- **Inheritance**

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. It is a feature that enables a class to acquire properties and characteristics of another class. Inheritance allows you to reuse your code since the derived class or the child class can reuse the members of the base class by inheriting them. Consider a real-life example to clearly understand the concept of inheritance. A child inherits some properties from his/her parents, such as the ability to speak, walk, and eat, and so on. But these properties are not especially inherited in his parents only. His parents inherit these properties from another class called mammals. This mammal class again derives these characteristics from the animal class. Inheritance works in the same manner. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. There are mainly five types of Inheritance in C++ that you will explore in this article. They are as follows:

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

- **Child and Parent classes :**

To understand the concept of Inheritance, two terms on which the whole concept of inheritance is based - Child class and Parent class.

**Child class:** The class that inherits the characteristics of another class is known as the child class or derived class. The number of child classes that can be inherited from a single parent class is based upon the type of inheritance. A child class will access the data members of the parent class according to the visibility mode specified during the declaration of the child class.

**Parent class:** The class from which the child class inherits its properties is called the parent class or base class. A single parent class can derive multiple child classes (Hierarchical Inheritance) or multiple parent classes can inherit a single base class (Multiple Inheritance). This depends on the different types of inheritance in C++.

The syntax for defining the child class and parent class in all types of Inheritance in C++ is given below:

```
class parent_class
{
   //class definition of the parent class
};
class child_class : visibility_mode parent_class
{
  //class definition of the child class
};
```

**Syntax Description**

- **parent_class:** Name of the base class or the parent class.
- **child_class:** Name of the derived class or the child class.
- **visibility_mode:** Type of the visibility mode (i.e., private, protected, and public) that specifies how the data members of the child class inherit from the parent class.

➢ **Uses of Inheritance:-**

Inheritance makes the programming more efficient and is used because of the benefits it provides. The most important usages of inheritance are discussed below:

- **Code reusability:** One of the main reasons to use inheritance is that you can reuse the code. For example, consider a group of animals as separate classes - Tiger, Lion, and Panther. For these classes, you can create member functions like the predator () as they all are predators, canine() as they all have canine teeth to hunt, and claws() as all the three animals have big and sharp claws. Now, since all the three functions are the same for these classes, making separate functions for all of them will cause data redundancy and can increase the chances of error. So instead of this, you can use inheritance here. You can create a base class named carnivores and add these functions to it and inherit these functions to the tiger, lion, and panther classes.
- **Transitive nature:** Inheritance is also used because of its transitive nature. For example, you have a derived class mammal that inherits its properties from the base class animal. Now, because of the transitive nature of the inheritance, all the child classes of ‘mammal’ will inherit the properties of the class ‘animal’ as well. This helps in debugging to a great extent. You can remove the bugs from your base class and all the inherited classes will automatically get debugged.

## ➢ Modes of inheritance:

There are three modes of inheritance:

- Public mode
- Protected mode
- Private mode

### ▪ Public mode:

In the public mode of inheritance, when a child class is derived from the base or parent class, then the public member of the base class or parent class will become public in the child class also, in the same way, the protected member of the base class becomes protected in the child class, and private members of the base class are not accessible in the derived class.

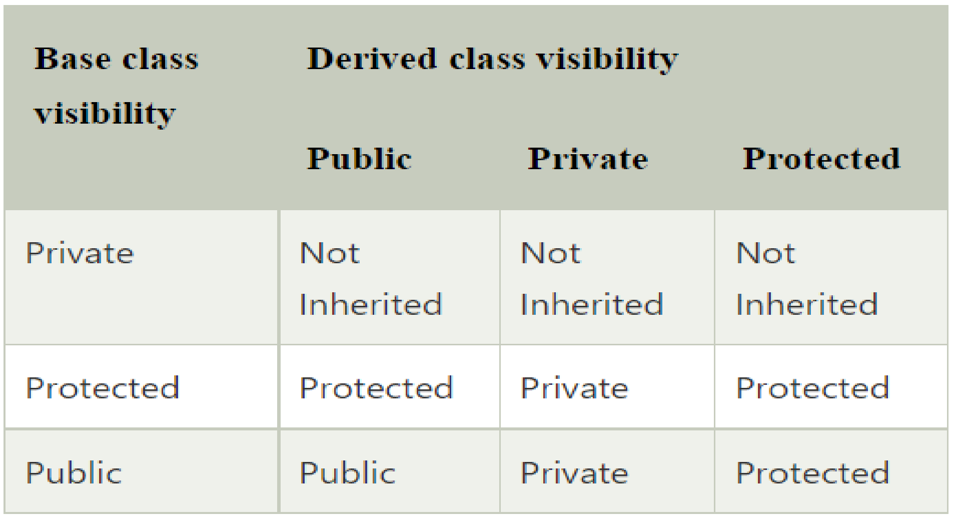
### ▪ Protected mode:

In protected mode, when a child class is derived from a base class or parent class, then both public and protected members of the base class will become protected in the derived class, and private members of the base class are again not accessible in the derived class. In contrast, protected members can be easily accessed in the derived class.

### ▪ Private mode:

In private mode, when a child class is derived from a base class, then both public and protected members of the base class will become private in the derived class, and private members of the base class are again not accessible in the derived class.

Given table will summarize the above three modes of inheritance and show the Base class member access specifier when derived in all three modes of inheritance in C++:

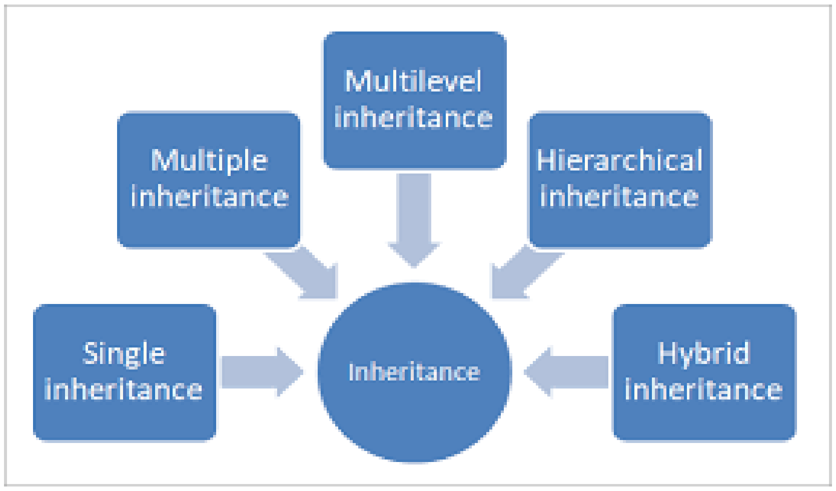


| **Base class visibility** | **Derived class visibility** | | |
|---|---|---|---|
| | **Public** | **Private** | **Protected** |
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

➢ **Types of inheritance:**

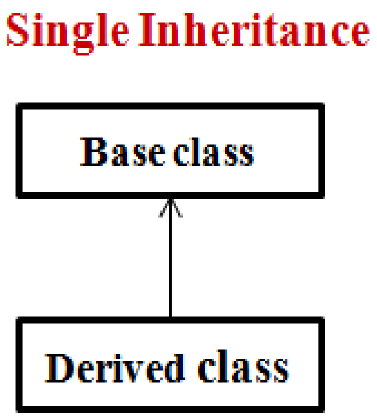
There are five types of inheritance:

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



- **Single Inheritance:**

When the derived class inherits only one base class, it is known as Single Inheritance.



In the above diagram, there is a Base class, and a derived class. Here the child class inherits only one parent class.

**Syntax:**

```
class subclass_name : access_mode base_class
{
  // body of subclass
};

OR

class A
{
... .. ...
};

class B: public A
{
... .. ...
};
```

**Example of Single Inheritance:**

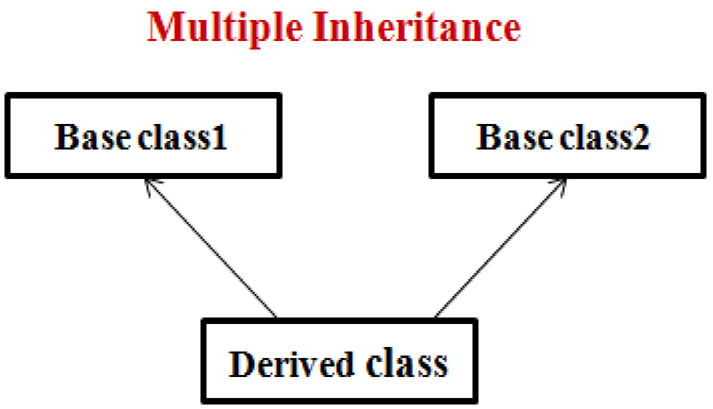
```
#include<iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
    }
};

// sub class derived from a single base classes
class Car : public Vehicle {

};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

**Output:**

```
This is a Vehicle
```

- **Multiple Inheritance:**

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.



**Syntax:**

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
```

```
{
... ... ...
};
```

**Example:**

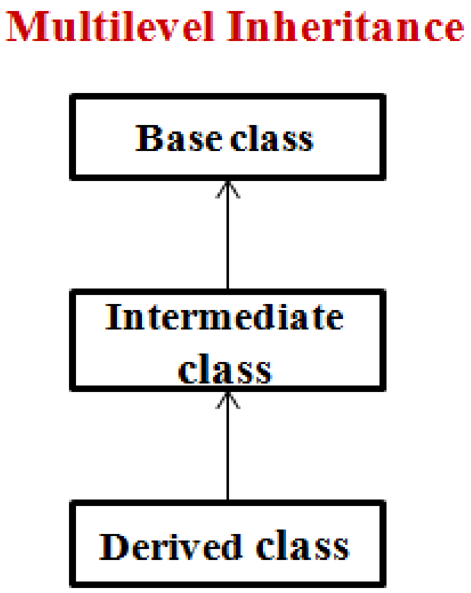
```
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
   FourWheeler()
   {
      cout << "This is a 4 wheeler Vehicle\n";
   }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

**Output :**

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

- **Multilevel Inheritance:**

In this type of inheritance, a derived class is created from another derived class.



**Syntax:-**

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

**Example:**

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

// base class
class Vehicle {
```

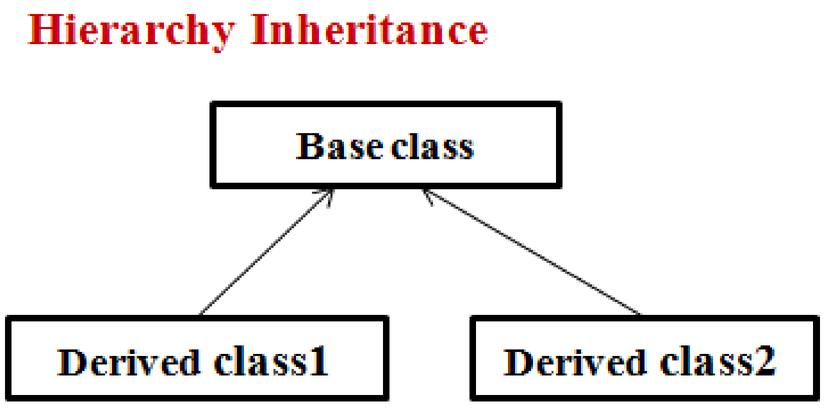
```
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
   fourWheeler()
   {
      cout << "Objects with 4 wheels are vehicles\n";
   }
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
   Car() { cout << "Car has 4 Wheels\n"; }
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

**Output:**

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

- **Hierarchical Inheritance:**

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



**Syntax:-**

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

**Example:**

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

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
```

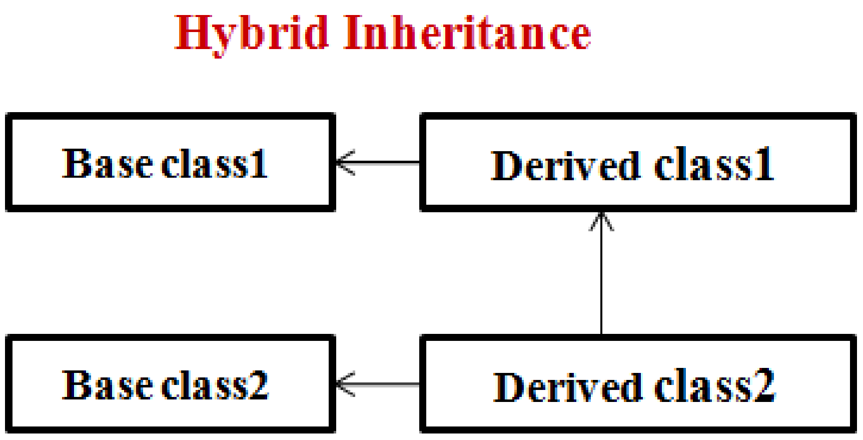
```
// first sub class
class Car : public Vehicle {
};

// 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 (Virtual) Inheritance:**

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.



**Example:**

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

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// 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
```

- **Virtual function**

  - A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
  - It is used to tell the compiler to perform dynamic linkage or late binding on the function.
  - There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
  - A 'virtual' is a keyword preceding the normal declaration of a function.
  - When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

By default, C++ matches a function call with the correct function definition at compile time. This is called static binding. You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

If we have created a virtual function in the base class and it is being overridden in the derived class then we don’t need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

- **Rules of Virtual Function:**

  - Virtual functions must be members of some class.
  - Virtual functions cannot be static members.
  - They are accessed through object pointers.
  - They can be a friend of another class.
  - A virtual function must be defined in the base class, even though it is not used.
  - The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
  - We cannot have a virtual constructor, but we can have a virtual destructor
  - Consider the situation when we don't use the virtual keyword.

```
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base {
```

```
public:
    virtual void print()
    {
        cout << "print base class\n";
    }

    void show()
    {
        cout << "show base class\n";
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class\n";
    }
void show()
    {
        cout << "show derived class\n";
    }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

**Output:**

```
print derived class
show base class
```

**Example:**

```
// working of Virtual Functions
#include<iostream>
using namespace std;

class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};
int main()
{
   base *p;
   derived obj1;
   p = &obj1;

   // Early binding because fun1() is non-virtual
   // in base
   p->fun_1();

   // Late binding (RTP)
   p->fun_2();

   // Late binding (RTP)
   p->fun_3();
```

```
        // Late binding (RTP)
        p->fun_4();

        // Early binding but this function call is
        // illegal (produces error) because pointer
        // is of base type and function is of
        // derived class
        // p->fun_4(5);
            return 0;
            }
```
**Output:**

```
    base-1
    derived-2
    base-3
    base-4
```

Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of Late and Early Binding is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

fun_4(int) in derived class is different from virtual function fun_4() in base class as prototypes of both the functions are different.

- **Limitations of Virtual Functions:**

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.

- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

- **Pure Virtual Function**

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

**virtual void display() = 0;**

**Example:**

```
#include<iostream>
using namespace std;
class B {
  public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
  public:
    void s() {
      cout << "Virtual Function in Derived class\n";
    }
};

int main() {
  B *b;
  D dobj;
  b = &dobj;
  b->s();
}
```

**Output:**

```
Virtual Function in Derived class
```