INHERITANCE

1 Introduction

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. Inheritance is the process of creating new classes, called derived classes, from existing or *base classes*. The mechanism of deriving a new class from an old one is called ***inheritance (or derivation).***

The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure 1.

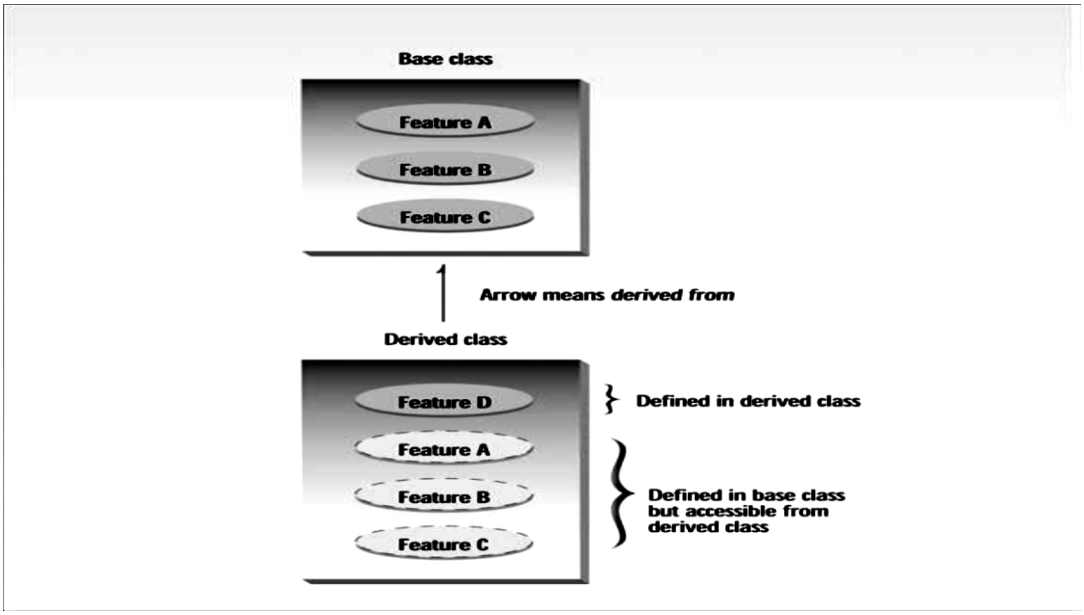


FIGURE (1) ***Inheritance.***

Inheritance is an essential part of OOP. Reusing existing code saves time and money and increases a program's reliability. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

2 Defining Derived Classes

A derived class is 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*** (access specifier) is private. Visibility mode specifies whether the features of the base class are ***private derived*** **or** ***public derived.***

3. Access Specifier

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 'public members' of the base class can only be accessed by member functions of the derived class. On other hand, when a base class is ***publicly inherited*** 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derive 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 derive class.

Example:-

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

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

```
class ABC :  XYZ    // private derivation by default
 {
        members of ABC
 };
```

A private member of a base class cannot be inherited and therefore it is not available for the derived class directly. To solve this problem, 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 eliminate 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
  ……                  // visibile to member functions
  ……..                // within its class
  protected :
   …….                 // visibile to member functions
   ……..                // of its own and derived class
  public:
  ……..                // visible to all functions
   …….                 // in the program
   };
```

Table 1: Inheritance and Accessibility

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside class
Public	Yes	yes	yes
Protected	Yes	yes	No
private	Yes	No	No

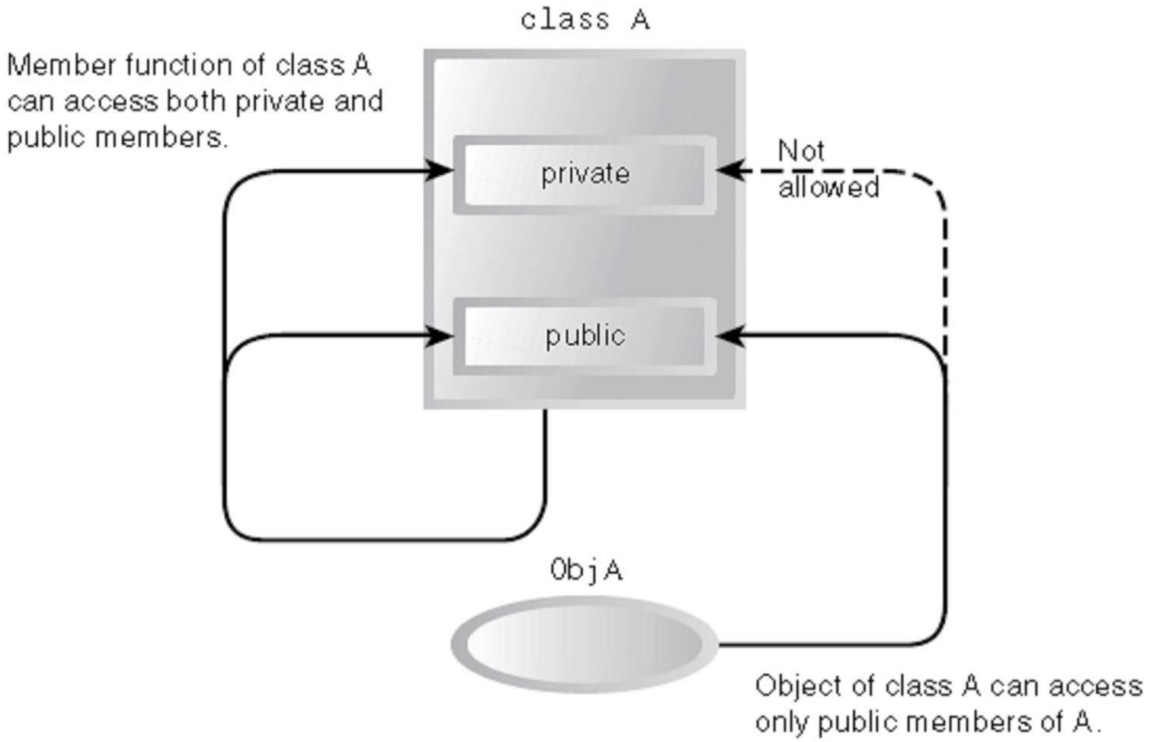


Figure 2: Access *specifiers without inheritance*

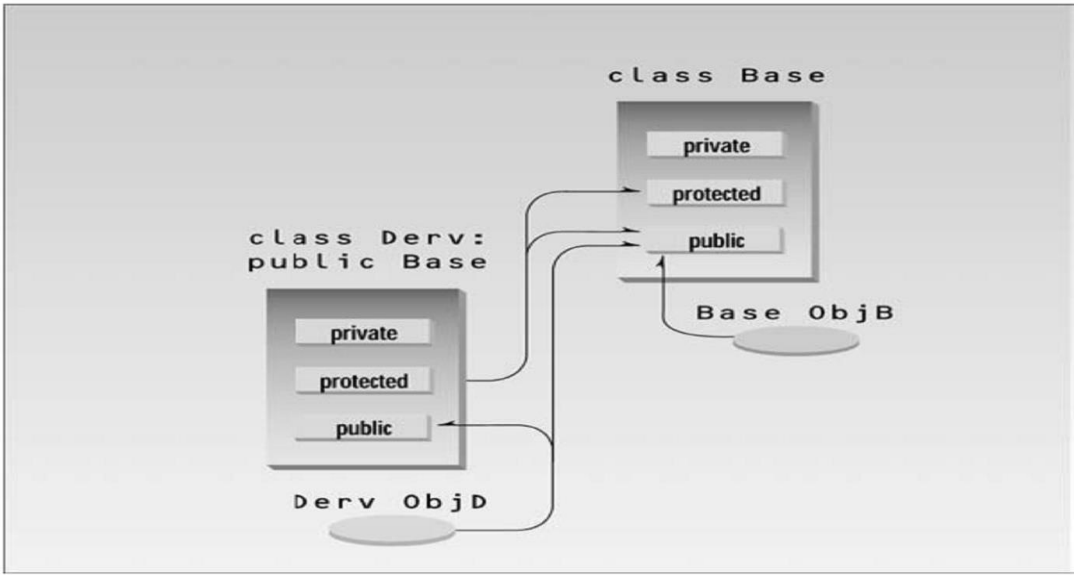


Figure 3: *Access specifiers with inheritance*

We can use inheritance to create a new class based on Counter, without modifying counter itself. A new class, CountDn, that adds a decrement operator to the Counter class

Example1:

```
// C++ program to demonstrate implementation
// of Inheritance

#include <bits/stdc++.h>
using namespace std;

// Base class
class Parent
{
  public:
    int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
public:
    int id_c;
};

// main function
int main()
{
    Child obj1;

    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " <<  obj1.id_c << '\n';
    cout << "Parent id is: " <<  obj1.id_p << '\n';

    return 0;
}
```

Output

```
Child id is 7

Parent id is 91
```

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

Example1:

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

Example:-

```
#include <iostream>
using namespace std;
class B
{
        int a;     //private;  not  inheritable
   public:
         int b;    //public ready for inheritance
         void  get_ab( );
         int     get_a(void);
         void  show_a(void);
};
class D : public B   //public derivation
{
int c;
   public:
         void  mul (void);
         void  display (void);
};
//  function defined //
void B :: get_ab(void)
{
a  =  5;     b  =  10;
    }
int  B :: get_a( )
{
```

```
 return  a;
  }
void B :: show_a( )
{
cout  <<  "a  =  " <<  a  <<  "\n";
 }
void  D ::  mul ( )
{
c  =  b *  get_a ( );
 }
void  D ::  display (void)
{
    cout  <<   " a=  " <<  get_a( )  <<  "\n";
    cout  <<   " b=  " <<  b  <<  "\n";
    cout  <<   " c=  " <<  c  <<  "\n";
}
//  main program //
int main()
{     D d;
   d.get_ab();
   d.mul();
   d.show_a();
   d.display();
   d.b=20;
   d.mul();
   d.display();
return 0;
}
```

The output of program:

a=5

a=5

b=10

c=50

a=5

b=20

c=100

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

Example2:-

```
using namespace std;
class B
{
        int a;                  //private;  not  inheritable
   public:
          int b;              //public ready for inheritance
          void  get_ab( );
          int     get_a(void);
          void  show_a(void);
};
class D : private B                  // private derivation
{       int c;
   public:
          void  mul (void);
          void  display (void);
};
//  function defined //
void B :: get_ab(void)
{
 cout<<"Enter the values for  a and b:";
  cin>>a>>b;
}
```

```
int  B :: get_a( )
{
return  a;
  }
void B :: show_a( )
{
cout  <<  "a  =  " <<  a  <<  "\n";
  }
void  D ::  mul ( )
{
      get_ab();
      c  =  b *  get_a ( );            // 'a' cannot be used directly
}
void  D ::  display (void)
{
     show_a();
      cout  <<   " b=  " <<  b  <<  "\n";
     cout  <<   " c=  " <<  c  <<  "\n";
}
```

// main program //

```
int main()
{
     D d;
    d.mul();
    d.display();
    d.b=20;   // won't work; b has become private
    d.mul();
    d.display();
return 0;
```

The output of program:

Enter the values for a and b:5 10

a=5

b=10

c=50

Enter the values for a and b:12 20

a=12

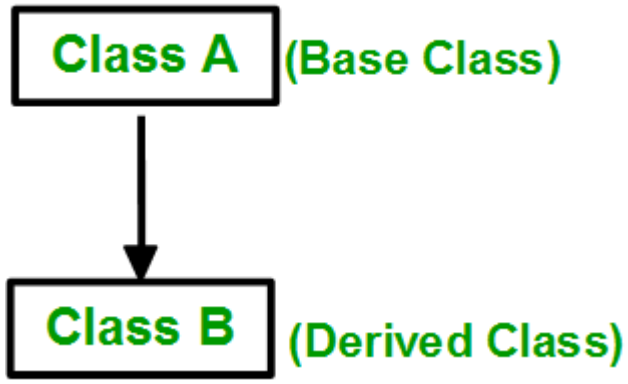
b=20

c=240

}

Types of Inheritance in C++

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

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

Example:

```
// C++ program to explain
// 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 {

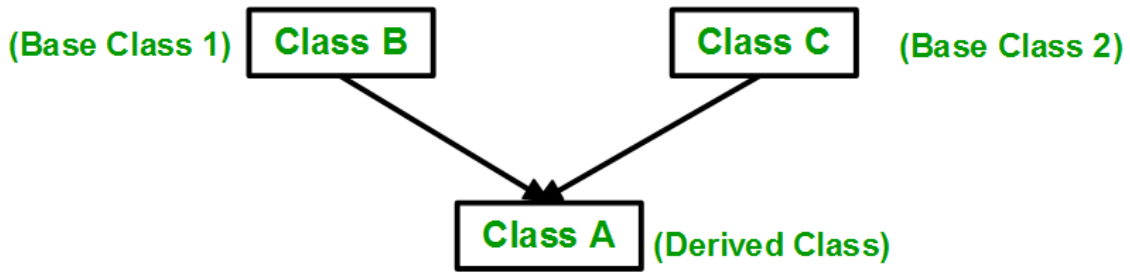
};

// 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

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode
base_class2, ....
{
  // body of subclass
```

```
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#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 {

};

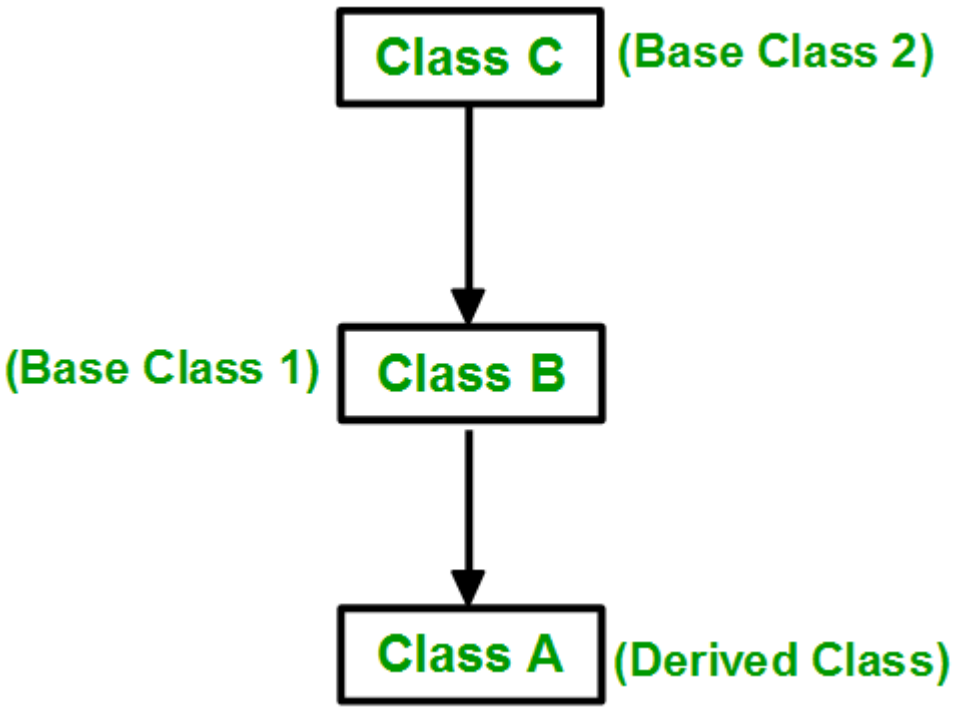
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle

This is a 4 wheeler Vehicle
```

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.




```
/ C++ program to implement
// Multilevel Inheritance
#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;

}
```

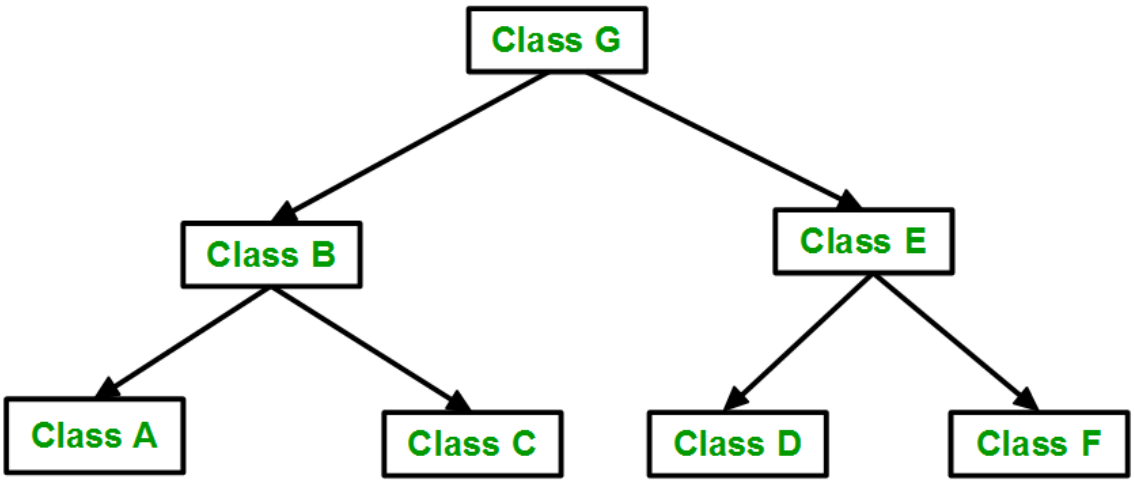
Output

This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

4. Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



// C++ program to implement

```
// Hierarchical Inheritance
#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
{
```

```
};

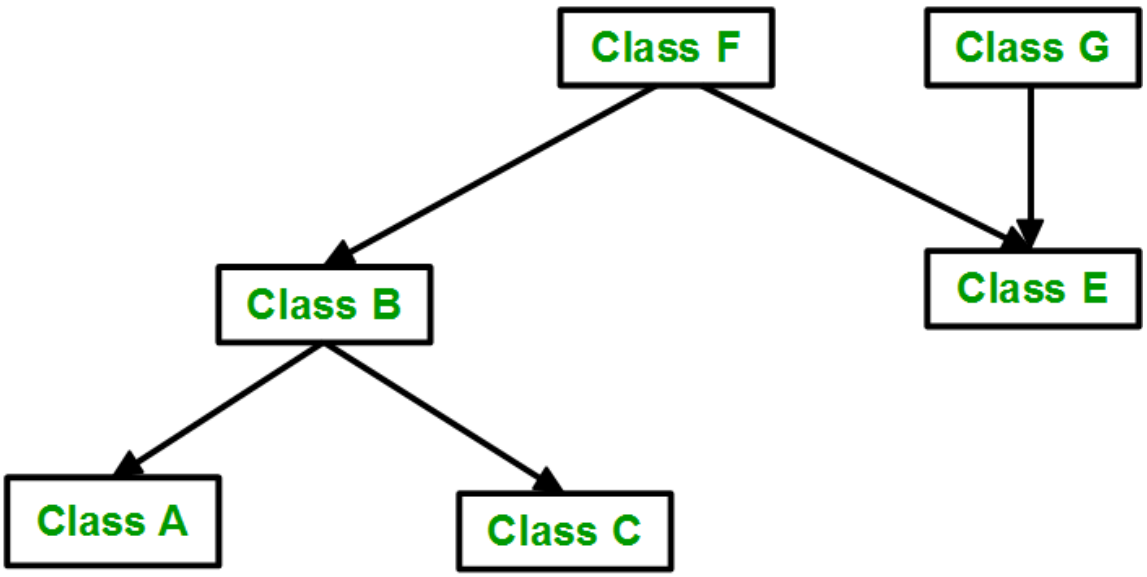
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output

This is a Vehicle

This is a Vehicle

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritance:




```
/ C++ program for Hybrid Inheritance

#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
```

1.Friend Functions

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows –

```
class Box {
  double width;

  public:
    double length;
friend void printWidth( Box box );
    void setWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne –

```
friend class ClassTwo;
```

Consider the following program −

```
#include <iostream>

using namespace std;

class Box {
double width;

  public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};

//  function definition
```

```
void Box::setWidth( double wid ) {
  width = wid;
}

// Note: printWidth() is not a function of any class.
void printWidth( Box box ) {
  /* Because printWidth() is a friend of Box, it can
  directly access any member of this class */
  cout << "Width of box : " << box.width <<endl;
}
// Main function for the program
int main() {
  Box box;

  // set box width without member function
  box.setWidth(10.0);

  // Use friend function to print the wdith.
  printWidth( box );
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Width of box : 10

2.Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

Example1:

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

class printData {
```

```
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};

int main(void) {
  printData pd;

  // Call print to print integer
  pd.print(5);

  // Call print to print float
pd.print(500.263);

  // Call print to print character
  pd.print("Hello C++");

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Example2:

```
// CPP program to illustrate // Function Overloading

#include <iostream>

using namespace std;

// overloaded functions

void test(int);

void test(float);

void test(int, float);

 int main()
{
    int a = 5;
    float b = 5.5;

    // Overloaded functions
    // with different type and
    // number of parameters
    test(a);
    test(b);
    test(a, b);

    return 0;
}

// Method 1
void test(int var)

{

    cout << "Integer number: " << var << endl;

}

 // Method 2

void test(float var)
```

```
{
    cout << "Float number: "<< var << endl;
}
 // Method 3
void test(int var1, float var2)
{
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Output:

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

3. Function Overriding using C++

When the base class and derived class have functions with exactly the same name, same return-type, and same arguments list, then it is said to be function overriding.It is the redefinition of base class function in its derived class with same signature i.e return type and parameters.

- It can only be done in derived class.

Example:

```
Class a
{
public:
    virtual void display(){ cout << "hello"; }
};

Class b:public a
{
public:
     void display(){ cout << "bye";}
};
```

Example1:

```
// CPP program to illustrate
// Function Overriding
#include<iostream>
using namespace std;

class BaseClass
{
public:
   virtual void Display()
   {
      cout << "\nThis is Display() method"
           " of BaseClass";
   }
   void Show()
```

```
    {
        cout << "\nThis is Show() method "
             "of BaseClass";
    }
};
class DerivedClass : public BaseClass
{
public:
    // Overriding method - new working of
    // base class's display method
    void Display()
    {
        cout << "\nThis is Display() method"
             " of DerivedClass";
    }
};

// Driver code
int main()
{
    DerivedClass dr;
    BaseClass &bs = dr;
    bs.Display();
    dr.Show();
}
```

Output:

This is Display() method of DerivedClass

This is Show() method of BaseClass

Example2:

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

```
class A {
  public:
  void display() {
    cout<<"Base class";
  }
};

class B:public A {
  public:
  void display() {
    cout<<"Derived Class";
  }
};

int main() {
  B obj;
  obj.display();
  return 0;
}
```

It will produce the following output : Derived Class

Function Overloading VS Function Overriding:

1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
2. **Function Signature:** Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in same scope.
4. **Behavior of functions:** Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.