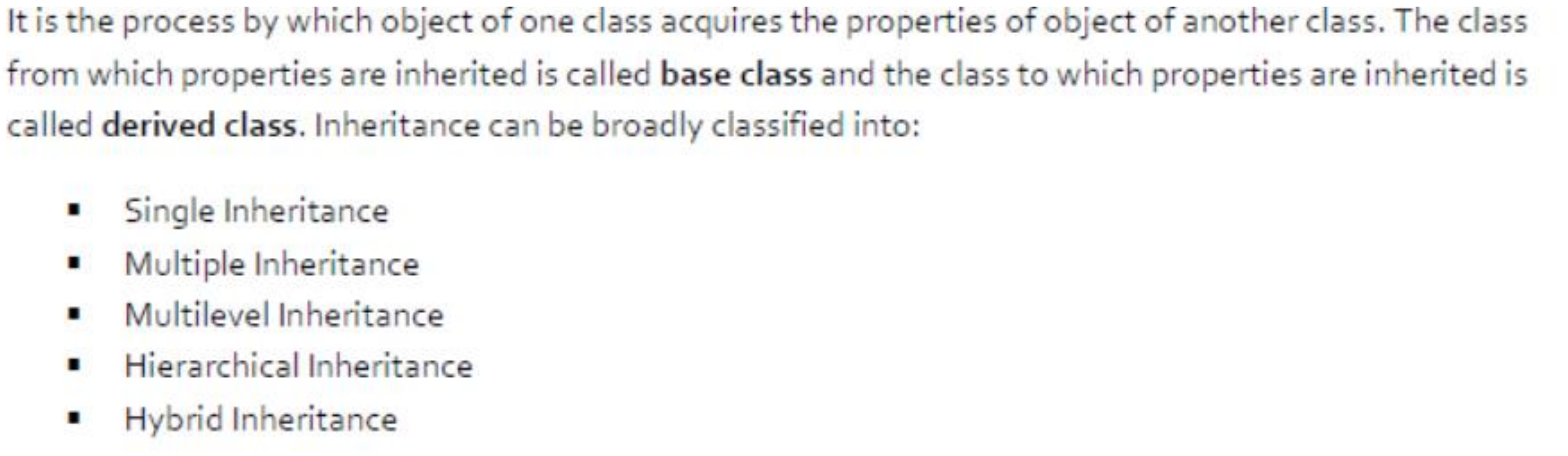
Chapter 3

INHERITANCE



It is the process by which object of one class acquires the properties of object of another class. The class from which properties are inherited is called **base class** and the class to which properties are inherited is called **derived class.** Inheritance can be broadly classified into:

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

BASE-CLASS ACCESS CONTROL

When a class inherits another, the members of the base class become members of the derived class.

Class inheritance uses this general form:

```
class derived-class-name : access base-class-name / visibility –mode base-class name
{
// body of class
};
```

Ex:

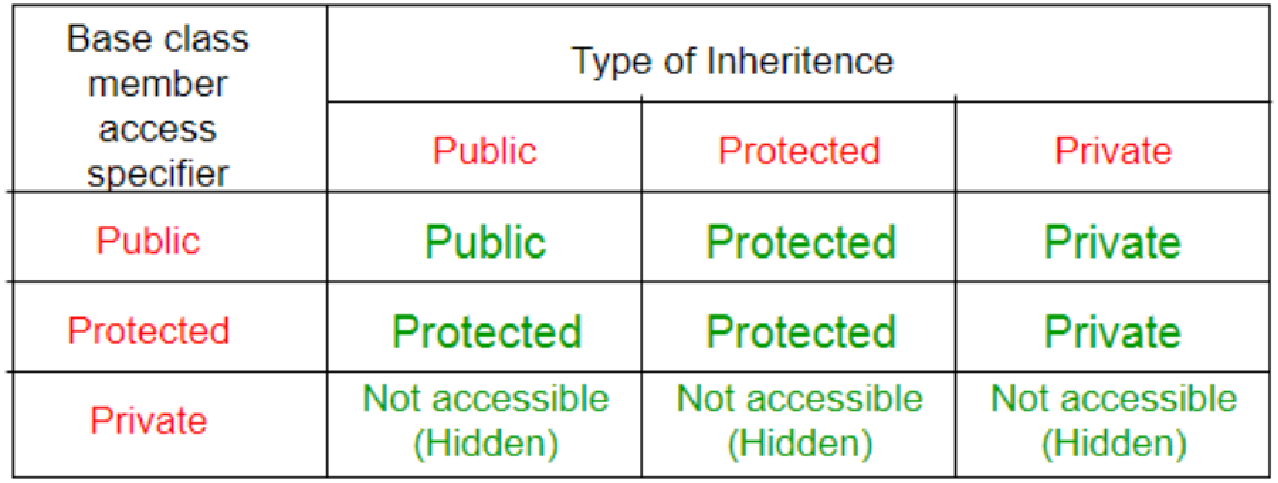
```
class parent
{
…
}
class child : private parent   //private derivation
{
….
}
```

MODE OF INHERITANCE

The access status of the base-class members inside the derived class is determined by access. The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier.

1. When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class.
2. When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class.
3. When a base class' access specifier is **protected,** public and protected members of the base become protected members of the derived class.

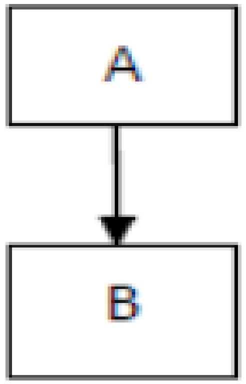
In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

SINGLE INHERITANCE

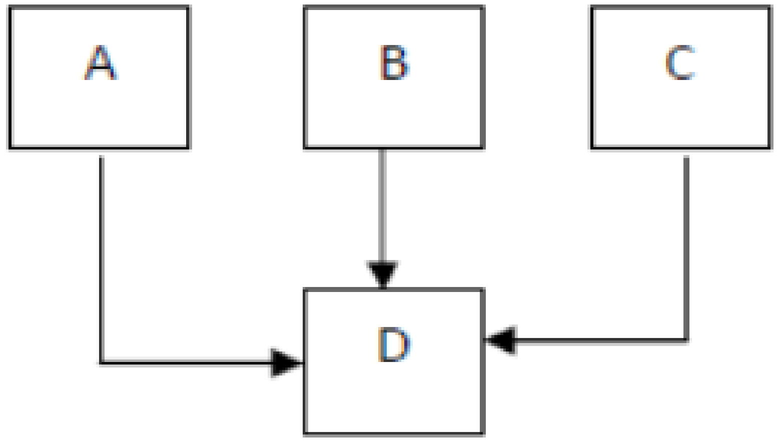
In a single inheritance the derived class is derived from a single base class.



(Single inheritance)

MULTIPLE INHERITANCE

In multiple inheritance derived class is derived from more than one base class.

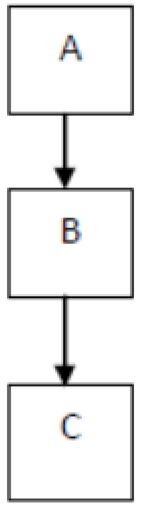


(Multiple Inheritance)

MULTILEVEL INHERITANCE

In multilevel inheritance class B is derived from a class A and a class C is derived from the class B.

Syntax:
class base-class-name1
{
Data members
Member functions
};
class derived-class-name : **visibility mode** base-class-name
{
Data members
Member functions
};
class derived-class-name1: **visibility mode** derived-class-name
{
Data members
Member functions
};
Note: visibility mode can be either private, public or protected
(Multilevel inheritance)
Hierarchical Inheritance



(Multilevel inheritance)

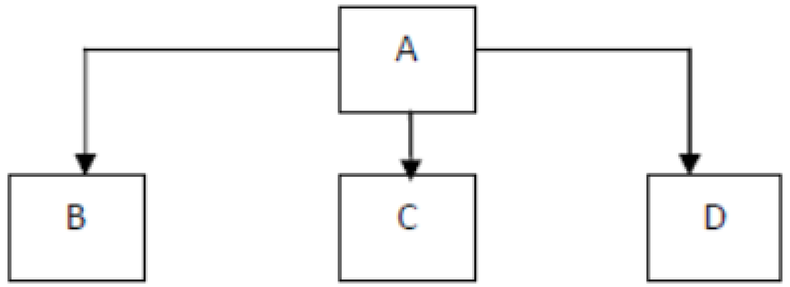
HIERARCHICAL INHERITANCE

In hierarchical inheritance several classes can be derived from a single base class

Syntax:

class base-class-name
{
Data members
Member functions
};
class derived-class-name1 : **visibility mode** base-class-name
{
Data members
Member functions
};
class derived-class-name2: **visibility mode** base-class-name
{
Data members
Member functions
};

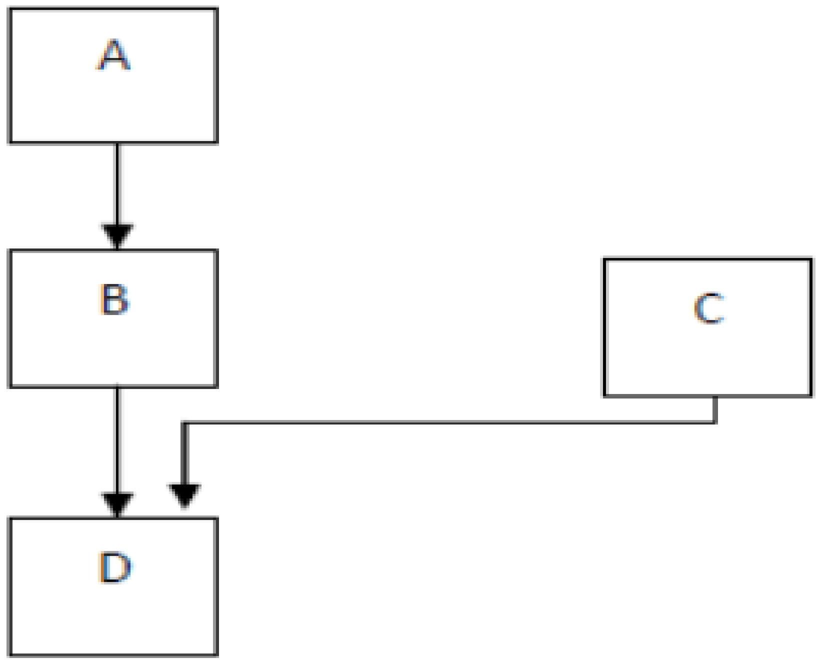
Note: visibility mode can be either private, public or protected



(Hierarchical inheritance)

HYBRID INHERITANCE

It is the mixture of one or more above inheritance.



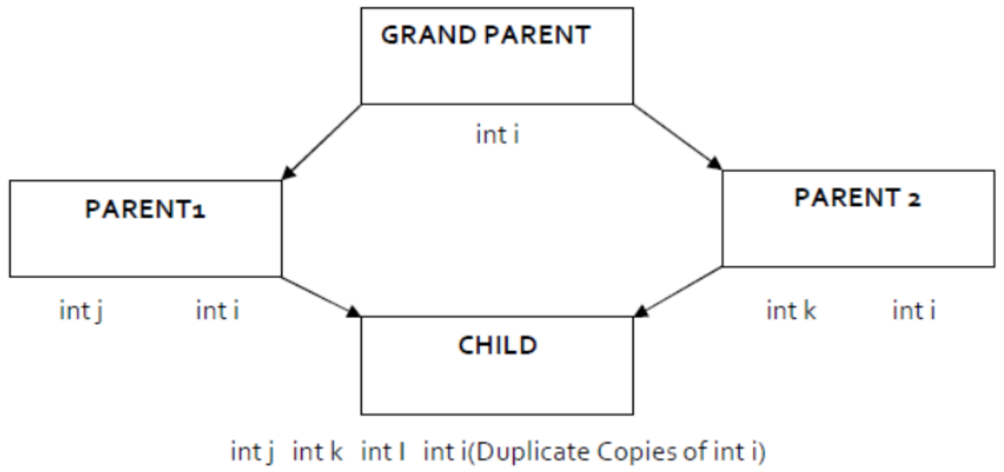
VIRTUAL BASE CLASSES

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

What is Multipath Inheritance?

Multipath Inheritance is a hybrid inheritance (also called as Virtual Inheritance). It is combination of hierarchical inheritance and multiple inheritance.

In Multipath Inheritance there is a one base class GRANDPARENT. Two derived class PARENT1 and PARENT2 which are inherited from GRANDPARENT. Third Derived class CHILD which is inherited from both PARENT1 and PARENT2



[Ambiguity in Multipath Inheritance]

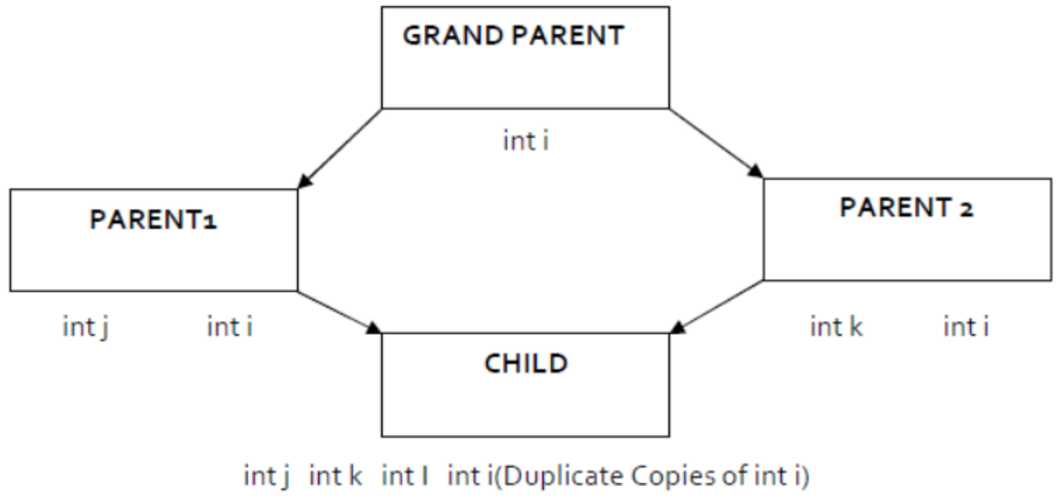
PROBLEM IN MULTIPATH INHERITANCE

There is an ambiguity problem. When we run program with such type inheritance, it gives a compile time error [Ambiguity]. If we see the structure of Multipath Inheritance then we find that there is shape like Diamond.

Why Ambiguity Problem in multipath (Virtual) Inheritance?

Suppose GRANDPARENT has a data member int i. PARENT1 has a data member int j. Another PARENT2 has a data member int k. CHILD class which is inherited from PARENT1 and PARENT2.
CHILD class have data member:
int j (one copy of data member PARENT1)
int k (one copy of data member PARENT2)
int i(two copy of data member GRANDPARENT)
This is ambiguity problem. In CHILD class have two copies of Base class. There are two duplicate copies of int i of base class. One copy through PARENT1 and another copy from PARENT2.

This problem is also called as DIAMOND Problem



[Ambiguity in Multipath Inheritance]

Demonstration of ambiguities in multipath inheritance.

Solution:

```
// This program contains an error and will not compile.
#include <iostream>
class base
{
public:
int i;
};
class derived1 : public base
{
public:
int j;
};
class derived2 : public base
{
public:
int k;
};
class derived3 : public derived1, public derived2
{
public:
int sum;
};
void main()
{
derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too
cout << ob.i << " "; // also ambiguous, which i?
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
}
```

There are two ways to remedy the preceding program. The first is to apply the **scope resolution operator** to i and manually select one i. The second is to use **virtual base class**.

Remove Ambiguities using scope resolution operator:

In above Program , the ambiguous statements

ob.i = 10;

ob.sum = ob.i + ob.j + ob.k;

cout << ob.i << " "; will be replaced by

ob.derived1::i = 10;

ob.sum = ob. derived1::i + ob.j + ob.k;

cout << ob. derived1::i << " "; respectively

As we can see, because the :: was applied, the program has manually selected derived1's version of base. However, this solution raises a deeper issue: What if only one copy of base is actually required? Is there some way to prevent two copies from being included in derived3? The answer, as you probably have guessed, is yes. This solution is achieved using virtual base classes.

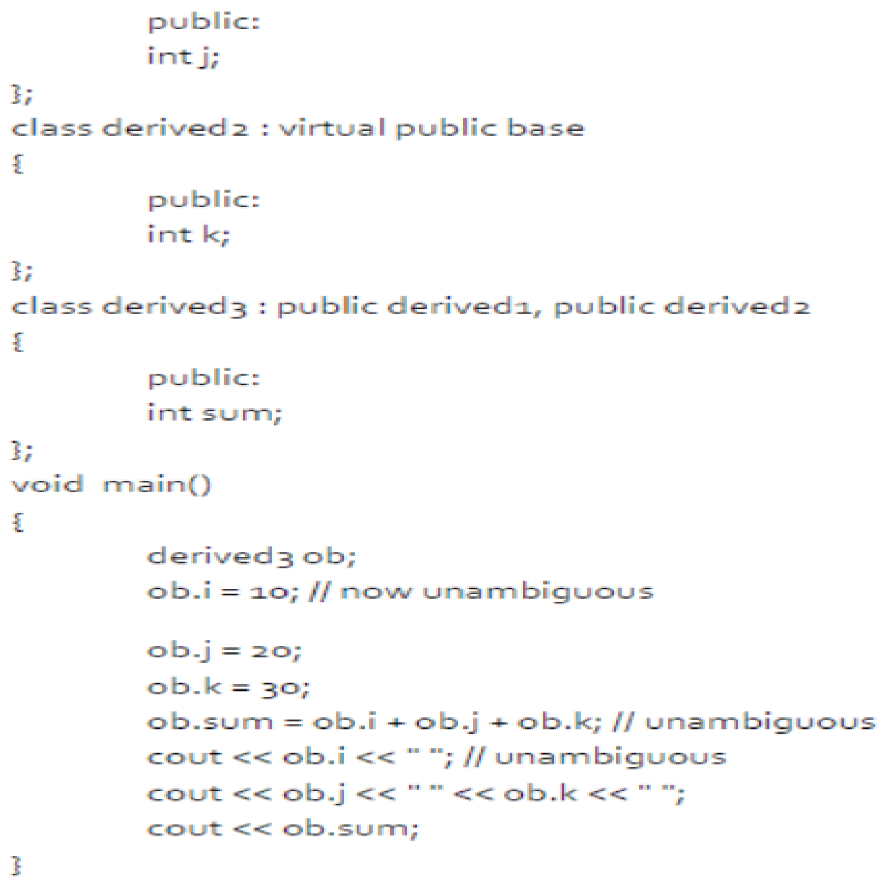
Remove Ambiguities using virtual base class:

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as virtual when it is inherited. We accomplish this by preceding the base class' name with the **keyword virtual** when it is inherited. For example, here is another version of the example program in which derived3 contains only one copy of base:

Remove Ambiguities using virtual base class.

Solution:

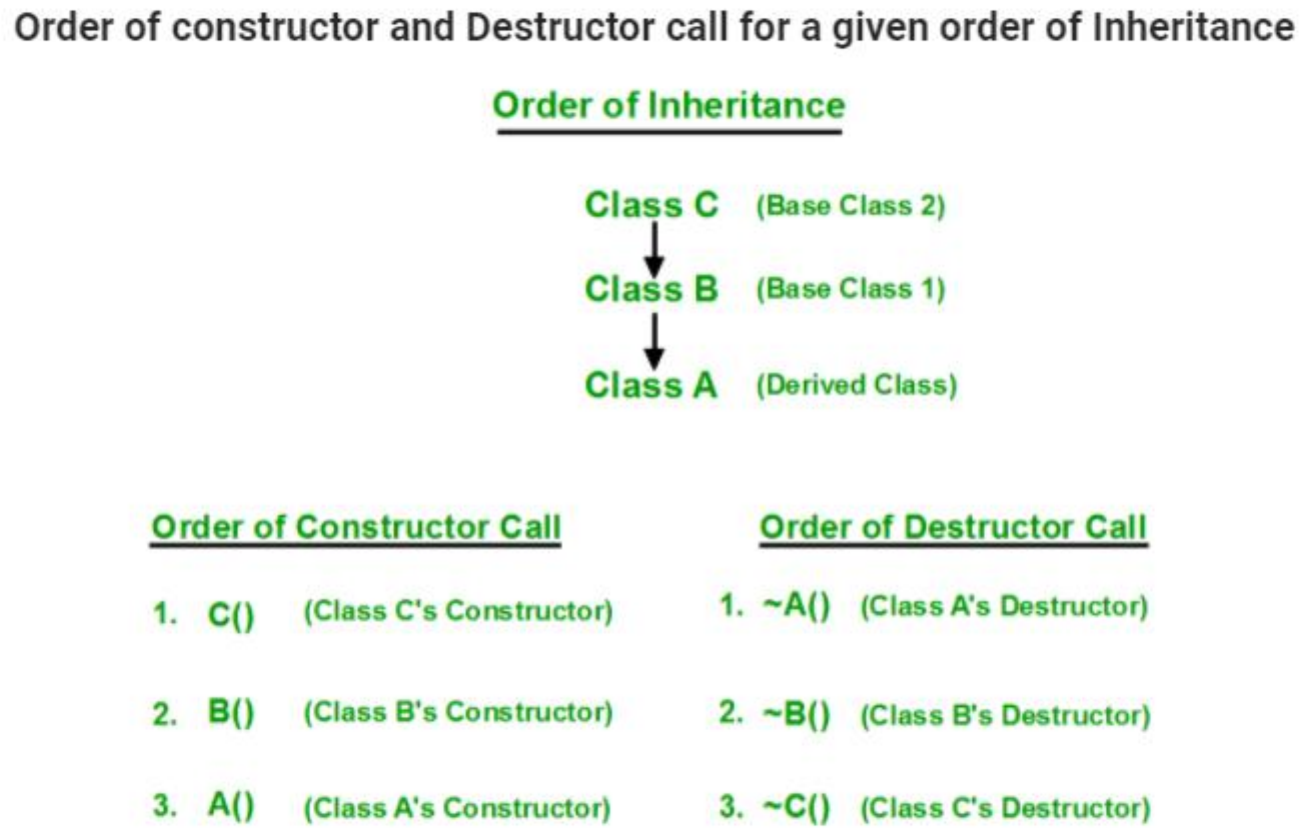
```
#include <iostream>
class base
{
public:
int i;
};
class derived1 : virtual public base
{
```



```
          public:
          int j;
};
class derived2 : virtual public base
{
          public:
          int k;
};
class derived3 : public derived1, public derived2
{
          public:
          int sum;
};
void  main()
{
          derived3 ob;
          ob.i = 10; // now unambiguous

          ob.j = 20;
          ob.k = 30;
          ob.sum = ob.i + ob.j + ob.k; // unambiguous
          cout << ob.i << " "; // unambiguous
          cout << ob.j << " " << ob.k << " ";
          cout << ob.sum;
}
```

Constructor and Destructor Execution in Inheritance

When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class destructor, if it exists (i.e. constructor functions are executed in their order of derivation. Destructor functions are executed in reverse order of derivation).


Order of constructor and Destructor call for a given order of Inheritance
Order of Inheritance
Class C (Base Class 2)
Class B (Base Class 1)
Class A (Derived Class)
Order of Constructor Call
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)
Order of Destructor Call
1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

How to call the parameterized constructor of base class in derived class constructor?

To call the parameterized constructor of base class when derived class’s parameterized constructor is called, you have to explicitly specify the base class’s parameterized constructor in derived class as shown in below program:

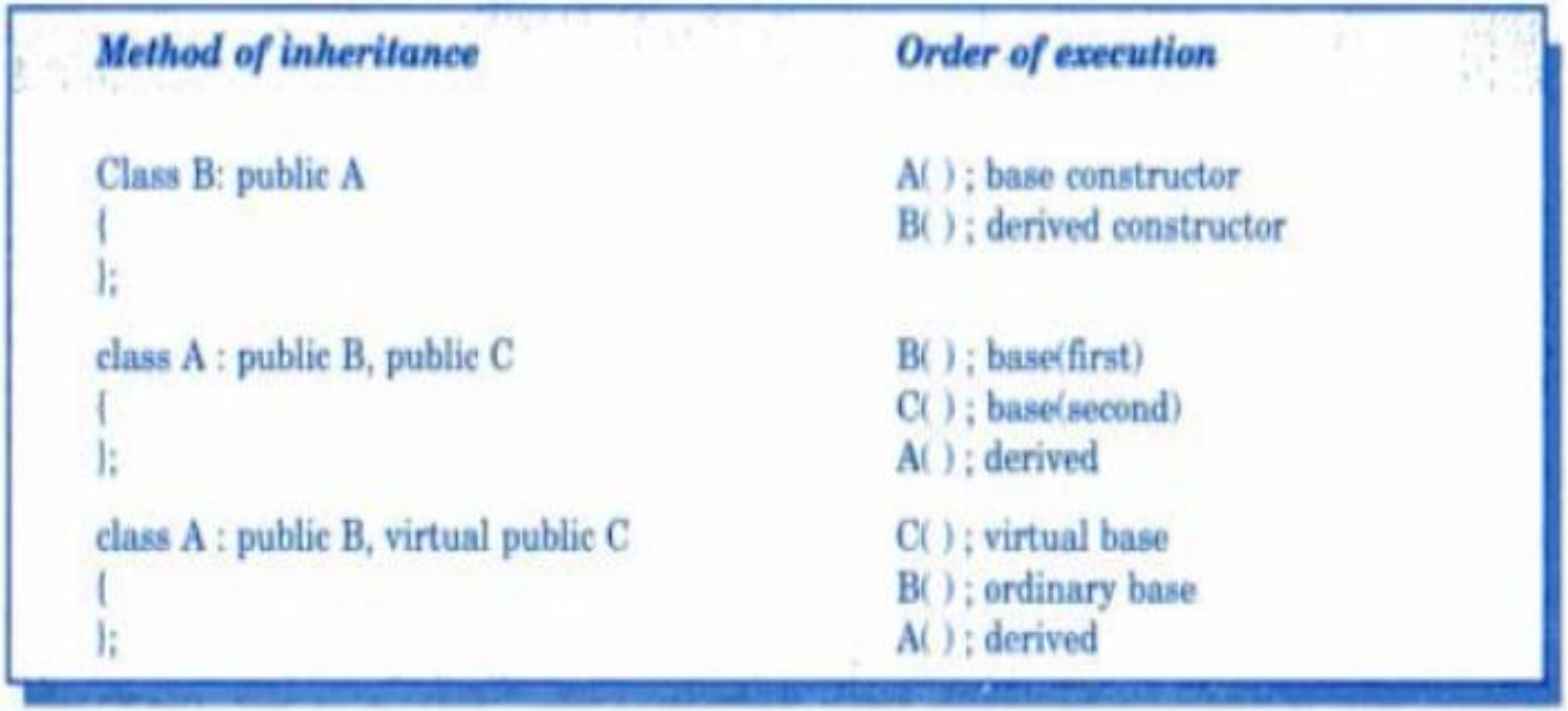
// C++ program to show how to call parameterized Constructor of base class when derived class's Constructor is called

```
#include <iostream>
using namespace std;

// base class
class Parent
{
public:
            // base class's parameterized constructor
            Parent(int i)
            {
            int x =i;
            cout << "Inside base class's parameterized
            constructor" << endl;
            }
};
```

```
// sub class
class Child : public Parent
{
  public:
 // sub class's parameterized constructor
            Child(int j): Parent(j)
            {
            cout << "Inside sub class's parameterized
                        constructor" << endl;
            }
};

// main function
int main() {
            // creating object of class Child
            Child obj1(10);
            return 0;
}
```

Execution of base class constructors

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