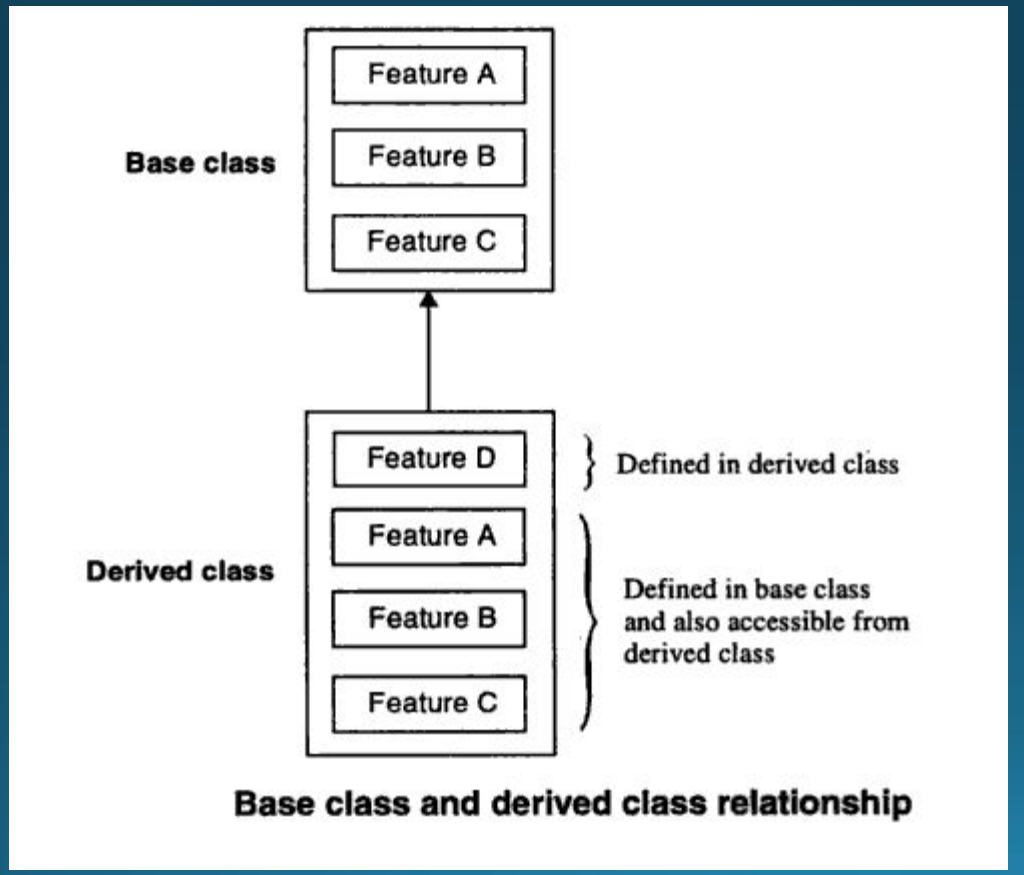
Inheritance

Inheritance

Reusability is yet another feature of OOP. C++ strongly supports the concept of reusability using Inheritance.

Inheritance is a technique of organizing information in a hierarchical form.

Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Classes are created by first inheriting all the variables and behavior defined by some primitive class and then adding specialized variables and behaviors. In object oriented programming, classes encapsulate data and functions into one package. New classes can be built from existing ones. *The technique of building new classes from the existing classes is called inheritance.*



Base class and derived class relationship

Inheritance, a prime feature of OOPs can be stated as *the process of creating new classes (called derived classes), from the existing classes (called base classes)*. The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own. The base class remains unchanged.
derived class inherits the features of the base class (A, B, and C) and adds its own features (D). The arrow in the diagram symbolizes *derived from*. Its direction from the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.

```
class ClassName
{
      private:
            ....      // visible to member functions within
            ....      // its class but not in derived class
      protected:
            ....      // visible to member functions within
            ....      // its class and derived class
      public:
            ....      // visible to member functions within
            ....      // its class, derived classes and through object
};
```

```
class X
{
   private:
      int a;
      void f1()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
   protected:
      int b;
      void f2()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
   public:
      int c;
      void f3()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
};
```

The following statements,

```
X objx;                    // objx is an object of class X
int d;              // temporary variable d
```

define the object `objx` of the class `X` and the integer variable `d`. The member access privileges are illustrated by the following statements referring to the object `objx`.

1. Accessing private members of the class X

```
d = objx.a; // Error: 'X::a' is not accessible
objx.f1();  // Error: 'X::f1()' is not accessible
```

Both the statements are invalid because the private members of a class are inaccessible to the object `objx`.

2. Accessing protected members of the class X

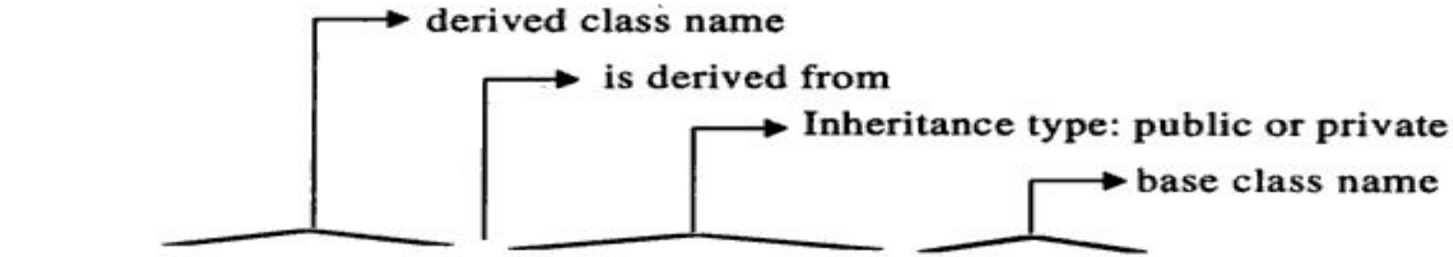
```
d = objx.b; // Error: 'X::b' is not accessible
objx.f2();  // Error: 'X::f2()' is not accessible
```

Both the statements are invalid because the protected members of a class are inaccessible since they are private to the class X.

3. Accessing public members of the class X

```
d = objx.c; // OK
objx.f3();  // OK
```

Both the statements are valid because the public members of a class are accessible to statements outside the scope of the class.

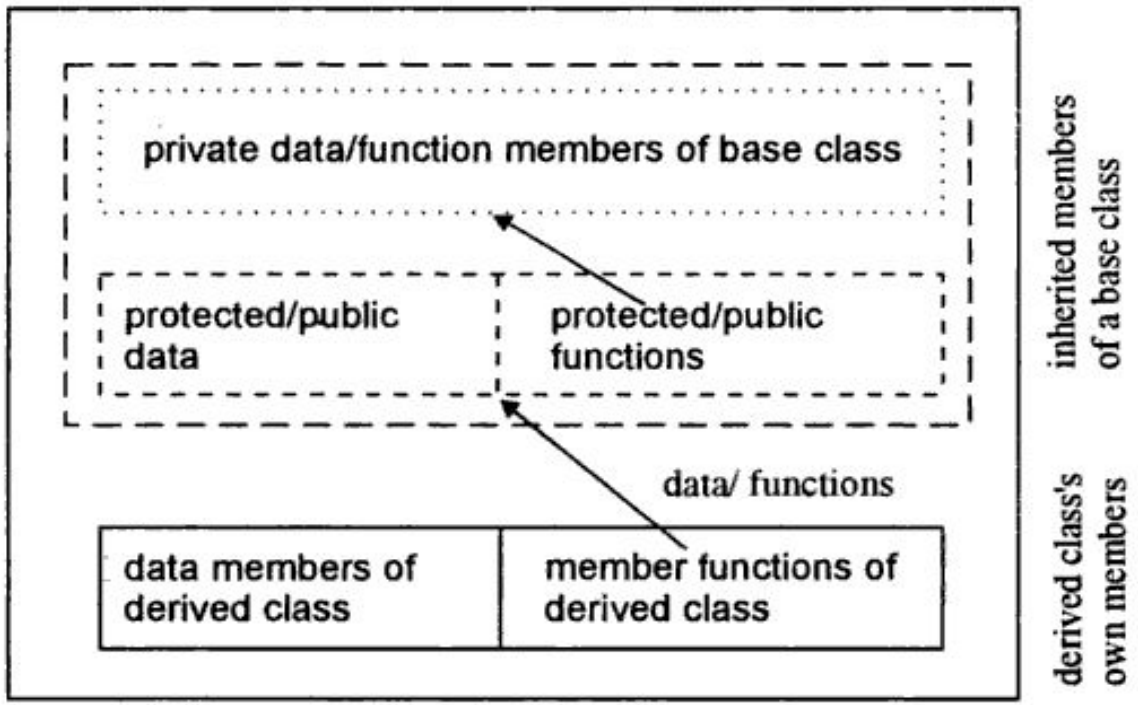


Syntax of derived class declaration

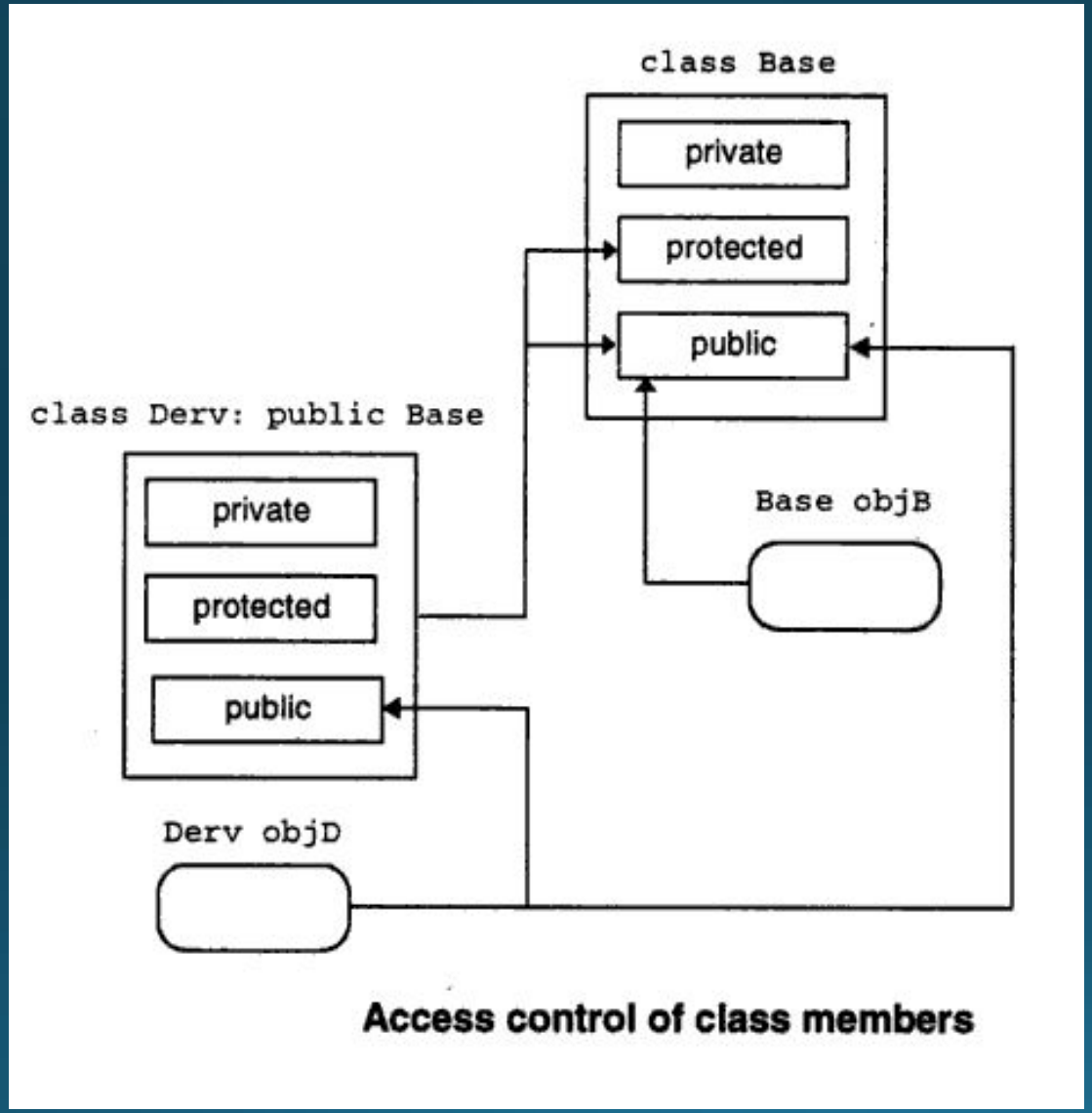
The derivation of `DerivedClass` from the `BaseClass` is indicated by the colon (:). The `VisibilityMode` enclosed within the square brackets implies that it is optional. The default visibility mode is `private`. If the visibility mode is specified, it must be either `public` or `private`. Visibility mode specifies whether the features of the base class are *publicly* or *privately inherited*.

The following are the three possible styles of derivation:

```
1.   class D: public B   // public derivation
     {
          // members of D
     };
2.   class D: private B  // private derivation
     {
          // members of D
     };
3.   class D: B          // private derivation by default
     {
          // members of D
     };
```



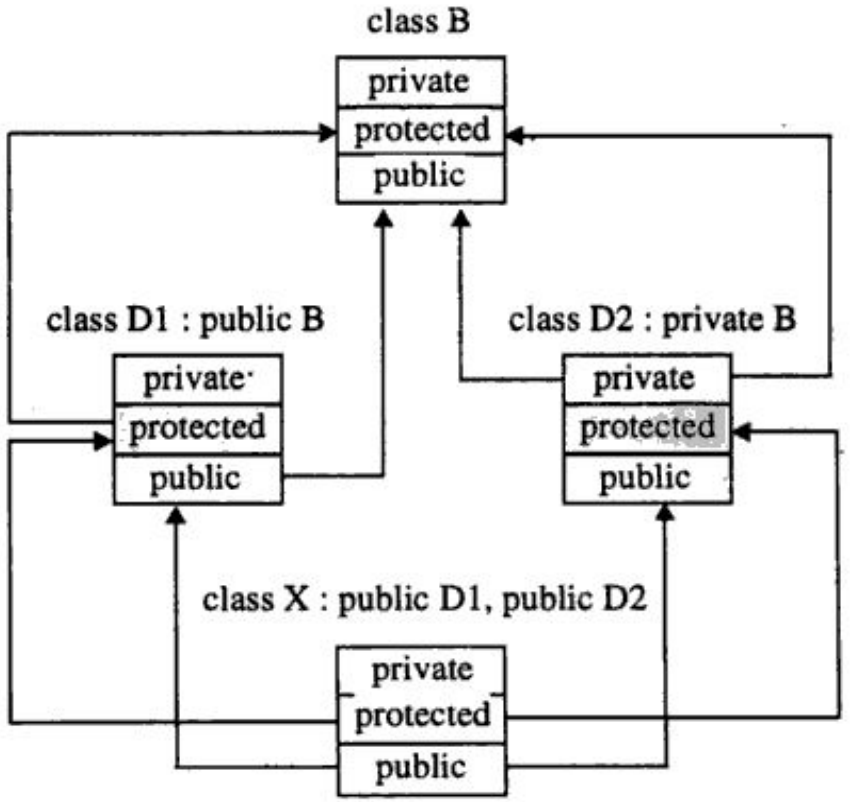
Members of derived class on inheritance



Access control of class members

Function Type	Access directly to		
	Private	Protected	Public
Class Member	Yes	Yes	Yes
Derived class member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend class member	Yes	Yes	Yes

Access control to class members



Access mechanism in classes

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
private	Not Inherited (inherited base class members can access)	Not Inherited (inherited base class members can access)
protected	protected	private
public	public	private

Visibility of class members

Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

Single Inheritance: Derivation of a class from only one base class is called single inheritance.

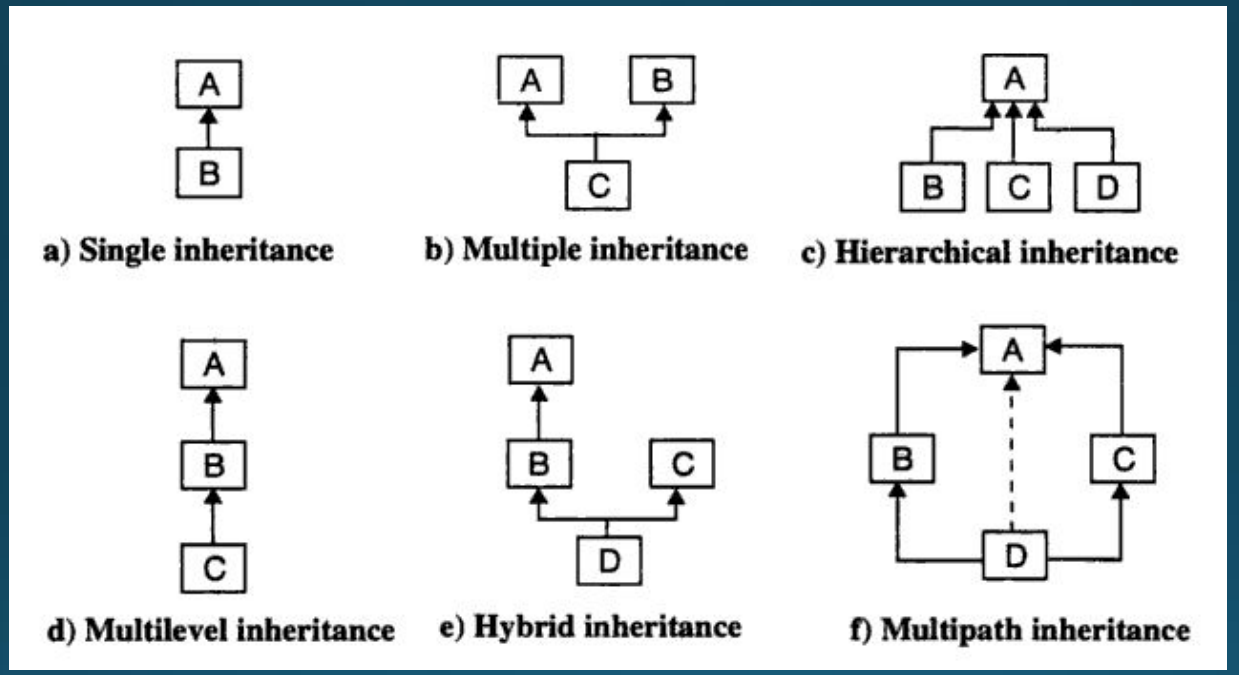
Multiple Inheritance: Derivation of a class from several (two or more) base classes is called multiple inheritance.

Hierarchical Inheritance: Derivation of several classes from a single base class i.e., the traits of one class may be inherited by more than one class, is called hierarchical inheritance.

Multilevel Inheritance: Derivation of a class from another *derived class* is called multilevel inheritance.

Hybrid Inheritance: Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

Multipath Inheritance: Derivation of a class from other *derived classes*, which are derived from the same base class is called multipath inheritance.


A
B
a) Single inheritance
A
B
C
b) Multiple inheritance
A
B
C
D
c) Hierarchical inheritance
A
B
C
d) Multilevel inheritance
A
B
C
D
e) Hybrid inheritance
A
B
C
D
f) Multipath inheritance

Inheritance and Member Accessibility

1. A private member is accessible only to members of the class in which the private member is 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 class and to any of the members in a derived class.
4. If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather than private.
5. A public member is accessible to members of its own class, members of the derived class, and outside users of the class.
6. The private, protected, and public sections may appear as many times as needed in a class and in any order. In case an inline member function refers to another member (data or function), that member must be declared before the inline member function is defined. Nevertheless, it is a normal practice to place the private section first, followed by the protected section and finally the public section.
7. The visibility mode in the derivation of a new class can be either private or public.
8. Constructors of the base class and the derived class are automatically invoked when the derived class is instantiated. If a base class has constructors with arguments, then their invocations must be explicitly specified in the derived class's initialization section. However, no-argument constructor need not be invoked explicitly. Remember that, constructors must be defined in the public section of a class (base and derived) otherwise, the compiler generates the error message: *unable to access constructor*.

Constructors in Derived Classes

The constructors play an important role in initializing an object's data members and allocating required resources such as memory. The derived class need not have a constructor as long as the base class has a no-argument constructor. However, if the base class has constructors with arguments (one or more), then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructor. In the application of inheritance, objects of the derived class are usually created instead of the base class. Hence, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the 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.

The following examples illustrate the order of invocation of constructors in the base class and the derived class.

1. No-constructors in the base class and derived class

When there are no constructors either in the base or derived classes, the compiler automatically creates objects of classes without any error when the class is instantiated.

```
// cons1.cpp: No-constructors in base class and derived class
#include <iostream.h>
class B      // base class
{
   // body of base class, without constructors
};
class D: public B    // publicly derived class
{
   // body of derived base class, without constructors
   public:
   void msg()
   {
      cout << "No constructors exists in base and derived class" << endl;
   }
};
void main()
{
   D objd; // base constructor
   objd.msg();
}
```

Run

```
No constructors exists in base and derived class
```

2. Constructor only in the base class

```
// cons2.cpp: constructor in base class only
#include <iostream.h>
class B     // base class
{
   public:
   B()
   {
     cout << "No-argument constructor of the base class B is executed";
   }
};
class D: public B    // publicly derived class
{
   public:
};
void main()
{
   D obj1; // accesses base constructor
}
```

Run

```
No-argument constructor of the base class B is executed
```

3. Constructor only in the derived class

```
// cons3.cpp: constructors in derived class only
#include <iostream.h>
class B      // base class
{
   // body of base class, without constructors
};
class D: public B    // publicly derived class
{
   // body of derived base class, without constructors
   public:
   D()
   {
      cout << "Constructos exists in only in derived class" << endl;
   }
};
void main()
{
   D objd;           // accesses derived constructor
}
```

Run

```
Constructos exists in only in derived class
```

4. Constructor in both base and derived classes

```
// cons4.cpp: constructor in base and derived classes
#include <iostream.h>
class B      // base class
{
  public:
  B()
  {
   cout<<"No-argument constructor of the base class B executed first\n";
  }
};
class D: public B    // publicly derived class
{
 public:
  D()
  {
   cout<<"No-argument constructor of the derived class D executed next";
  }
};
void main()
{
   D objd;  // access both base constructor
}
```

Run

```
No-argument constructor of the base class B executed first
No-argument constructor of the derived class D executed next
```

5. Multiple constructors in base class and a single constructor in derived class

```
// cons5.cpp: multiple constructors in base and single in derived classes
#include <iostream.h>
class B     // base class
{
   public:
      B() { cout << "No-argument constructor of the base class B"; }
      B(int a) { cout <<"One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
   public:
      D( int a )
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

Run

```
No-argument constructor of the base class B
One-argument constructor of the derived class D
```

6. Constructor in base and derived classes without default constructor

The compiler looks for the no-argument constructor by default in the base class. If there is a constructor in the base class, the following conditions must be met:

- The base class must have a no-argument constructor
- If the base class does not have a default constructor and has an argument constructor, they must be explicitly invoked, otherwise the compiler generates an error.

```
// cons6.cpp: constructor in base and derived class
#include <iostream.h>
class B     // base class
{
   public:
      B(int a) { cout << "One-argument constructor of the base class B";  }
};
class D: public B    // publicly derived class
{
   public:
      D( int a )
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

The compilation of the above program generates the following error:

```
Cannot find 'default' constructor to initialize base class 'B'
```

This error can be overcome by explicit invocation of a constructor of the base class as illustrated in the program `cons7.cpp`.

7. Explicit invocation in the absence of default constructor

```
// cons7.cpp: constructor in base and derived classes
#include <iostream.h>
class B      // base class
{
   public:
      B(int a)
      { cout << "One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
   public:
      D( int a ) : B(a)
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

Run

```
One-argument constructor of the base class B
One-argument constructor of the derived class D
```

In the derived class D, the statement

```
D( int a ):B(a)
```

defines the derived class constructor D(int a) and calls the constructor of the base class using the special form :B(a). Here, the constructor of B is first invoked with an argument a specified in the constructor function D and then the constructor of D is invoked.

8. Constructor in a multiple inherited class with default invocation

```
// cons8.cpp: constructor in base and derived class, order of invocation
#include <iostream.h>
class B1     // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2     // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B2, public B1    // publicly derived class
{
   public:
      D()
      { cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B2, public B1    // publicly derived class
```

specifies that the class `D` is derived from the base classes `B1` and `B2` in order. Hence, constructors are invoked in the order `B2()`, `B1()`, and `D()`; the constructors can be defined with or without arguments.

9. Constructor in a multiple inherited class with explicit invocation

```
// cons9.cpp: constructors with explicit invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() {  cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
      B2() {  cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B1, public B2
{
   public:
      D(): B2(), B1()    // explicit call to constructors
      {  cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
```

In the above program, the statement

```
class D: public B1, public B2    // publicly derived class
```

specifies that, the class `D` is derived from the base classes `B1` and `B2` in order. The statement

```
D(): B2(), B1()
```

in the derived class `D`, specifies that, the base class constructors must be called. However, the constructors are invoked in the order `B1()`, `B2`, and `D`, the order in which the base classes appear in the declaration of the derived class.

10. Constructor in base and derived classes in multiple inheritance

```
// cons10.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of a base class B2"; }
};
class D: public B1, virtual B2      // public B1, private virtual B2
{
   public:
      D(): B1(), B2()
      { cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;  // base constructor
}
```

Run

```
No-argument constructor of a base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B1, virtual B2      // public B1, private virtual B2
```

specifies that the class `D` is derived from the base classes `B1` and `B2`. The statement

```
D():B1(), B2()
```

in the derived class `D`, specifies that, the base class constructors must be called. However, the constructors are invoked in the order `B2()`, `B1`, and `D()`, instead of the order in which base classes appear in the declaration of the derived class, since, the virtual base class constructors are invoked first followed by an orderly invocation of constructors of other classes.

11. Constructor in multilevel inheritance

```
// cons11.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B     // base class
{
   public:
      B() { cout << "\nNo-argument constructor of a base class B"; }
};
class D1: public B     // derived class
{
   public:
      D1() { cout << "\nNo-argument constructor of a base class D1"; }
};
class D2: public D1    // publicly derived class
{
   public:
      D2()
      { cout << "\nNo-argument constructor of a derived class D2"; }
};
void main()
{
   D2 objd;  // base constructor
};
```

Run

```
No-argument constructor of a base class B
No-argument constructor of a base class D1
No-argument constructor of a derived class D2
```

The statement

```
class D2: public D1    // publicly derived class
```

specifies that the class `D2` is derived from the derived class `D1` of `B`. The constructors are invoked in the order `B()`, `D1()`, and `D2()` corresponding to the order of inheritance.

In the derived class, first the constructors of virtual base classes are invoked, second any non-virtual classes, and finally the derived class constructor. Table 14.3 shows the order of invocation of constructors in a derived class.

Method of Inheritance	Order of Execution

`class D: public B`
`{`
`...`
`};`
	B( ): base constructor
	D( ): derived constructor

`class D: public B1, public B2`
`{`
`...`
`};`
	B1( ): base constructor
	B2( ): base constructor
	D( ): derived constructor

`class D: public B1, virtual B2`
`{`
`..`
`};`
	B2( ): virtual base constructor
	B1( ): base constructor
	D( ): derived constructor

`class D1: public B`
`{`
`...`
`};`
`class D2: public D1`
`{`
`..`
`};`
	B( ): super base constructor
	D1( ): base constructor
	D2( ): derived constructor

Order of invocation of constructors

Destructors in Derived Classes

Unlike constructors, destructors in the class hierarchy (parent and child class) are invoked in the reverse order of the constructor invocation. The destructor of that class whose constructor was executed last, while building object of the derived class, will be executed first whenever the object goes out of scope. If destructors are missing in any class in the hierarchy of classes, that class's destructor is not invoked.

```
// cons12.cpp: order of invocation of constructors and destructors
#include <iostream.h>
class B1     // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
       ~B1()
       {
          cout << "\nDestructor in the base class B1";
       }
};
class B2     // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
       ~B2()
       {
          cout << "\nDestructor in the base class B2";
       }
};
class D: public B1, public B2    // publicly derived class
{
   public:
       D()
       { cout << '\nNo-argument constructor of the derived class D"; }
       ~D()
       {
          cout << "\nDestructor in the base class D";
       }
};
void main()
{
   D objd;
}
```

Run

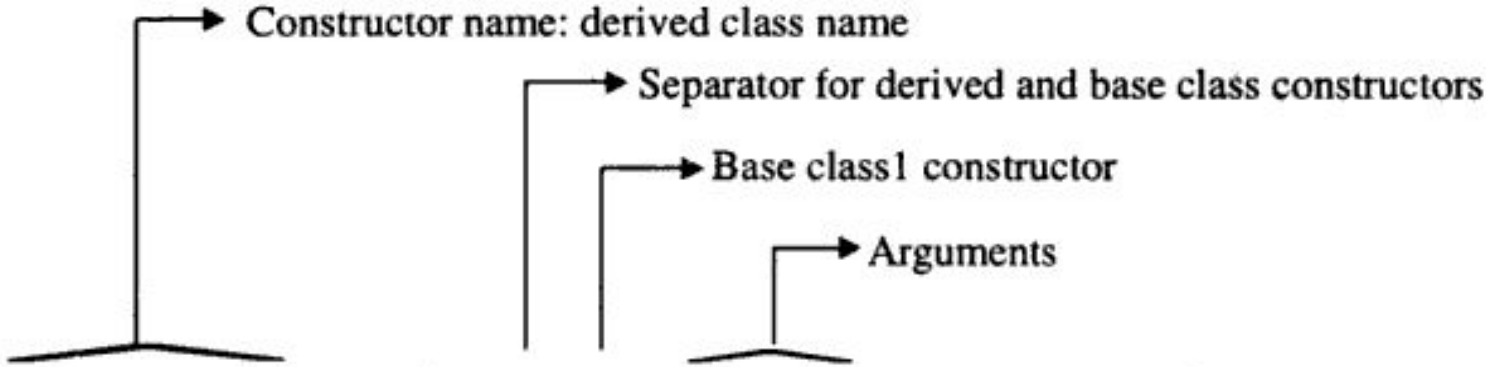
```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
Destructor in the base class D
Destructor in the base class B2
Destructor in the base class B1
```

Note that, in this program the constructors are invoked in the order of `B1()`, `B2()`, `D()` whereas, the destructors are invoked in the order of `D()`, `B2()`, `B1()`, which is in reverse order.

In case of dynamically created objects using the `new` operator, they must be destroyed explicitly by invoking the `delete` operator. More specialized class's (which are at the bottom of the hierarchy) destructors are called before a more general one (which are at the top of the hierarchy). As usual, no arguments can be passed to destructors, nor can any return type be declared.

Constructors Invocation and Data Members Initialization

In multiple inheritance, the constructors of base classes are invoked first, *in the order in which they appear in the declaration of the derived class*, whereas in the case of multilevel inheritance, they are executed *in the order of inheritance*. It is the responsibility of the derived class to supply initial values to the base class constructor, when the derived class objects are created. Initial values can be supplied either by the object of a derived class or a constant value can be mentioned in the definition of the constructor. The syntax for defining a constructor in a derived class is shown

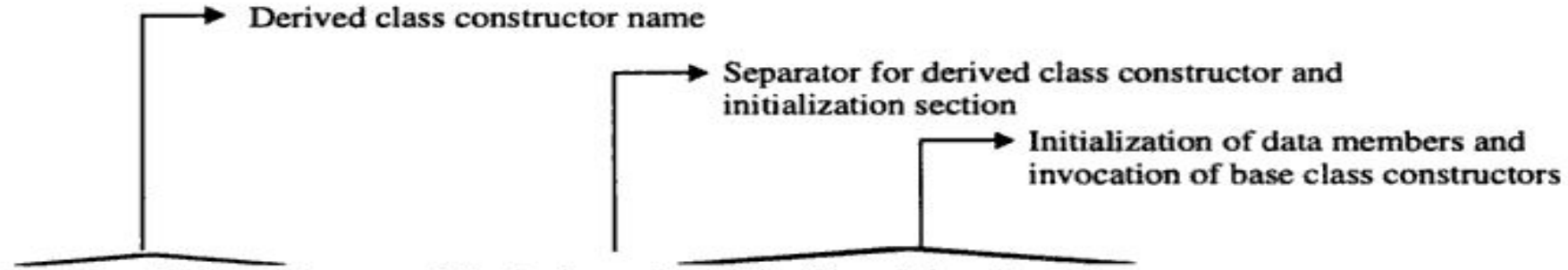



```
DerivedClass(arg_list):Base1(arg_list1),Base2(arg_list2)..,BaseN(arg_listM)
{
   // body of the constructor of derived class
};
```

Syntax of derived class constructor

The parameters arg_list1, arg_list2, .., arg_listM are the list of arguments passed to the constructor or they can be any constant value those match with the arguments of the *constructor list* of base classes.

C++ supports another method of initializing the objects of classes through the use of the *initialization list* in the constructor function. It facilitates the initialization of data members by specifying them in the header section of the constructor. The general form of this method is shown



Syntax of initialization at derived class constructor

Data member initialization is represented by

```
DataMemberName( value )
```

The data members (DataMemberName) to be initialized are followed by the initialization value enclosed in parentheses (resembles a function call). The `value` can be arguments of a constructor, expression or other data members. In the initialization section, any parameter of the argument-list can be used as an initialization value. The data member to be initialized must be a member of its own class. The program `cons14.cpp` illustrates the use of initialization section of the constructor. The following rules must be noted about the initialization and order of invocation of constructors:

- The initialization statements (in the initialization section) are executed in the order of definition of data members in the class.
- Constructors are invoked in the order of inheritance. However, the following rules apply when class is instantiated: first, the constructors of virtual base classes are invoked, second, any non-virtual classes, and finally, the derived class constructor.

```
// cons13.cpp: data members initialization through initialization-section
#include <iostream.h>
class B      // base class
{
   protected:
      int x, y;
   public:
      B(int a, int b): x(a), y(b) {} // x = a, y = b
};
class D: public B    // derived class
{
   private:
      int a, b;
   public:
      D(int p, int q, int r): a(p), B( p, q ), b(r) {}
   void output()
   {
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << "a = " << a << endl;
      cout << "b = " << b << endl;
   }
};
void main()
{
   D objb(5, 10, 15);
   objb.output();
}
```

Run

```
x = 5
y = 10
a = 5
b = 15
```

Single Inheritance

When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
int age;
char name [10];
public:
void getworker ( );
void showworker( );
};
void worker : : getworker( )
{
cout <<"yout name please"
cin >> name;
cout <<"your age please" ;
cin >> age;
}
void worker :: showworker( )
{
cout <<"\n My name is :"<<name<<"\n My age is :"<<age;
}


class manager . public worker //derived class (publicly)
{
int now;
public:
void getnow( ) ;
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
cout << "number of workers under you";
cin >> now;
}
void manager :: shownow( )
{
cout <<"\n No. of workers under me are: " << now;
}


main ( )
{
clrscr ( ) ;
manager M1;
M1.getworker( ); M1.getnow( );
M1.showworker( ); M1.shownow( );
}
```

RUN OUTPUT:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker                                    //Base class declaration
{
int age;
char name [10] ;
public:
void getworker( ) ;
void showworker( ) ;
};
void worker : : gctworker( )
{
cout << “your name please” ;
cin >> name;
cout << “your age please”;
cin >>age;
}
void worker : showworker( )
{
cout << “\n my name is: “ <<name<< “\n” << “my age is : “ <<age;
}
class manager : private worker                  //Derived class (privately by default)
{
int now;
public:
void gctnow( ) ;
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
getworker( );       //calling the get function of base
cout << "number of worker under you";
cin >>now;
}
void manager : : shownow( )
{
showworker( ) ;
cout << "in no. of worker under me are : " <<now;
}
main ( )
{
clrscr ( ) ;
manager ml;
ml.getnow( ) ;
ml.shownow( );
}
```

The following program shows the single inheritance using protected derivation

```
#include<conio.h>
#include<iostream.h>
class worker                                    //Base class declaration
{ protected:
int age; char name [20];
public:
void getworker( );
void showworker( );
};
void worker :: getworker( )
{
cout >> "your name please";
cin >> name;
cout << "your age please";
cin >> age;
}
void worker :: showworker( )
{
cout << "in my name is: " << name << "in my age is " <<age;
}
class manager:: protected worker                    // protected inheritance
{
int now;
public:
void getnow( );
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
cout << “please enter the name In”;
cin >> name;
cout<< “please enter the age In”; //Directly inputting the data
cin >> age; members of base class
cout << “ please enter the no. of workers under you:”;
cin >> now;
}
void manager : : shownow( )
{
cout « "your name is : "«name«" and age is : "«age;
cout «“No. of workers under your are : "«now;

main ( )
{
clrscr ( ) ;
manager ml;
ml.getnow( ) ;
cout « "\n \n";
ml.shownow( );
}
```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

a. **Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited

Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

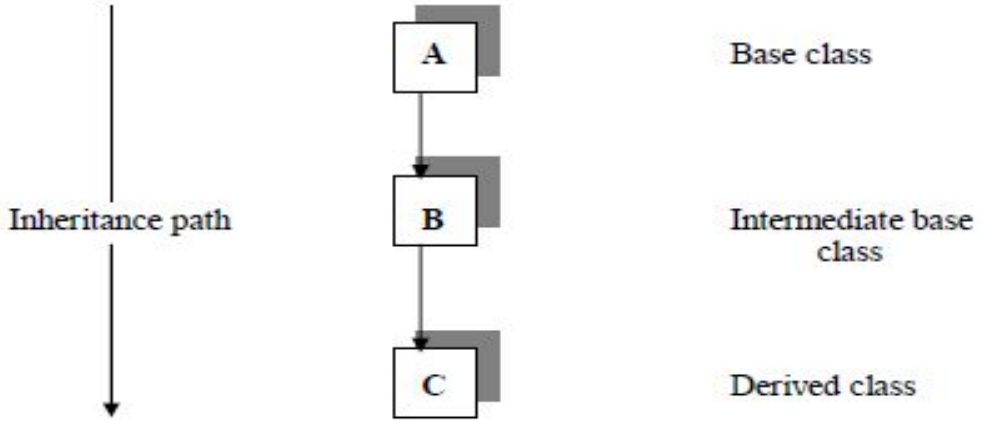
a. A function i.e. friend of a class.

b. A member function of a class that is the friend of the class.

c. A member function of a derived class.

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g.



The declaration for the same would be:

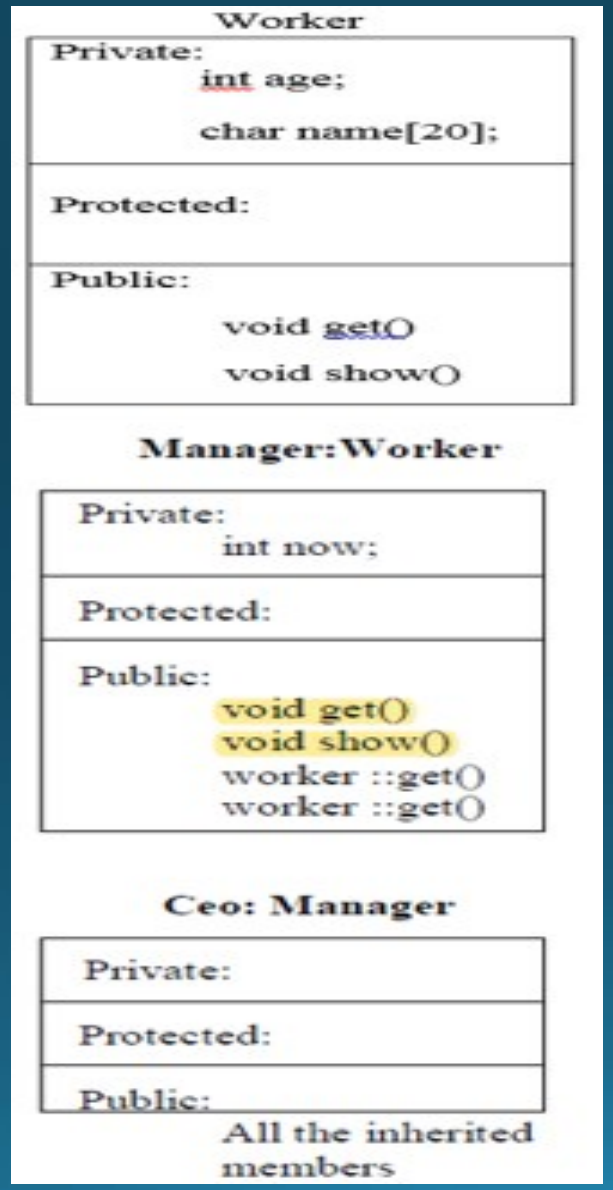
```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.
Following program exhibits the multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
class worker                                        // Base class declaration    worker
{
int age;
char name [20] ;
public;
void get( ) ;
void show( ) ;
}
void worker: get ( )
{
cout << “your name please” ;
cin >> name;
cout << “your age please” ;
}
void worker : : show ( )
{
cout << “In my name is : “ <<name<< “ In my age is : “ <<age;
}
class manager : public worker                       //Intermediate base class derived    manager
{ //publicly from the base class
int now;
public:
void get ( ) ;
void show( ) ;
};
```

```
void manager :: get ( )
{
worker : :get () ; //calling get ( ) fn. of base class
cout << "no. of workers under you:";
cin >> now;
}
void manager : : show ( )
{
worker : : show ( ) ; //calling show ( ) fn. of base class
cout << "In no. of workers under me are: "<< now;
}
class ceo: public manager                    //publicly inherited from the intermediate base class  ceo
{
int nom;
public:
void get ( ) ;
void show ( ) ;
};
void ceo : : get ( )
{
manager : : get ( ) ;
cout << "no. of managers under you are:"; cin >> nom;
}
 void manager : : show ( )
{
cout << "No. of managers under me are: "<<nom;
}
```

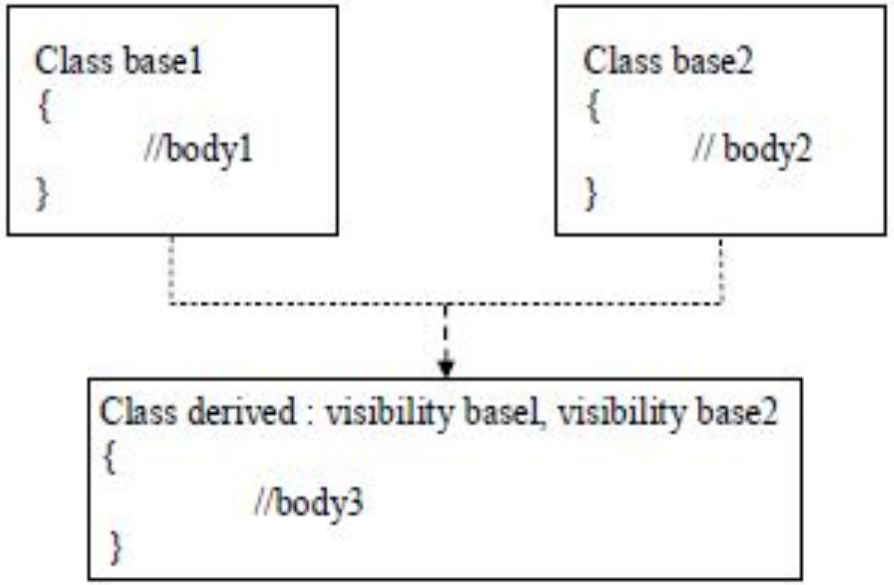
```
main ( )
{
clrscr ( ) ;
ceo cl ;
cl.get ( ) ; cout << "\n\n";
cl.show ( ) ;
}
```



Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features

of several existing classes as a starring point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is as follows:



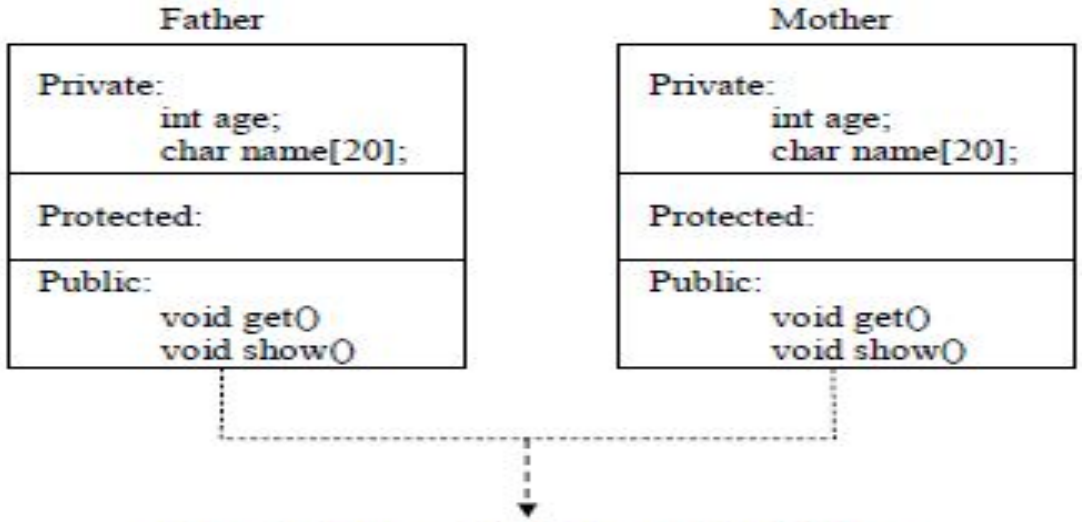
Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
#include<iostream.h>
#include<conio . h>
class father                                                    //Declaration of base classl
{
int age ;
char name [20] ;
public:
void get ( ) ;
void show ( ) ;
};
void father : : get ( )
{
cout << “your father name please”;
cin >> name;
cout << “Enter the age”;
cin >> age;
}
void father : : show ( )
{
cout<< “My father’s name is: ‘ <<name<< “My father’s age is:<<age;
}
```

```
class mother                                                    //Declaration of base class 2
{
char name [20] ;
int age ;
public:
void get ( )
{
cout << "mother's name please" ;
cin >> name;
cout << "mother's age please" ;
cin >> age;
}
void show ( )
{
cout << "My mother's name is: " <<name;
cout << "My mother's age is: " <<age;
}
class daughter : public father, public mother  //derived class inheriting publicly the features of both the base class
{
char name [20] ;
int std;
public:
void get ( ) ;
void show ( ) ;
};
```

```
void daughter :: get ( )
{
father :: get ( ) ;
mother :: get ( ) ;
cout << “child's name:
“;  cin >> name;
cout << “child's
standard”;  cin >> std;
}
void daughter :: show ( )
{
father :: show ( );
mother :: show ( ) ;
cout << “In child’s name is : “
<<name;  cout << “In child's standard:
“ << std;
}
main ( )
{
clrscr ( ) ;
```

Diagrammatic Representation of Multiple Inheritance is as follows:



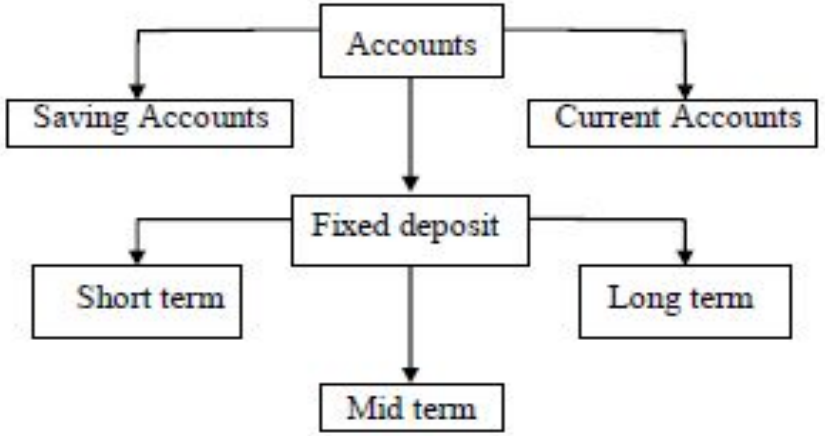
Class daughter: public Father, public Mother

Private: char name[20]; int age;
Protected:

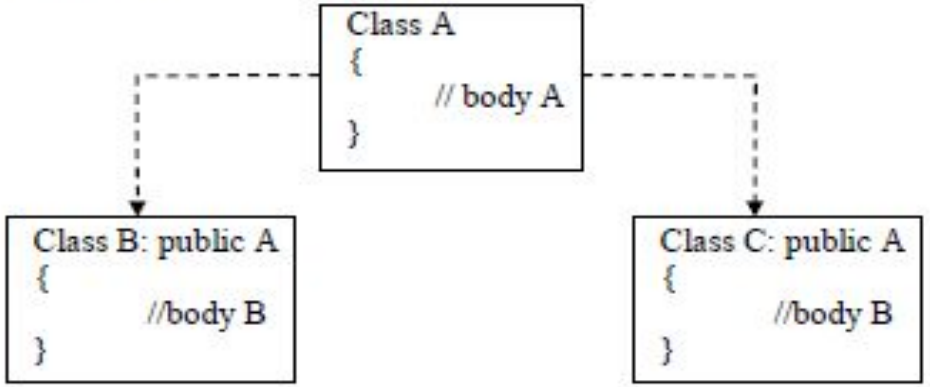
Public:
//self
void get(); void showQ;
//from Father
void get(); void show();
//from Mother
void get(); void show();

Hierarchical Inheritance

Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

```
// Program to show the hierarchical
 inheritance  #include<iostream.h>
# include<conio. h>
class father                          //Base class declaration
{
int age;
char name [15];
public:
void get ( )
{
cout<< “father name please”; cin >>
name;  cout<< “father’s age please”; cin
>> age;
}
void show ( )
{
cout << “father’s name is ‘: “<<name;
cout << “father’s age is: “<< age;
}
};
class son : public father                  //derived class 1
{
char name [20] ;
int age ;
public;
void get ( ) ;
void show ( ) ;
} ;
```

```
void son : : get ( )
{
father :: get ( ) ;
cout << “your (son) name please” ; cin
>>name;  cout << “your age please” ; cin>>age;
}
void son :: show ( )
{
father : : show ( ) ;
cout << “my name is : “
<<name;  cout << “my age is : “
<<age;
}                                                    //derived class 2.
class daughter : public father
{
char name [15] ;
int age;
public:
void get ( )
{
cout <<“your (daughter’s) name  please”;
father : : get ( ) ;
cin>>name;  cout << “your age please”; cin >>age;
}
```

```
void show (
)
father : : show ( ) ;
{
cout << “my name is: “ <<
name;  cout << “my age is: “
<<age;
}
};
main ( )
{
clrscr ( ) ;
son S1;
daughter D1 ;
S1. get ( ) ;
D1. get ( ) ;
S1 .show( ) ;
D1. show ( ) ;
}
```

Hybrid Inheritance

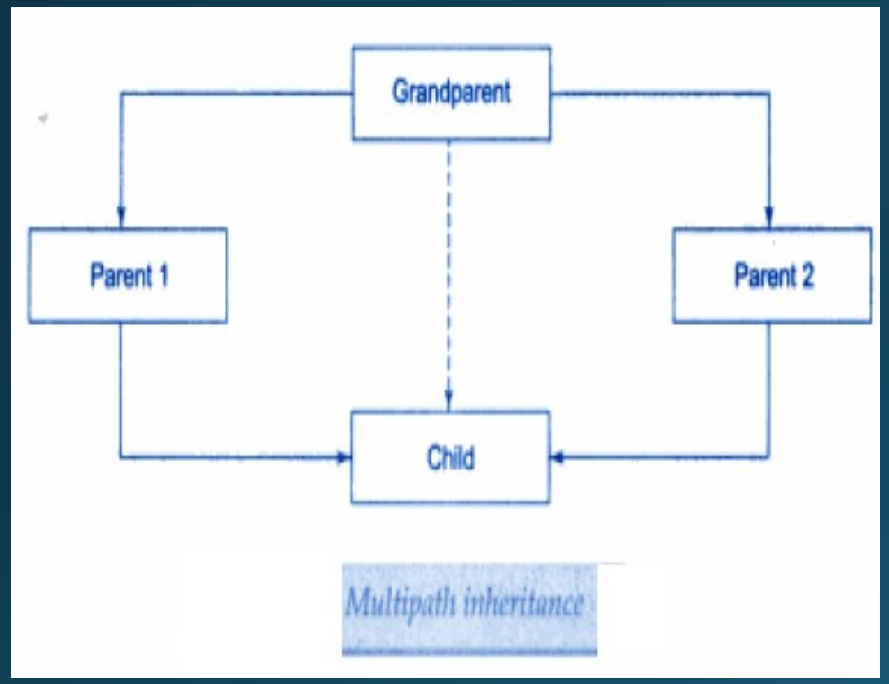
There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

```
//Program to show the simple hybrid
inheritance  #include<i sos t ream. h>
#include<conio . h>
class student                          //base class declaration
{
protected:
int r_no;
public:
void get _n (int a)
{
r_no =a;
}
void put_n (void)
{
cout << “Roll No. : “<< r_no;
cout << “\n”;
}
};
```

```
class test : public student            //Intermediate base class
{
protected : int part1, part2;
public :
void get_m (int x, int y)
{ parti = x; part 2 = y; }
void put_m (void) {
cout << "marks obtained: " << "In"
<< "Part 1 = " << part1 << "in"
<< "Part 2 = " << part2 << "In";
}
};
class sports                                        // base for result
{
protected : int score;
public:
void get_s (int s)
 { score = s }
void put_s (void)
{ cout << " sports wt. : " << score << "\n\n"; }
};
class result : public test, public sports          //Derived from test & sports
{
int total;
public:
void display (void);
};
```

```
void result : : display (void)
{
total = part1 + part2 + score;
put_n ( ) ;.
put_m ( );
put_S ( );
cout << “Total score: “ <<total<<
“\n”
}
main ( )
{
clrscr ( ) ;
result S1;
S1.get_n (347) ;
S1.get_m (30, 35);
S1.get_s (7) ;
S1.dciplay ( ) ;
}
```

Virtual Base Classes



The duplication of the inherited members can be avoided by making common base class as the virtual base class:
for e.g.

```
class g_parent
{
//Body
};
class parent1: virtual public g_parent
{
// Body
};
class parent2: public virtual g_parent
{
// Body
};
class child : public parent1, public parent2
{
// body
```

The 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’.

The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line.

The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’. Now, the inheritance by the child might cause some problems.

All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’.

So, there occurs a duplicacy which should be avoided.

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class.

The keywords ‘virtual’ and ‘public’ can be used in either order.

```
//Program to show the virtual base class
#include<iostream.h>
#include<conio . h>
class student // Base class declaration
{
protected:
int r_no;

public:
void get_n (int a)
{ r_no = a; }

void put_n (void)
{ cout << “Roll No. “ << r_no<< “ln”;}
};

class test : virtual public student // Virtually declared common { //base  class 1
protected:
int part1;  int part2;

 public:
void get_m (int x, int y)
{ part1= x; part2=y;}
void putm (void)
{
cout << “marks obtained: “ << “\n”;  cout << “part1 = “ <<
part1 << “\n”;  cout << “part2 = “<< part2 << “\n”;}
};
```

```
class sports : public virtual student // virtually declared common { //base
class 2  protected:
int score;
public:
void get_s (int a)
{  score = a ;
}
void put_s (void)
{ cout << “sports wt.: “ <<score<< “\n”;}
};
class result: public test, public sports //derived class
{
private : int total
;  public:
void show (void) ;
};
void result : : show (void)
{ total = part1 + part2 + score
;  put_n ( );
put_m ( );
```

```
main ( )
{
clrscr ( ) ;
result S1 ;
S1.get_n
(345)
S1.get_m (30, 35)
;
S1.get-S (7) ;
S1. show ( ) ;
}
```

```
//Program to show hybrid inheritance using virtual base
classes  #include<iostream.h>
#include<conio.h
>  Class A
{
protected:
int x; public:
void get (int) ;
void show (void)
;
};
void A : : get (int a)
{
x = a ;
}
void A : : show (void)
{
cout << X ;
}
```

```
Class A1 : Virtual Public
A
{
protected:
int y ;
public:
void get (int) ;
void show
(void);
};
void A1 :: get (int a)
{ y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public
A
{
protected:
```

```
class A12 : public A1, public
A2
{
int r, t ;
public:
void get (int a)
{ r = a;}
void show (void)
{ t = x + y + z + r ;
cout << “result =” << t ;
}
};
main ( )
{
clrscr ( )
;  A12 r ;
r.A : : get (3) ;
r.A1 : : get (4) ;
r.A2 : : get (5)
```

Abstract Classes

An *abstract class* is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built.

Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class alpha {....};
class beta {....};
class gamma
{
        alpha a;            // a is an object of alpha class
        beta b;             // b is an object of beta class
        .....
};
```

All objects of **gamma** class will contain the objects **a** and **b**. This kind of relationship is called *containership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
        .....
        alpha a;          // a is object of alpha
        beta b;           // b is object of beta
  public:
        gamma(arglist): a(arglist1), b(arglist2)
        {
                  // constructor body
        }
};
```

arglist is the list of arguments that is to be supplied when a **gamma** object is defined. These parameters are used for initializing the members of **gamma**. *arglist1* is the argument list

for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglist1* and *arglist2* may or may not use the arguments from *arglist*. Remember, **a**(*arglist1*) and **b**(*arglist2*) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
gamma(int x, int y, float z) : a(x), b(x,z)
{
        Assignment section(for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

REFERENCES:

1. E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.

2.LECTURE NOTES ON Object Oriented Programming Using C++ by Dr. Subasish Mohapatra, Department of Computer Science and Application College of Engineering and Technology, Bhubaneswar Biju Patnaik University of Technology, Odisha

3. K.R. Venugopal, Rajkumar, T. Ravishankar, “Mastering C++”, Tata McGraw-Hill Publishing Company Limited

4. Object Oriented Programming With C++ - PowerPoint Presentation by Alok Kumar

5. OOPs Programming Paradigm – PowerPoint Presentation by an Anonymous Author