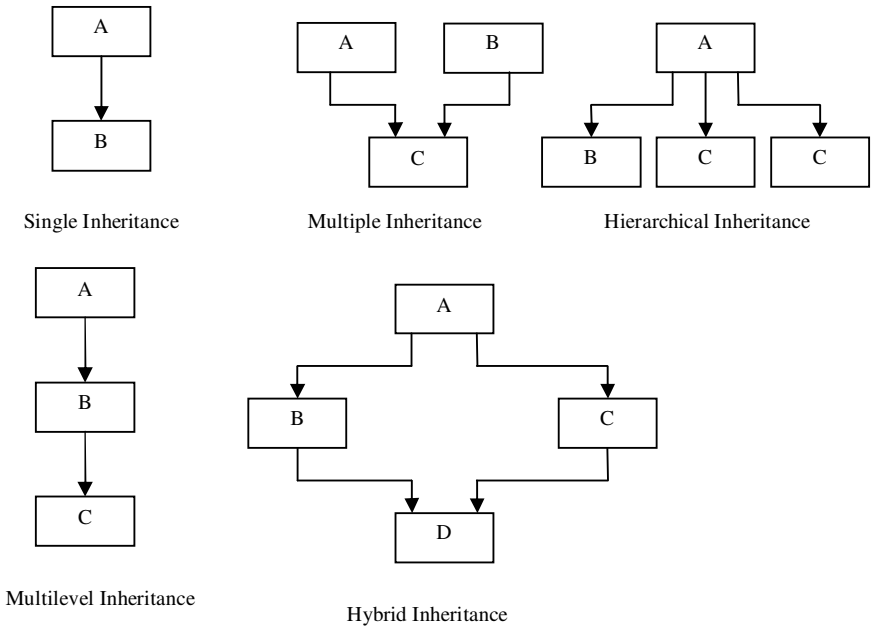
Inheritance: Extending Classes

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save lime and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of ***reusability.*** The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called ***inheritance (or derivation).*** The old class is referred to as the ***base class*** and the new one is called the ***derived class or subclass.***

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called ***single inheritance*** and one with several base classes is called ***multiple inheritance****.* On the other hand, the traits of one class may be inherited by more than one class. This process is known as ***hierarchical inheritance.*** The mechanism of deriving a class from another 'derived class' is known as ***multilevel inheritance****.* Following figure shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the opposite direction meaning inherited from.



Forms Of Inheritance

Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
    .....
    .....  //  members of derived class
    .....
};
```

The colon indicates that the ***derived-class-name*** is derived from the ***base-class-name.*** The ***visibility-mode*** is optional and, if present, may be either **private or public.** The default visibility-mode is **private.** Visibility mode specifies whether the features of the base class ***privately derived or publicly derived.***

Examples:

```
class ABC: private XYZ
{
    members of ABC
};

class ABC:   public XYZ
{
    members of ABC
};
```

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 only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the ***dot operator.*** The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class ***is publicly inherited,*** **'public members'** of the class become **'public members'** of the derived class and therefore they are accessible to the objects of the derived class. In ***both the cases, the private members are not inherited*** and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the functionality of the existing classes, becomes a very powerful tool for incremental program development.

Single Inheritance

Let us consider a simple example to illustrate inheritance. Following program shows a base class B and a derived class D, The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

```
#include <iostream.h>
class B
{
   int a;

   public:

   int b;

   void get_ab();

   int get_a(void);

   void show a(void);
};

void B :: get_ab(void)
{
       a = 5;
       b = 10;
}

int B :: get_a( )
{
       return a;
}

void B :: snow_a( )
{
       cout  «  "a = " « a « "\n";
}

class  D   :   public  B
{
    int c;

    public:

    void mul (void);

    void display (void);
};

void D :: mul()
{
   c = b * get_a( );
}
```

```
void  D :: display( )
{
        cout    « "a =     "  « get_a() « "\n";
        cout    « "b =     "  « b  « "\n";
        cout    « "c =     " « c « "\n\n";
}

int main ()
{
         D d;
        d.get_ab ( );
d.mul ();
d.show_a ();
d.display ( );

        d.b = 20;
        d.mul ();
        d.display ();

        return 0;
}
```

Given below is the output of above program

```
a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100
```

The class D is a public derivation of the base class B. Therefore, D inherits all the **public** members of B and retains their visibility. Thus a **public** member of the base class B is also a public member of the derived class **D.** The **private** members of B cannot be inherited by D. The class D, in effect, will have more members than what it contains at the time of declaration as shown in following Fig.

Making a Private Member Inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited; it is not available for the derived class directly. What do we do if the **private** data member be inherited by a derived class? This can be accomplished by modifying the visibility limit of the **private** member by making it **public.** This would make it accessible to all the functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, **protected,** which serve a limited purpose in 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 the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
    private:     //    optional
            ... ...  //    within its class
            ... ...
    protected :
            ... ...  //    visible to members functions
            ... ...  //    of its own and derived class
    public :
            ... ...  //    visible to all functions
            ... ...  //    in the program
};
```

When a **protected** member is inherited in public mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member indented in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private members** cannot be inherited). Following Figure is the pictorial representation for the two levels of derivation.

The keywords private, protected, and public may appear in any order and any numbers of times in the declaration of class. It is also possible a base class in protected mode known as protected derivation. In protected derivation, both the public and protected members of the base class become protected members of the derived class. Following table summarizes how the visibility of base class members undergoes modification in all three types of derivation.

While the friend function and the member functions of a friend class can have direct access to both the private and protected data, the member functions of a derived class can directly access only the protected data. However they can access the private data through the member functions of the base class.

Base class visibility	**Derived class visibility**		
	public derivation	**private derivation**	**protected derivation**
Private	**Not inherited**	**Not inherited**	**Not inherited**
Protected	**protected**	**private**	**protected**
public	**public**	**private**	**protected**

Multilevel Inheritance :

A class is inherited from a base class which in tern inherited from another base class,

```
#include<iostreaam.h>

class student
{
     protected:
          int roll_number;
     public:
          void get_number(int a)
          {
               roll_number = a;
          }
          void put_number ( )
          {
               cout << "Roll Number : " << roll_number << "\n';
          }
};

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 subject 1 : " << sub1 << "\n";
          cout << "Marks in subject 2 : " << sub2 << "\n";
     }
};

class result : public test
{
     float total;
     public :
          void display(void)
          {
               total = sub1 + sub2;
               put_number ( );
               put_marks ( );
               cout << "Total = " <<  total < "\n";
          }
};
```

```
int main( )
{
      result student1;
      student1.get_number(111);
      student1.get_marks(75.0, 59.5);
      student1.display( );
}
```

Multiple inheritance

A class can inherit the attribute of two or more classes is known as multiple inheritance.
For example :

```
#include<iostream.h>
class M
{
      protected:
            int m;
      public :
      void get_m(int x)
      {
            m = x;
      }
};

class N
{
      protected:
            int n;
      public :
      void get_n(int y)
      {
            n = y;
      }
};

class P : public M, public N
{
      public :
      void display ( )
      {
            cou << "m = " << m << "\n";
            cou << "n = " << n << "\n";
            cou << "m * n = " << m * n << "\n";
      }
};

int main( )
{
      P p;

      p.get_m(10);
      p.get_n(20);
      p.display( );

      return 0;
}
```

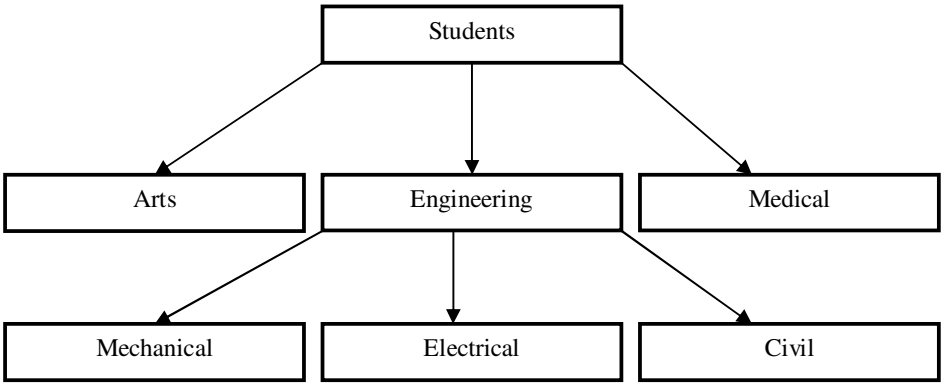
Ambiguity resolution in inheritance

In using multiple inheritance where a function with the same name appears in more than one base class then it can be resolved using scope resolution operator. i.e. M :: display() ; N :: display( );

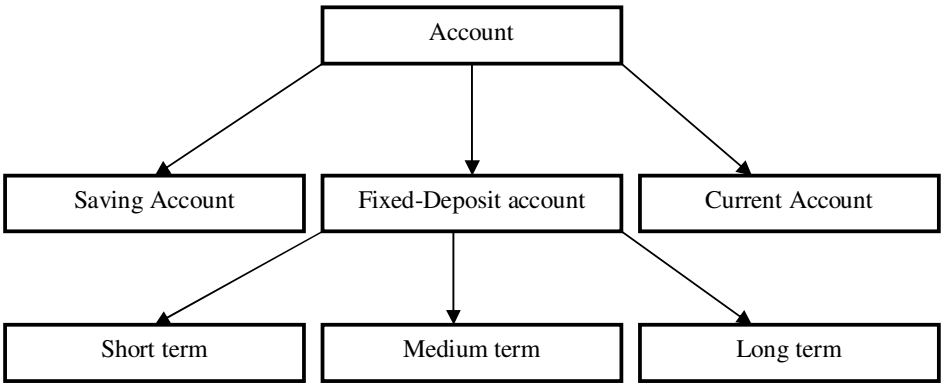
Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is a support to the hierarchical design of a program. Many programming problems can be cast, into a hierarchy where certain features of one level shared by many others below that level

As an example following Fig. shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in second Fig.



Hierarchical classification of students



Hierarchical classification of bank accounts

In **C++** such a classification problems can be easily converted into a class hierarchies. The base class will include the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower classes and so on.

Hybrid inheritance

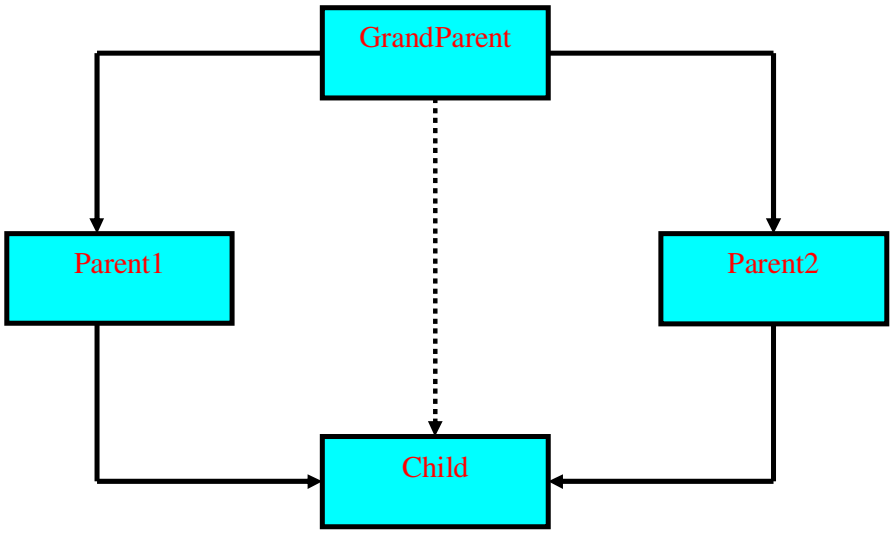
There could be situations where we need to apply two or more type of inheritance to design a program. For example consider the student result discussed above. Assume that we have to give weightage for sports before finalizing the results. The weightage for sports stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in following figure.

```
#include<iostreaam.h>
class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number ( )
        {
            cout << "Roll Number : " << roll_number << "\n';
        }
};

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 subject 1 : " << sub1 << "\n";
        cout << "Marks in subject 2 : " << sub2 << "\n";
    }
};

class sports
{
    protected:
        float score;
    public:
        void get_score (float s)
        {
            score = s;
        }
        void put_score ( )
        {
            cout << "Sports weight : " << score << "\n';
        }
};
```

```
class result : public test, public sports
{
      float total;
      public :
            void display(void)
            {
                  total = sub1 + sub2 + score;
                  put_number ( );
                  put_marks ( );
        put_total ( );
                  cout << "Total  Score= " <<  total < "\n";
            }
};
int main( )
{
      result student1;
      student1.get_number (1234);
      student1.get_marks (27.50, 33.0);
      student1.get_score(6.);
      student1.display ( );
}
```

virtual Base Classes

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inherits multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in following Fig. The **'child'** has two ***direct base classes*** **'parent1' and 'parent2'** which' have a common base class **'grandparent'.** The **'child'** inherits the traits of **'grandparent'** two separate paths. It can also inherit directly as shown by the broken line. The **'grandparent'** is sometimes referred to as ***indirect base class*****.**



Multipath Inheritance

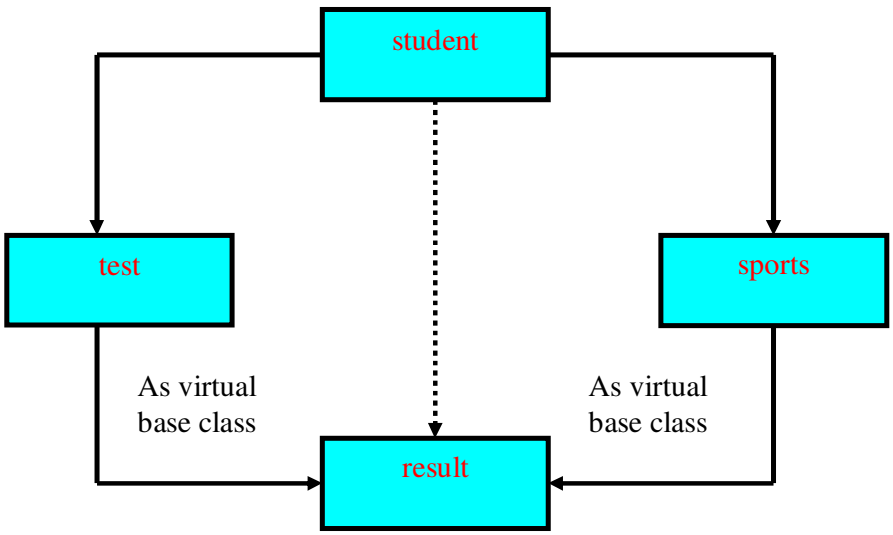
.Inheritance by the 'child' as shown in above Fig. might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have ***duplicate*** sets of the members inherited from grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by using the common base class (ancestor class) as ***virtual base class*** while declaring the direct or intermediate base classes as shown below:

```
Class A     //      grandparent
{
     … …
     … …
};
class B1 : virtual public A       //      parent1
{
     … …
     … …
};
class B2 : public virtual A       //      parent2
{
     … …
     … …
};

class C : public B1, public B2
{
     … …          //      only one copy of A will be inherited
     … …
};
```

When a class is made a virtual base class, C++ takes necessary care to see tat only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a friend classes.

A program to implement the concept of virtual base class is given below.



Multipath Inheritance : virtual base class

```
#include<iostreaam.h>

class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number ( )
        {
            cout << "Roll Number : " << roll_number << "\n';
        }
};

class test :: virtual 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 subject 1 : " << sub1 << "\n";
        cout << "Marks in subject 2 : " << sub2 << "\n";
    }
};

class sports : public virtual student
{
    protected:
        float score;
    public:
        void get_score (float s)
        {
            score = s;
        }
        void put_score ( )
        {
            cout << "Sports weight : " << score << "\n';
        }
};
```

```
class result : public test, public sports
{
      float total;
      public :
            void display(void)
            {
                  total = sub1 + sub2 + score;
                  put_number ( );
                  put_marks ( );
        put_total ( );
                  cout << "Total  Score= " <<  total < "\n";
            }
};

int main( )
{
      result student1;
      student1.get_number (678);
      student1.get_marks (30.50, 25.5);
        student1.get_score (7.0);
      student1.display ( );
}
```

Abstract Classes

An ***abstract*** class is one that is not used to create objects. An abstract class is designed 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. In student's result example, the student class is an abstract class since it was not used to create any object.

Constructors in Derived Classes

As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is ***mandatory*** for the derived class to have a constructor **and** pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed ***in the order in which they appear in the declaration of the derived class.*** Similarly, in a multilevel inheritance, the constructors will be executed ***in the order of inheritance.***

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statement of the derived constructor.

General form of defining a derived constructor is:

```
Derived – constructor (arglist1, arglist2, … … arglistN, arglist(D) :
            base1(arglist1),
            base2(arglist2),
            … …
            … …
            baseN(arglistN),
{
     Body of derived constructor
}
```

The header line ***of derived-constructor*** function contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the base class constructors and the second part lists the function calls to the base constructors.

Base1(arglist1), base2(arglist2) … … are function calls to base constructors base1( ), base2( ), … … and therefore arglist1, arglist2,… …. arglistN etc. represent the actual parameters that are passed to the base class constructors. Arglist1 through arglistN are the argument declarations for base class constructors base1 through baseN. arglistD provides the parameters that are necessary to initialize the members of the derived class.

Example:

```
D (int al, int a2, float b1, float b2, int dl):
A (a1, a2),     /* call to constructor A */
B (bl, b2)      /* call to constructor B *,/
{
  d = dl;   // executes its own body
}
```

A(al, a2) invokes the base constructor A( ) and B(bl,b2) invokes another base class constructor B( ). The constructor D ( ) supplies the values for these four arguments. In addition it has one argument of its own. The constructor D ( ) has a total of five arguments. D ( ) may be invoked as follows:

```
… …
D objD (5, 12, 2.5, 7.54, 30);
```

Method of Inheritance	Order of execution

Class B : public A
{
};
	A( ); Base constructor
	B( ); Derived constructor

Class A : public B, public C
{
};
	B( ); Base constructor first
	C( ); Base constructor first
	A ( ); Derived constructor

Class A : public B, virtual public C
{
};
	C( ); Virtual Base constructor first
	C( ); Ordinary Base constructor first
	A ( ); Derived constructor

Execution of base class constructors.	

```
#include<iostream.h>
class alpha
{
      int x;
      public :
      alpha(int i)
        {
              x = i;
              cout << "alpha initialized \n";
        }
        void show_x (void)
        {
              cout << "x = " << x << "\n";
        }
};
class beta
{
      int p, q;
      public :
      beta (int j)
        {
              y = j;
              cout << "beta initialized \n";
        }
        void show_y (void)
        {
              cout << "y = " << y << "\n";
        }
};
class gamma : public beta, public alpha
{
      int m, n;
      public :
      gamma (int a, int b, int c, int d) :
      alpha(a), beta(b)
        {
              m = c;
              n = d;
              cout << "gamma initialized \n";
        }
        void show_mn (void)
        {
              cout << "x = " << x << "\n";
        }
};
int main( )
{
      gamma g(5, 10.75, 20, 30);

      cout << "\n";
      g.show_x ( );
      g.show_y ( );
      g.show_mn ( );

      return 0;
}
```

Here in above example beta is initialized first, although it appears second in the derived class because it has been declared first in the derived class header line. Output will be as follows

beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30

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 vet 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 a class can contain objects of other classes as its members as shown below:

```
classalpha  {....};
classbeta  {....};
classgamma
{
        alpha a;
        beta b;
};
```

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 objects should be called before its own constructor body is executed. This is accomplished by using an initialization list in the constructor of the nested class.
Example:

```
classgamma
{
        … …
        alpha a;    // a is object of alpha
        beta b;             // b is object of beta
        public :
        gamma (arglist) : a(arglist1, arglist(2)
        {
             // 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 be variables or constants.

Example

```
gamma (int x, int y, int z) : a(x), b(x, z)
{
    // constructor body
    // Assignment section for ordinary the members
}
```

We can use as many member objects as are required in a class. For each member object we add 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.