;

Where the data type is any allowed C++ data type and the pointer_variable is a pointer of the same data type. For example,

```
char * cptr
cptr = new char;
```

The above statements allocate 1 byte and assigns the address to cptr.

The following statement allocates 21 bytes of memory and assigns the starting address to cptr :

```
char * cptr;
cptro = new char [21];
```

We can also allocate and initialize the memory in the following way :

```
Pointer_variable = new data_type (value);
```

Where value is the value to be stored in the newly allocated memory space and it must also be of the type of specified data_type. For example,

```
Char *cptr = new char ('j');
Int *empno = new int [size]; //size must be specified
```

Delete Operator

It is used to release or deallocate memory. The syntax of **delete** operator is :

```
delete_pointer_variable;
```

For example,

```
delete cptr;
delete [ ] empno; //some versions of C++ may require size
```

This Pointer

The this pointer is a special pointer that contains the address of an object of a class currently calling the member function of the class. whenever, an object of the class calls any of its non static member function, the address of the object is passed as an implicit argument to the function and is available as a local variable within the body of function.

this pointer can be used to return the object pointed to by it, to the calling program using this statement.

Syntax,

return (*this);

Single and Multiple Inheritance

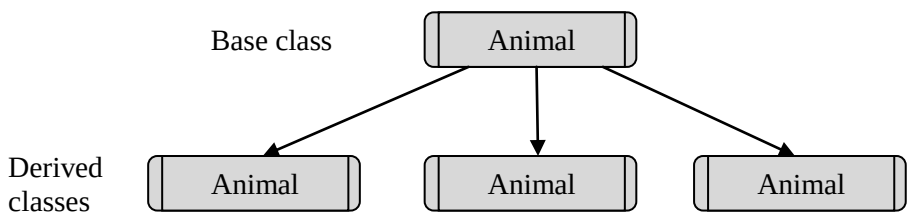
The logical relationship between the classes is achieved through inheritance. Inheritance, is one of the major strengths of an object oriented programming.

Inheritance is a mechanism of deriving a new class from the old class in such a way that the new class inherits all the member of the old class. in other words, inheritance facilitates a class to acquire the properties and functionality of another class. the new class depicts the acquired properties and behavior of the existing class as well as its own unique properties and behavior.

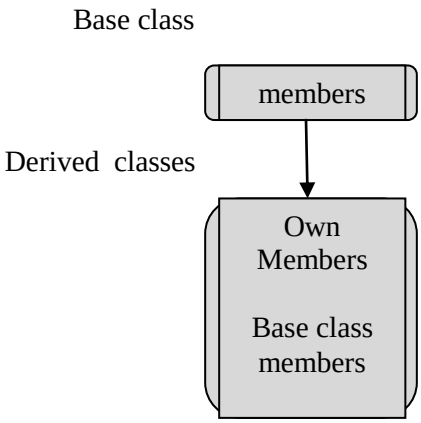
Base class and derived class

In inheritance the class that is inherited by the new class is called ***base class*** or ***superclass or parent.*** The class that inherits the members of the old class is called ***derived class or subclass or child class.***

For example,



In addition to the base class members the derived class also contains its own members. It implies the derived class extends the base class.

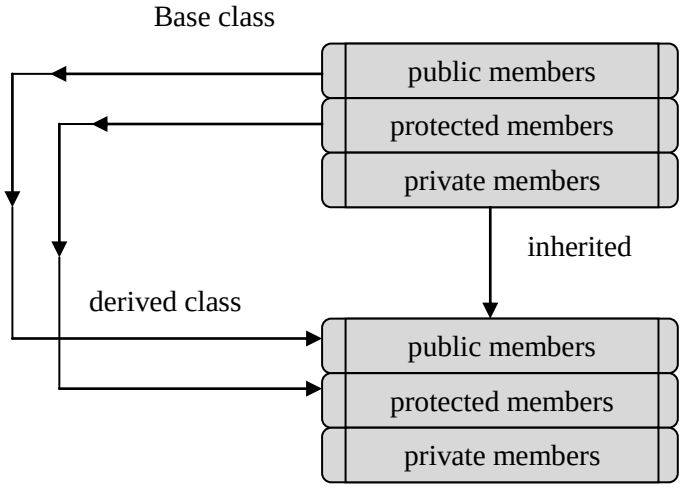


Access Specifier of the base class (Types)

The access specifier of the base class in the derived class definition determines the way the derived class inherits the base class. it determines the access specifier of the member of the base class inside the derived class. depending on the access specifies public, protected, or private, a base class can be *publicly inherited, protectedly inherited or privately inherited.*

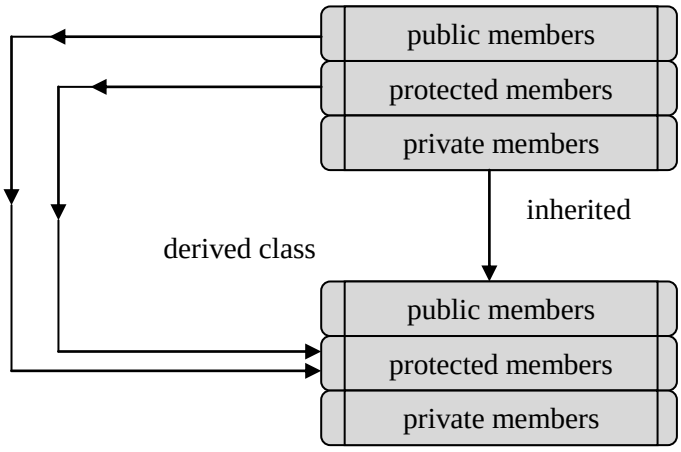
1) Publicly Inherited Base Class

When the access specifier of the base class in the derived class definition is Public, the base class is publicly inherited. The access specifier of the base class members remains the same in the derived class. That is, the public members of the base class remain the same in the derived class and so on.



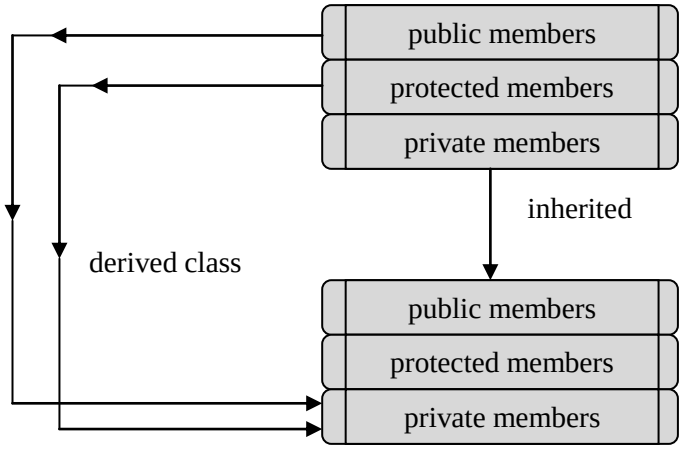
Protectedly inherited base class

When the access specifier of the base class in the derived class definition is protected, the base class is protectedly inherited. Both the public and the protected members of the base class become the protected members of the derived class,



Privately Inherited Base class

When the access specifier of the base class in the derived class definition is private the base class is privately inherited. In this case, both the public and the protected members of the base class become the private members of the derived class.

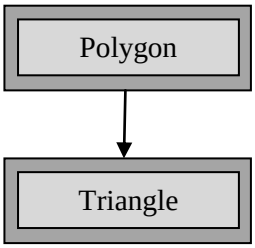


Types of Inheritance

Depending on the number of classes involved and the way the classes are inherited, inheritance are of several forms, namely, single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.

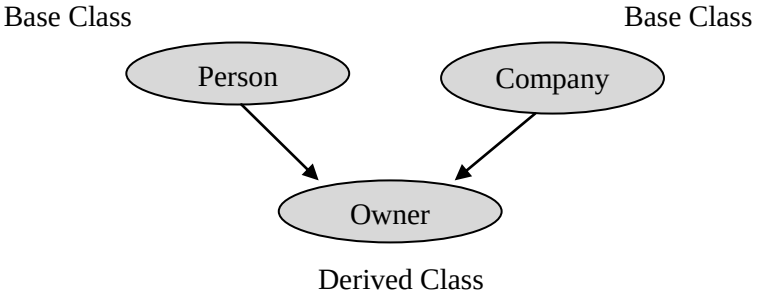
1) Single Inheritance:

When a derived class inherits from a single base class, it is referred to as single inheritance. In single inheritance, the derived class inherits all the members of its base class and can directly access the public and protected members of the base class.



2) Multiple Inheritance:

When a derived class inherits from more than one base class simultaneously, it is referred to as multiple inheritance. In multiple inheritance, the derived class inherits the members of all its base classes and can directly access the public and protected members of its base classes.



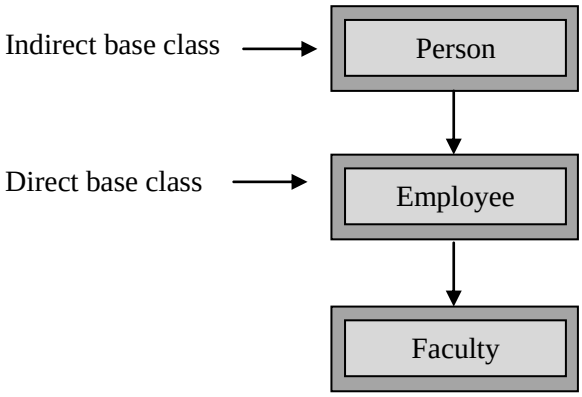
Syntax for define the derived class

Class derived_class: access_specifier1 base_class1,
access_specifier2 base_class2, ... access_specifiern base_classn
{
.
.
};

3) Multilevel Inheritance

When one class is inherited from another class, which in turn is inherited from some other class, it is referred to as multilevel inheritance. Multilevel inheritance comprises two or more levels.

The derived class is inherited from the direct base class, which in turn is derived from another base class. hence, the derived class has all the members of its direct base class as well as its indirect base class. This is known as the Transitive nature of multilevel of inheritance.

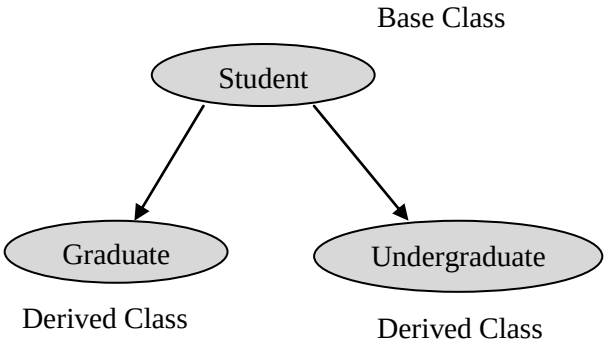


The syntax to implement multilevel inheritance is:

```
class base_class1
        {
                // members of base_class 1
        };
class base_class2: access_specifier1 base_class1
        {
                // members of base_class 2
        };
class derived_class: access_specifier2 base_class 2
        {
                // members of derived_class
        };
```

4) Hierarchical Inheritance:

Hierarchical inheritance is a type of inheritance in which more than one class is derived from a single base class. in hierarchical inheritance, a base class provide members that are common to all of its derived classes.



In hierarchical inheritance, each of the derived class inherits all the members of its base class. In addition, all the derived classes can directly access the public and protected members of the base class. However, one derived class cannot access the members of another derived class.

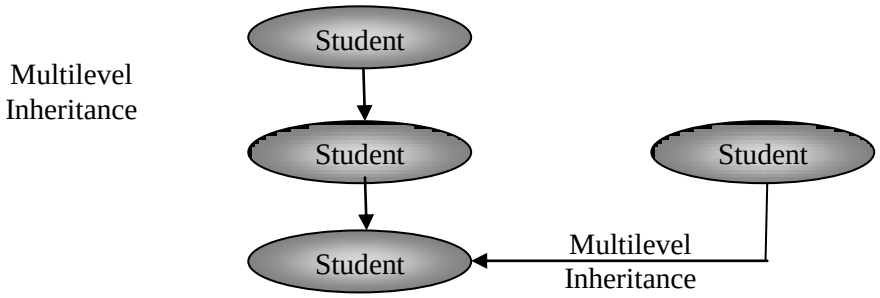
The syntax,

```
class base_class
        {
                //members of base_class
        };
class derived_class1 : access_specifier1 base_class
        {
```

```
        // member of derived_class
    };
class derived_class2 : access_specifier2 base_class
    {
        // members of derived_class
    };
```

5) Hybrid Inheritance:

According to the user requirement the various types of inheritance, can be combined in a program. Such inheritance is called hybrid inheritance.



Container classes

A container stores many entities and provide sequential or direct access to them. List, vector and strings are such containers in standard template library. The string class is a container that holds chars. All container classes access the contained elements safely and efficiently by using iterators. Container class is a class that hold group of same or mixed objects in memory. It can be heterogeneous and homogeneous. Heterogeneous container class can hold mixed objects in memory whereas when it is holding same objects, it is called as homogeneous container class.

The following are the standardized container classes :

std::map :

Used for handle sparse array or a sparse matrix.

std::vector :

Like an array, this standard container class offers additional features such as bunds checking through the at () member function, inserting or removing elements, automatic memory management and throwing exceptions.

std::string :

A better supplement for arrays of chars.

Difference between Container and Inheritance

Container	Inheritance
1) The object of one class is used as member in another class	1) The object of one class inherits the property of another class
2) Does not support the concept of reusability	2) Support the concept of reusability.
3) Does not provide additional features to existing class.	3) Provides additional features to an existing class.
4) Represents 'has-a' relationship	4) Represents 'is-a-kind-of' relationship.