# Inheritance: Extending Classes

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.***

Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs.

A

**B**

**C**

**A**

**B**

a) Single inheritance b)Multiple inheritance

**B**

c) Multilevel inheritance d)Hierarchical inheritance

**D**

**A**

**B**

**C**

**B**

**C**

**D**

**A**

**B**

**C**

(e) Hybrid 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 are *privately derived or publicly derived.*

Examples:

**class ABC: private XYZ *II private derivation***

**{**

**members of ABC**

**} ;**

**class ABC: public XYZ *II public derivation***

**{**

**members of ABC**

**} ;**

**class ABC: XYZ *II private derivation by default***

**{**

**members of ABC**

}

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 ofthe base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# SINGLE INHEIRITANCE

**A derived class with only one base class, is called *single inheritance***. The below Program shows a base class B and a derived class D. The class B contains one private data member, one public data member, and one public member functions. The class D contains one private data member and one public member functions.

#include<iostream.h> class B

{

int a; / / *private; not inheritable*

public:

int b; //*public; ready for inheritance*

void display()

{

cout<< ―a=‖<<a<<endl; cout<< ―a=‖<<a<<endl;

}

};

class D : **public** B // *public derivation*

{ public:

int c;

void show()

{

cout<< ―c=‖<<c<<endl;

}

} ;

main()

{

D d;

//d.a=10; // Not possible, because ‗a‘ is private data d.b=20;

* 1. =30;
  2. isplay();

d.show();

}

**OUTPUT:**

**a=1042 //because ‗a‘ cannot be accessible b=20**

**c=30**

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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Let us now consider the case of private derivation.**

#include<iostream.h> class B

{

int a; / / *private; not inheritable*

public:

int b; //*public; ready for inheritance*

void display()

{

cout<< ―a=‖<<a<<endl; cout<< ―a=‖<<a<<endl;

}

};

class D : **private** B // *public derivation*

{ public:

int c;

void show()

{

cout<< ―c=‖<<c<<endl;

}

} ;

main()

{

D d;

//d.a=10; // Not possible, because ‗a‘ is private data

//d.b=20; //Not accessible , because *b has become private*

d.c=30;

//d.display(); //Not accessible d.show();

}

**OUTPUT:**

**c=30**

In **private** derivation, the **public** members of the base class become **private** members of the derived class. Therefore, the objects of **D** can not have direct access to the public member functions of **B**.

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. The base class function. will be called only if the derived class does not redefine the function.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 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 and therefore it is not available for the derived class direct! **'What do we do if the private data needs to 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 other functions of the program, thus taking away the advantage of data hiding.

C++ provides a **third *visibility modifier,* protected.** A member declared as protected is accessible by the member functions within its class and any class *immediately* derived from, it. It *cannot* be 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*

/ / *visible to member functions within its class*

protected:

........................ / / *visible to member 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, inherited 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).

It is also possible to inherit 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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# MULTILEVEL INHERITANCE

A class can be derived from another derived class is known as multilevel inheritance. The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as *intermediate* base class since it provides a link for the inheritance between A and C. The chain **ABC** is known as *inheritance path.*

A derived class with multilevel inheritance is declared as follows:

class A

{

... .. ...

};

class B: public A

{

... .. ...

};

class C: public B

{

... ... ...

};

Here, class B is derived from the base class A and the class C is derived from the derived class B.

## Example 1: C++ Multilevel Inheritance

#include <iostream>

using namespace std;

class A

{

public:

void display()

{

cout<<"Base class content.";

}

};

class B : public A

{

};

class C : public B

{

};

int main()

{

C obj;

obj.display();

return 0;

}

**Output**

Base class content.

In this program, class C is derived from class B (which is derived from base class A).

The obj object of class C is defined in the main() function.

When the display() function is called, display() in class A is executed. It's because there is no display() function in class C and class B.

The compiler first looks for the display() function in class C. Since the function doesn't exist there, it looks for the function in class B (as C is derived from B).

The function also doesn't exist in class B, so the compiler looks for it in class A (as B is derived from A).

If display() function exists in C, the compiler overrides display() of class A (because of [member function overriding](https://www.programiz.com/cpp-programming/function-overriding)).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# MULTIPLE INHERITANCE

**A class can inherit the attributes of two or more classes is known as *multiple inheritance****.* Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes.

**The syntax of a derived class with multiple base classes is as follows:**

**Class D: visibility B1, visibility B2, ....**

**{**

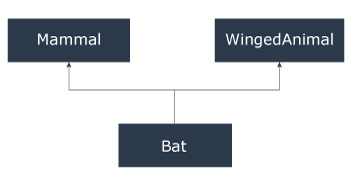
**...............**

**...............(Body of D)**

**}**

Where visibility may be either public or private. The base classes are separated by commas.

**Example:**



## Example 2: Multiple Inheritance in C++ Programming

This program calculates the area and perimeter of an rectangle but, to perform this program, multiple inheritance is used.

#include <iostream>

using namespace std;

class Mammal {

public:

Mammal()

{

cout << "Mammals can give direct birth." << endl;

}

};

class WingedAnimal {

public:

WingedAnimal()

{

cout << "Winged animal can flap." << endl;

}

};

class Bat: public Mammal, public WingedAnimal {

};

int main()

{

Bat b1;

return 0;

}

**Output**

Mammals can give direct birth.

Winged animal can flap.

# Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same inheritance appears in more than one base class. **Consider the following two classes.**

class M

{

public:

void display(void)

{

cout « "Class M\n";

}

} ;

class N

{

public:

void display(void)

{

cout « "Class N\n";

}

} ;

class P : public M, public N

{

public:

void display(void)

{

Which display() function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

class P : public M, public N

{

public:

void display(void) // *overrides display() of M and N*

{

M :: display();

}

} ;

**We can now use the derived class as follows:**

int main()

{

P p;

p . display();

}

Ambiguity may also arise in single inheritance applications. For instance, consider the

following situation: class A

{

*public:*

*void display()*

{ cout<<‖A‖<<endl;

}

};

class B : public A

{

public:

void display()

{

cout « "B:<<endl;

}

} ;

In this case, the function in the derived class overrides the Inherited function and, therefore, a simple call to **display()** by B type object will invoke function defined in B only. However, we may invoke the function defined in A by using the scope resolution operator to specify the class.

Example: int main()

{

B b; *//* derived class object

b . display () ; *//*invokes display() in B b . A: : d i s play () ; *//*invokes display() in A b . B: : d i s play () ; *//* invokes display() in B

}

**This will produce the following output:**

**B**

**A B**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# HIERARCHICAL INHERITANCE

One class may be inherited by more than one class. This process is known as *hierarchical inheritance.*The base class will include all 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 level classes and so on.

### Syntax of Hierarchical Inheritance

class base\_class {

... .. ...

}

class first\_derived\_class: public base\_class {

... .. ...

}

class second\_derived\_class: public base\_class {

... .. ...

}

class third\_derived\_class: public base\_class {

... .. ...

}

**Example :**

|  |
| --- |
| // C++ program to implement  // Hierarchical Inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };      // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Car obj1;      Bus obj2;      return 0;  } |

**Output :**

This is a Vehicle

This is a Vehicle

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# HYBRID INHERITANCE

There could be situations where *We* need to apply two or more types of inheritance to design

a program.

|  |
| --- |
| // C++ program for Hybrid Inheritance    #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };    //base class  class Fare  {      public:      Fare()      {          cout<<"Fare of Vehicle\n";      }  };    // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle, public Fare  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Bus obj2;      return 0;  } |

**Output:**

This is a Vehicle

Fare of Vehicle

The example illustrates the implementation of both multilevel and multiple inheritance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# VIRTUAL BASE CLASSES

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class.*

Parent 2

Child

Grandparent

Parent 1

**Fig. 8.1 2***Multipath inheritance*

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 making the common base class (ancestor class) as *virtual base class* while declaring the**

Direct or intermediate base classes as shown

Class A //grandparent

{

.............

............

};

Class B1:virtual public A //parent1

{.............

................

};

Class B2: public virtual A //parent2

{

................

................

};

Class C: public B1,public B2 //child

{

.............. //only one copy of A will be inherited

.............

};

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

### Example using virtual base class

#include<iostream.h>

#include<conio.h>

class ClassA

{

public:

int a;

};

class ClassB : **virtual** public ClassA

{

public:

int b;

};

class ClassC : **virtual** public ClassA

{

public:

int c;

};

class ClassD : public ClassB, public ClassC

{

public:

int d;

};

void main()

{

ClassD obj;

obj.a = 10; **//Statement 1**

obj.a = 100; **//Statement 2**

obj.b = 20;

obj.c = 30;

obj.d = 40;

cout<< "\n A : "<< obj.a;

cout<< "\n B : "<< obj.b;

cout<< "\n C : "<< obj.c;

cout<< "\n D : "<< obj.d;

}

**Output :**

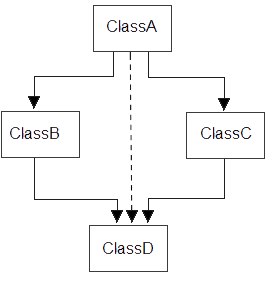
A : 100

B : 20

C : 30

D : 40

According to the above example, **ClassD** have only one copy of **ClassA** and statement 4 will overwrite the value of **a**, given in statement 3.



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ABSTRACT CLASSES

An *abstract class* is one that is not used to create object . 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.

#include<iostream.h>

#include<conio.h>

class BaseClass **//Abstract class**

{

public:

**virtual** void Display1()**=0;** **//Pure virtual function or abstract function**

**virtual** void Display2()**=0;** **//Pure virtual function or abstract function**

void Display3()

{

cout<<"\n\tThis is Display3() method of Base Class";

}

};

class DerivedClass : public BaseClass

{

public:

void Display1()

{

cout<<"\n\tThis is Display1() method of Derived Class";

}

void Display2()

{

cout<<"\n\tThis is Display2() method of Derived Class";

}

};

void main()

{

DerivedClass D;

D.Display1(); **// This will invoke Display1() method of Derived Class**

D.Display2(); **// This will invoke Display2() method of Derived Class**

D.Display3(); **// This will invoke Display3() method of Base Class**

}

**Output :**

This is Display1() method of Derived Class

This is Display2() method of Derived Class

This is Display3() method of Base Class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# CONSTRUCTORS IN DERIVED CLASSES

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 multi level 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 statements in the body of the derived constructor.

**The general form of defining a derived constructor is:**

Derived-constructor *(*Arglist1,Arglist2,........ ArglistN, Arglist D ):

base1(arglist1), base2 (arglistst2),

::::::::::::::::::::

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 *derived· constructor* and the second part lists the function calls to the base constructors.

*base1(arglist 1), base2(arglist2)* are function calls to base constructors base1(), base2(),

... and therefore *arglist1 , arglist2,* etc. represent the actual parameters that are passed to

the base constructors. *Arglist1* through *ArglistN* are the argument declarations for base constructors *base1* through *baseN. Arglist D* provides the parameters that are necessary to initialize the members of the derived class.

Example:

D(int a1 , int a2, float b1, float b2, int d1 ) : A(a1, a2),

/\* call to constructor A \*/

B(b1, b2) */\** call to constructor B \*/

{

d = d1; / / *executes its own body*

}

A(a1, a2) invokes the base constructor A() and B(bl, b2) invokes another base 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. DO may be invoked as follows:

D.objD(5, 12, 2.5, 7.54, 30);

These values are assigned to various parameters by the constructor DO as follows: 5 – a1

12 – a2

2.5—b1

7.54—b2

30 – d1

The constructors for virtual base classes are **invoked before any non-virtual base classes**.

If there are multiple virtual base classes, they are invoked in the **order in which they are declared**. Any non-virtual bases are then constructed before the derived class constructor is executed. See Table 8.2.

**Table 8.2 Execution of Base Class Constructors**

|  |
| --- |
| **Method of inheritance Order of execution**  Class B: public A A( ) ; base constructor  { B( ) ; derived constructor  };  class A : public B, public C B ( ) ; base(first)  { C( );base(second)  }; A( ) ; derived  class A : public B. virtual public C  {}; C( ) ; virtual base  B();ordinary  base A( ) ; derived |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# CONSTRUCTORS IN DERIVED CLASS

#include <iostream> us i ng names pace std;

class alpha

{

int X; public:

alpha(int i)

{

x = i;

cout « "alpha initialized \n";

}

void show\_x(void)

{ cout « "x = " « X « "\n"; }

} ;

Class beta

{

float y;. public:

beta(float j)

{

y = j;

cout « "beta initia1ized \n";

}

void show\_y(void)

{ cout «"y ="« y« "\n"; }

} ;

class gamma: public beta,public alpha

{

int m, n;

public:

gamma(int a,float b,int c,int d):

alpha(a), beta(b)

{ m=c;

n = d;

cout « "gamma initialized \n";

}

void show\_mn(void)

{

cout << "m = " « m « "\n"

cout<< "n = " << n « "\n";

}

} ;

int main()

{

gamma g(5,10.75,20,30);

g.show\_x();

g.show\_y();

g.show\_mn();

}

**The output of Program 8.7 would be:**

**Beta initialized alpha initialized gamma initialized x=5**

**y=10.75 m=20 n=30**

**beta** is initialized first,. although it appears second in the derived constructor. This is because it has been declared first in the derived class header line.

Also, note that

**alpha(a)** and **beta(b**) are function calls. Therefore, the parameters should not include types.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# MEMBER CLASSES: NESTING OF 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; *II a* is *an object of alpha class*

beta b; *II 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

{

al pha a; *II a* is *an object of alpha class*

beta b; *II b* is *an object of beta class*

public:

gamma(arglist): a(arglist1), b(arglist2)

{

*I I 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(arglistl)* and *b(arglist2)* are function calls and therefore the arguments do not contain the data types, They are simply variables or constants.

|  |
| --- |
| #include <iostream.h> class Nest { public: class Display { private: int s; public: void sum( int a, int b) { s =a+b; } void show( ) { cout << "\nSum of a and b is:: " << s;} }; }; void main() { Nest::Display x; x.sum(12, 10); x.show(); } |

**Result :**

Sum of a and b is::22



**Pitfalls of operator overloading and conversion**

1. Use similar meanings:

We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus( +) operator to subtract one value from the other.

1. Use similar syntax:

Overloaded operators follow the syntax rules of the original operators. They cannot be overridden. Some syntactical characteristics of operators cannot be changed even if we want to.

If a and b are basic types, the assignment operator in the statement, a += b;

sets a=a+b;

Any overloaded version of this operator cannot be used in a operator overloading.

It can be

written only as, a = a+ b;

1. Show restraint:

If the number of operators grows too large, then listing becomes less readable. So when the usage is in doubt, use a function instead of an overloaded operator, since a function name can state its own purpose.

For example, if you write a function to find the left side of a string, then it is better to use getleft() than trying to overload some operator like && to do the same thing.

1. Avoid Ambiguity:

Suppose you use both a one argument constructor and a conversion function to perform the same conversion , then the compiler does not know what to do, and will signal an error. So avoid using same conversion in more than one way.

1. All operators cannot be overloaded:

The operators such as , member access ordot operator(.), the scope resolution operator(::) , the condi tional operator(?:) , pointer to the member operator (→) cannot be overloaded.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*