Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

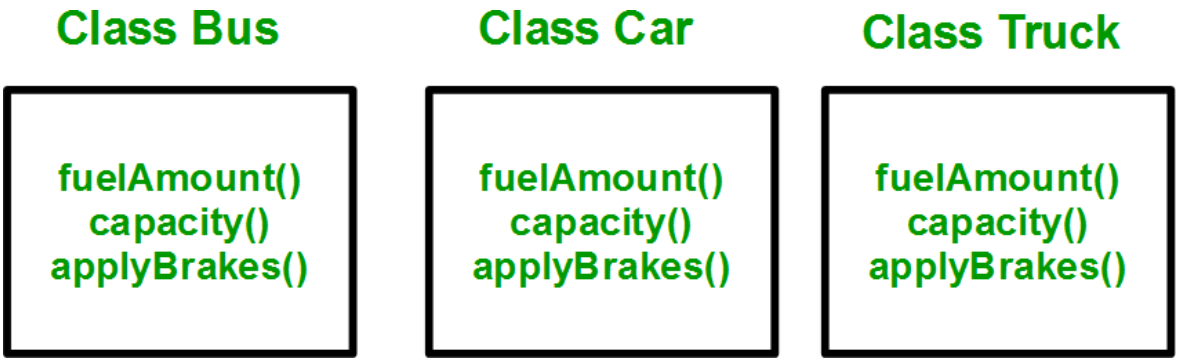
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

The article is divided into following subtopics:

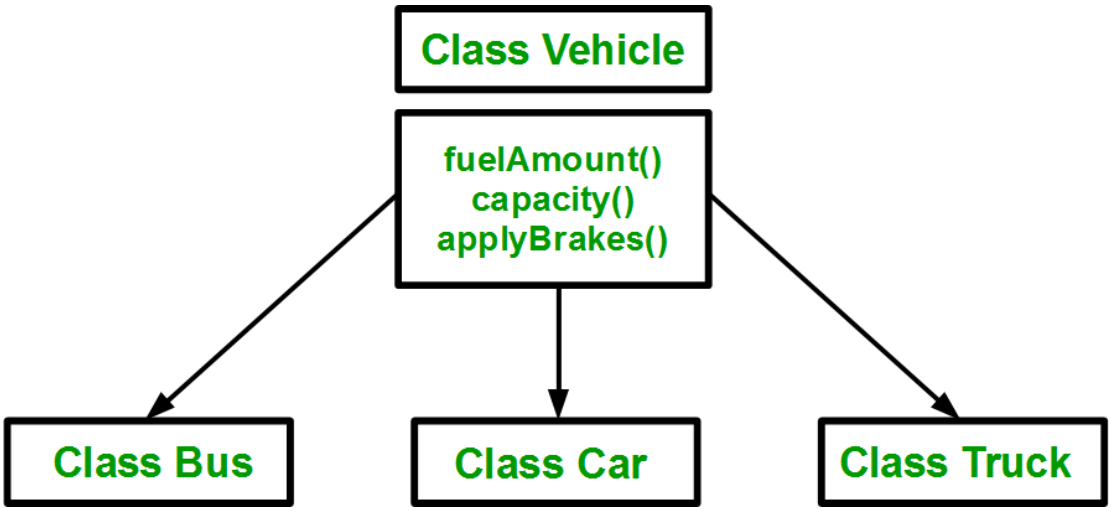
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).
Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax. **Syntax**:

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

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// 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 << endl;
      cout << "Parent id is " << obj1.id_p << endl;

      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'.

Modes of Inheritance

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

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.

```
// 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;
};
```

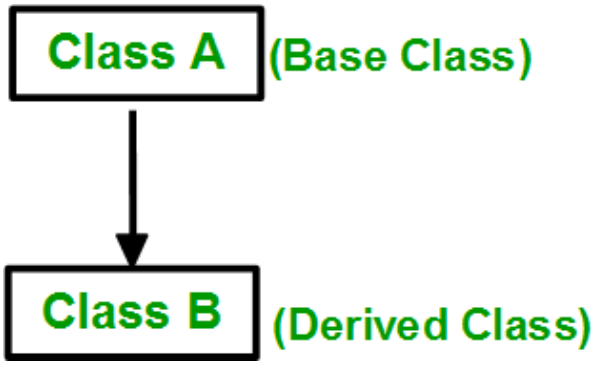
```
class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

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
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

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)

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:

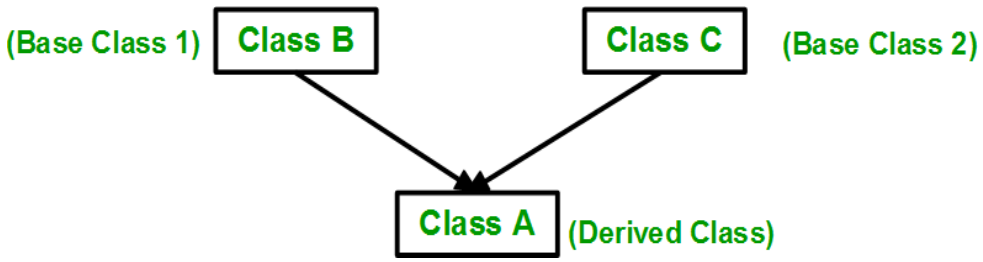
```
class subclass_name : access_mode base_class

{

//body of subclass

};
```

```
// C++ program to explain
// Single inheritance  #include <iostream>
using namespace std;

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

// sub class derived from two 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 class**



Syntax:

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

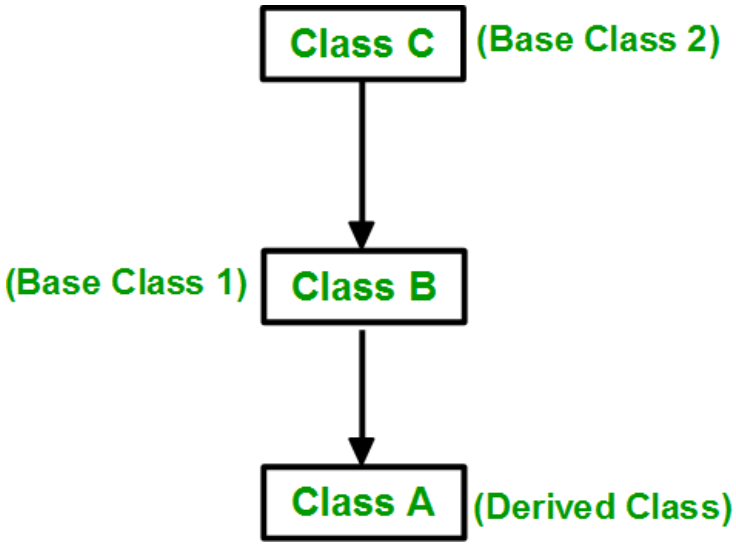
Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance  #include <iostream>  using namespace std;

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

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

// 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.

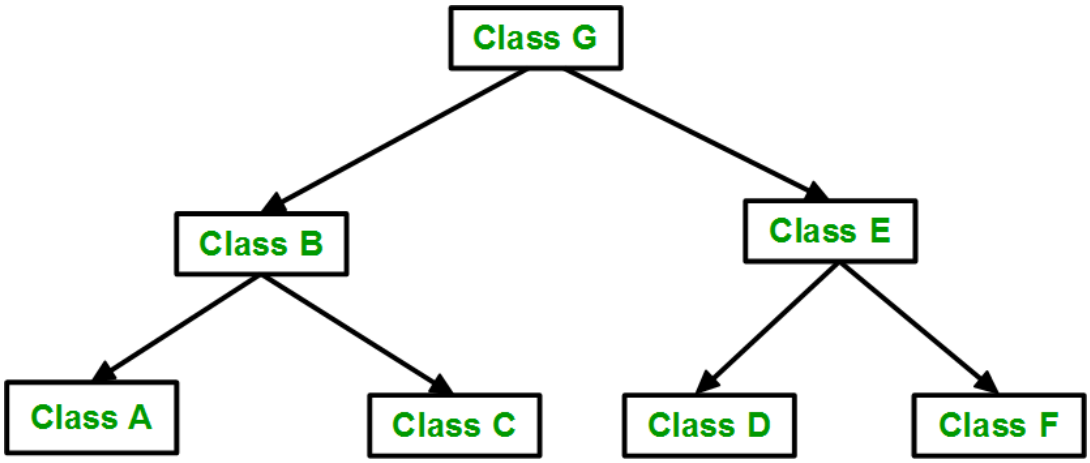



```
// C++ program to implement
// Multilevel Inheritance  #include <iostream>  using
namespace std;

// base class  class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{ public:  fourWheeler()
{
  cout<<"Objects with 4 wheels are vehicles"<<endl;
}
};
// sub class derived from two base classes  class Car:
  public fourWheeler{
 public:
  car()
  {
    cout<<"Car has 4 Wheels"<<endl;
  }
```

```
};

// 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.



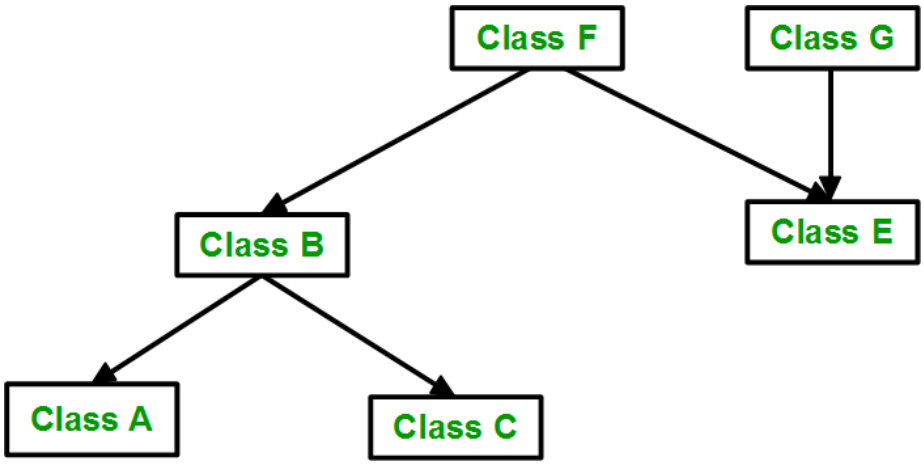

```
// C++ program to implement
// Hierarchical Inheritance  #include
<iostream>  using namespace std;

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

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:



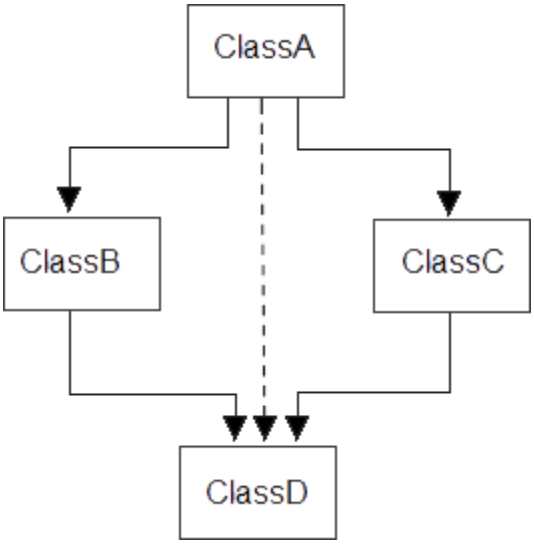

```
// C++ program for Hybrid Inheritance

#include <iostream>  using namespace
std;

// base class  class Vehicle
{
```

```
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << 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
```

A special case of hybrid inheritance : Multipath inheritance:
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



Consider the following program:
// C++ program demonstrating ambiguity in Multipath Inheritance

```
#include<iostream.h>  #include<conio.h>  class ClassA
    {
        public:
        int a;
    };

    class ClassB : public ClassA
    {
        public:
        int b;
    };
    class ClassC : public ClassA
    {
        public:
        int c;
    };

    class ClassD : public ClassB, public ClassC
    {
        public:  int d;
    };

    void main()
    {

        ClassD obj;



        //obj.a = 10;              //Statement 1, Error
        //obj.a = 100;             //Statement 2, Error
```

```
        obj.ClassB::a = 10;       //Statement 3
        obj.ClassC::a = 100;      //Statement 4

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A from ClassB : "<< obj.ClassB::a;
        cout<< "\n A from ClassC : "<< obj.ClassC::a;

        cout<< "\n B : "<< obj.b;  cout<< "\n C : "<<
        obj.c;  cout<< "\n D : "<< obj.d;

    }
```

Output:

```
A from ClassB : 10  A from ClassC : 100  B : 20
C : 30
D : 40
```

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.
If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

1. **Use scope resolution operator**
2. **Use virtual base class**

Avoiding ambiguity using scope resolution operator:
Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.
filter_none
edit play_arrow brightness_4

```
obj.ClassB::a = 10;       //Statement 3
obj.ClassC::a = 100;      //Statement 4
```

Note : Still, there are two copies of ClassA in ClassD.

Avoiding ambiguity 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 3
            obj.a = 100;      //Statement 4

            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
```

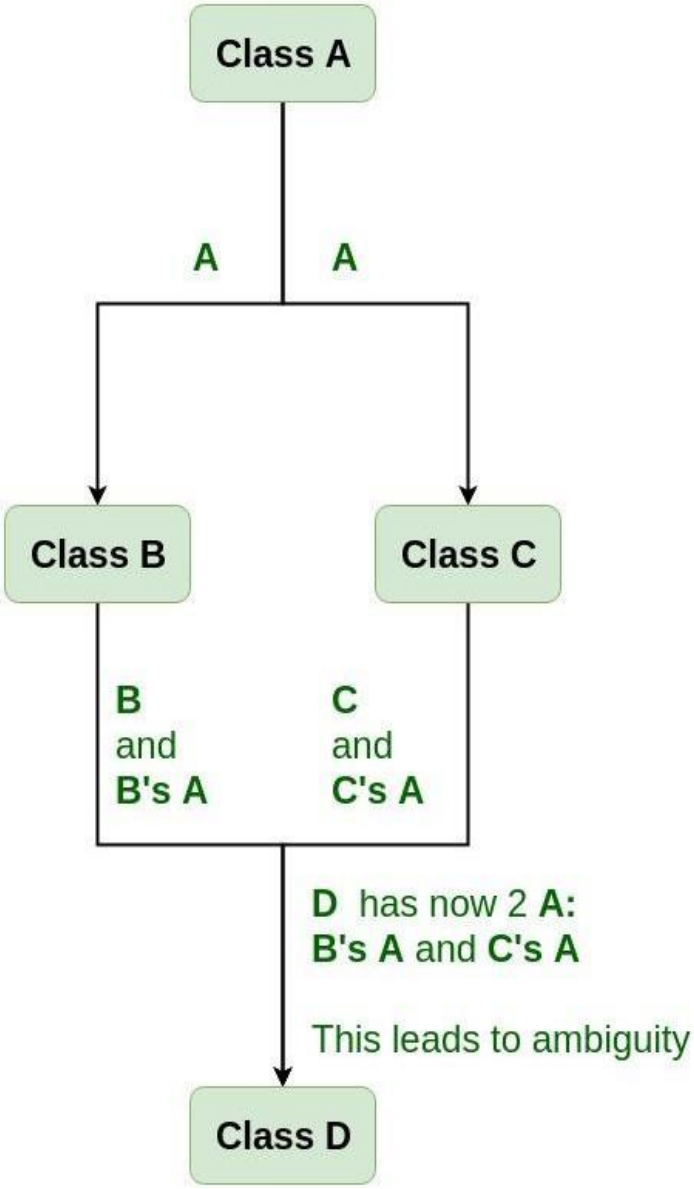
According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:

Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function

member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.
Example: To show the need of Virtual Base Class in C++ #include <iostream>
using namespace std;

```
class A {  public:
   void show()
   {
      cout << "Hello form A \n";
   }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
   D object;  object.show();
}
```

Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for member 'show' is ambiguous  object.show();
      ^
prog.cpp:8:8: note: candidates are: void A::show()  void show()
    ^
prog.cpp:8:8: note:                 void A::show()
```

How to resolve this issue?
To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:
Syntax 1:

```
class B : virtual public A
```

```
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Note: virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

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

class A {  public:
   int a;
   A() // constructor
   {
      a = 10;
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
   D object; // object creation of class d  cout << "a = " << object.a << endl;

   return 0;
}
```

Output:

```
a = 10
```

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Example 2:

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

```
class A {  public:
   void show()
   {
      cout << "Hello from A \n";
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
   D object;  object.show();
}
```

Output:

```
Hello from A
```

Inheritance

→ The process of coping the properties from one class to another is called inheritance.

→ The mechanism of deriving or creating a new class from existing old is called inheritance

→ The old class is known as "base" class "super" class or "parent class."

→ The new class is known as "sub" class "derived" class or child class.

=> Reusability: boilding new components by utilizing existing components.

Types of Inheritance

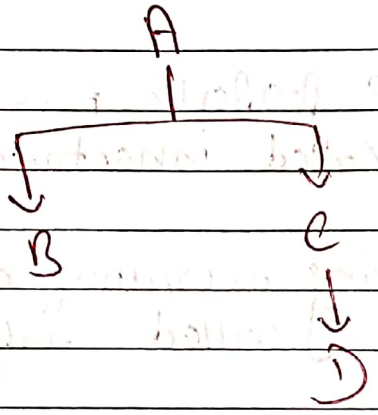
(I) Single Inheritance (one base and one derived class):

(II) Multiple Inheritance (More than one base classes for one derived class).

(III) Hierarchical Inheritance (More than one derived class for one basic class).

(IV) Multilevel Inheritance.

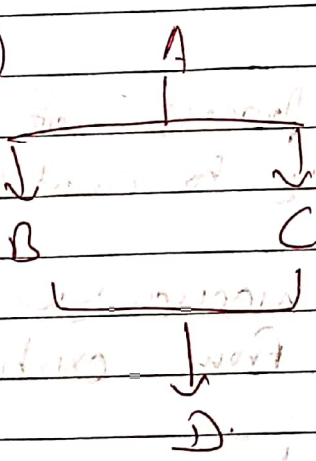
(V) Hybrid Inheritance (Combination of more than one types of inheritance.



(i) Hirarchical.

(ii) A → C → D.

by multilevel.



1) Hirarchical.

2) Multiple.

A → B → D

or

A → C → D

3) Multilevel.

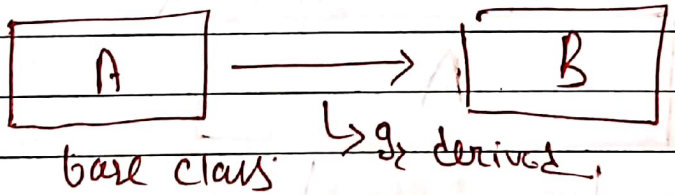
Accessibility area

Specifier / class	Same class	Derived class	Any other class	friend fuction	friend class
Private	Yes	No	No	Yes	Yes
Protected	Yes	Yes	No	Yes	Yes
Public	Yes	Yes	Yes	Yes	Yes

* Summary of modes:

Access in Base class	Base class inherited as	Access in derived class
Public	Public	Public
Protected		Protected
Private		No access
Public	Protected	Protected
Protected		Protected
Private		No access
Public	Private	Private
Protected		Private
Private		No access

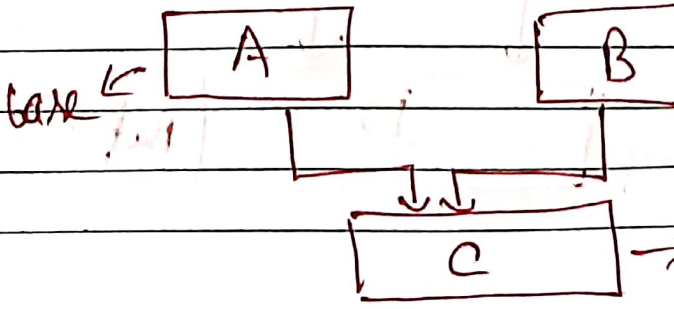
1) Single Inheritance: A derived class with only base class.



↳ is derived.

2) Multiple Inheritance:

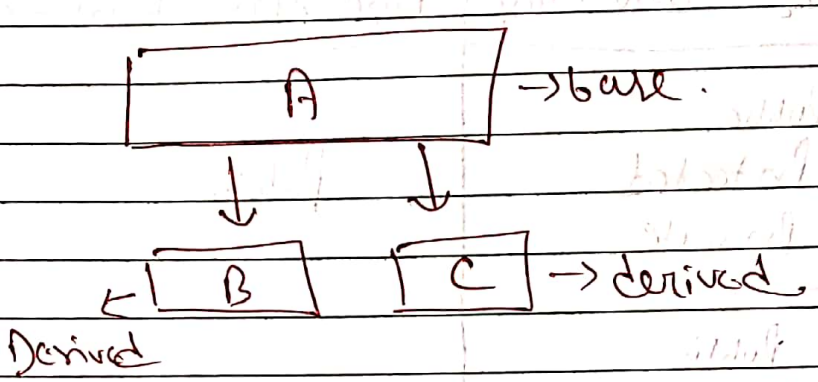
A derived class with more than one base class.



iii) Multilevel Inheritance: one base and two derived class.

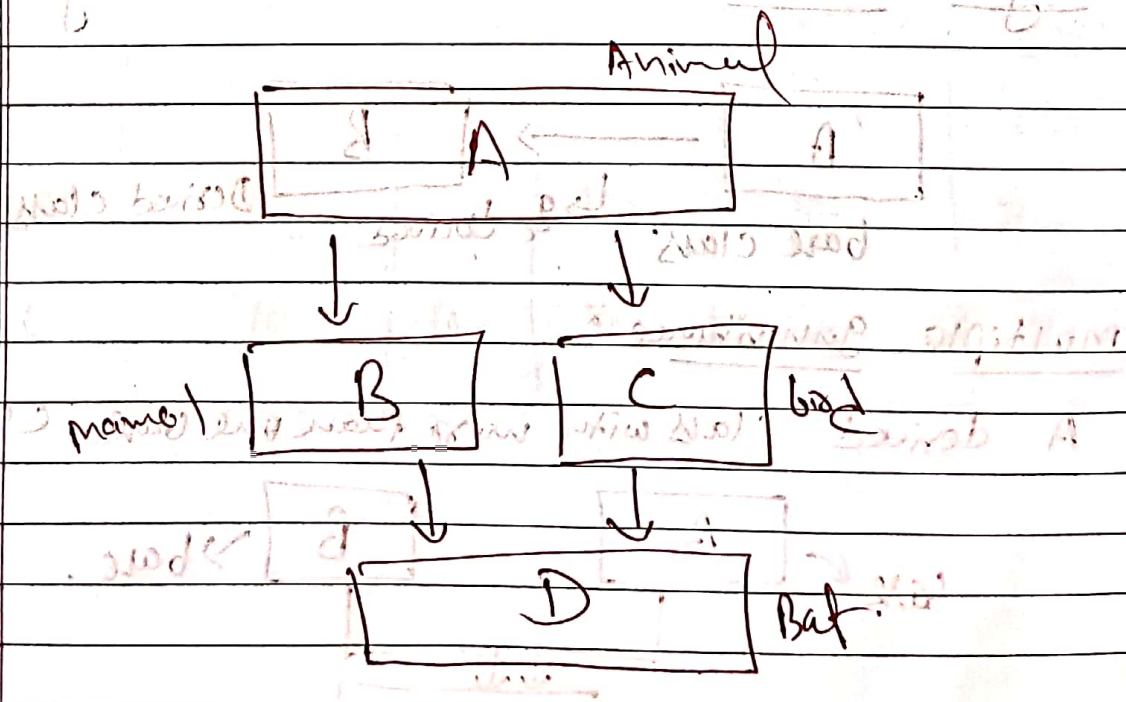
iii) Hierarchical inheritance:

Several derived class from a single base class.



(iv) Hybrid Inheritance:

- Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance.
- A class is derived from two class as in multiple inheritance.
- However, one of the parent classes is not a base or bare class.



Syntax and visibility mode

```
#include <iostream>
using namespace std;
    // base class
class employee
{
  public:
      int id;
      float salary;
  employee (int inpid) {
        id = inpid;
        salary = 34.0;
    }
    Employee() {}
    };
// Derived class syntax

class id derived class name : (visibility mode) base class name
    {
        class member/method etc.
    };
```

Note.:

i) Default visibility mode is private.
ii) Public visibility mode: Public members of the base class becomes public member of derived class.
iii) Private visibility mode: Public member of the base class becomes private members of the derived class.
iv) Private members are never inherited.

// creating a programme class derived from employee base class.

```
class Programmer : public employee
{
public:
    int languageCode;
    Programmer (int inpId)
    {
        id = inpId;
        language code = 9;
    }
void getdata() {
    cout << id << endl;
    }
};

int main ()
{
    employee harry(1), roman(2);

cout << harry.salary << endl;
cout << roman.salary << endl;
    Programmer skill(10);
cout << skill.language << endl;

    cout << skill.id << endl;
    skill.getdata();
    return 0;
}
```

Single Inheritance

```
class base {
  int data1;
  public:
    int data2;
  void setdata();
    int getdata1();
    int getdata2();
};
void base :: setdata(void
  {
      data1 = 10; data2 = 20; }
int base :: getdata1() {
    return data1;
    }
int base :: getdata2() {
    return data2;
  }
class derived : public base
  { int data3;
    public:
  void process();
  void display();
  };
void derived :: process()
  {
    data3 = data2 * getdata1();
  }
void Derived :: display() {
Cout<<" data 1 value " << getdata1() << endl;
Cout<< "data 2 value" << data2 << endl;
Cout << "value of data3" << data3 << endl;
  }
```

```
int main()
{
    Derived der;
    der.setdata();
    der.Process();
    der.display();

    return 0;
}
```

Protected Access Modifier

```
class base {
protected:
    int a;
private:
    int b;
};
```

/*

for a protected member.

	Public derivation	Private derivation	Protected
1. Private members	Not Inherited	Not Inherited	Not Inherited
2. Protected members	Protected	Private	Protected
3. Public members	Public	Private	Protected

*/

```
class Derived : protected base {
};
```

```
int main()
{
    base b;
    derived d;
// cout << d.a; (will not work since a is protected
}                 in both base as well as derived class)
```

Multilevel Inheritance

```
class student
{
    protected:
    int roll_no;
    public:
        void set_roll_number(int);
        void get_roll_number(void);
};

void student :: set_roll_number(int x)
{
    roll_number = x;
}
void student :: get_roll_number()
{
  cout << "the roll no is" << roll_number << endl;
}
class Exam : public student
{
    Protected :
        float Maths;
        float Physics;
```

```
public:
    void get_marks(float, float);
    void get_marks(void);
};

void Exam :: get_marks(float m1, float m2)
{
    maths = m1;
    maths = m2;
}

void exam :: get_marks()
{
    cout << "the marks in math are" << math << endl;
    cout << "the " << " physics " << " " << physics << endl;
}

class Result : public exam
{
    float percentage;
    public:
        void display_results()
        {
            get_roll_number();
            get_marks();
        cout << "your result is " << (math + physics)/2 << "%"
                                                    endl;
        }
};

int main()
{
*/
```

Notes:

If we are inheriting B from A and C from B:
[A → B → C]

1. A is the base class for B, B is the base class for C.
2. A → B → C is called inheritance path.

```
Result harry;
harry.set_roll_number(420);
harry.set_marks(94.0, 90.0);
harry.display_result();
return 0;
}
```

Multiple Inheritance

Syntax:

```
class derivedC: visibility_mode base1, visibility_mode base2
{
    class body of class "derivedC"
};
```

```
class Base1 {
  protected:
    int base1int;
  public:
    void set_base1int(int a)
    {
        base1int = a;
    }
    };
class base2 {
    protected:
      int base2int;
    public:
      void set_base2int(int a)
      {
          base2int = a;
      }
      };
  class Base3 {
    protected:
      int base3int;
    public:
      void set_base3int(int a)
      {
          base3int = a;
      }
      };
class derived : public base1, public base2, public base3
  {
      public:
        void show()
        {
```

```
cout<<" base1 value"<< base1int<<endl;
cout<<" base2 value"<< base2int<<endl;
cout<<"base3 value"<<base3int<<endl;
cout<<"Sum"<<base1int+base2int+base3int<<endl;
}
};
```

The inherited class will look something like this:

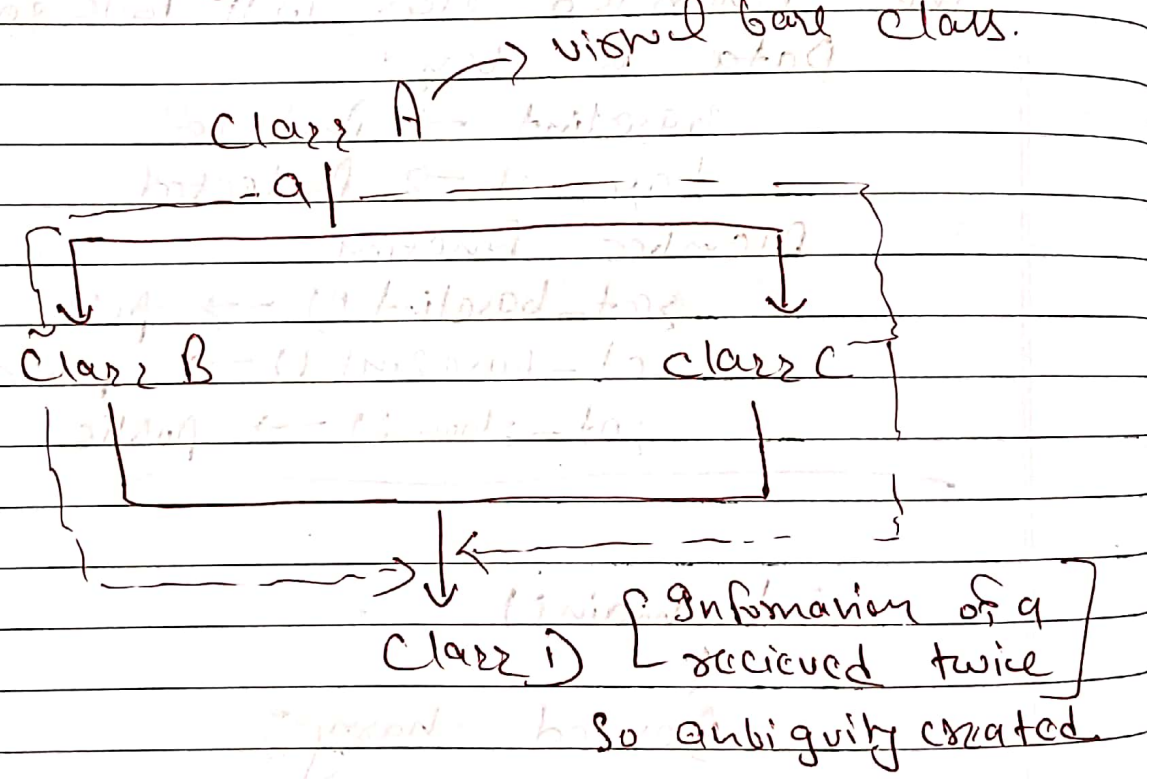
Data members:
- base1int -> Protected
- base2int -> Protected

Member function:
- set_base1int() -> Public
- set_base2int() -> Public
- set_show() -> Public

```
int main()
{
Derived harry;

harry.set_base1int(25);
harry.set_base2int(5);
harry.set_base3int(15);
harry.show();

return 0;
}
```

"Ambiguities During Inheritance"

When function with same signature (or prototype) appears in multiple base classes, then on accessing the same function with the help of the object of derived class, ambiguity may arise, as compiler cannot decide which ~~not~~ base class function to call. So ambiguity may arises.



So, solve this problem we use virtual base class.

→ class B : virtual public A {

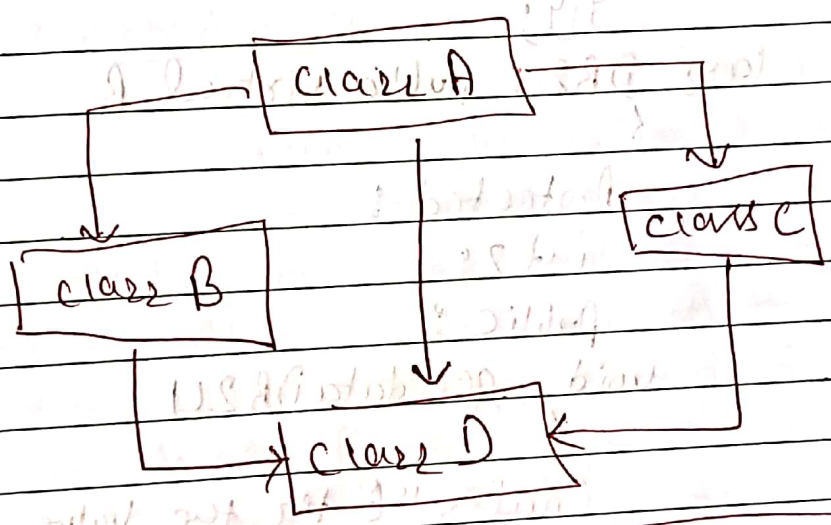
}

→ class C : virtual public A {

}

* Virtual Base class:

→ Virtual base class can be used to remove the ambiguity problem.
→ It will create a virtual path from grandparent class to the child class, so that only one copy of grandparents member is available in the child class, hence ambiguity is resolved.




```
class B
{
protected:
  int x;
public:
void get data-B()
  {
   cout<<"Enter value of x:";
   cin>>x;
  }
};
```

```
class DB1 : public virtual B
{
  protected:
    int Y;
  public:
  void get_dataDB1()
  {
    Cout "Enter the value of Y";
    cin>> Y;
  }
class DB2 : public virtual B
{
  protected:
    int Z;
  public:
  void get_dataDB2()
  {
  Cout<< "Enter the value of Z";
    cin>> Z;
  }
class D : public DB1, public DB2
  {
  public:
    void Sum()
    {
    int result;
    result = X+Y+Z;
    Cout<< "In Result is :" <<result;
    }
  };
```

```
int main()
{
    D obj1;
    obj1.get_dataB();
    obj1.get_dataDB1();
    obj1.get_dataDB2();
    obj1.sum();
}
```

Function Overriding :

- When function with same signature (or prototype) is redefine in the derived class [i.e base and derived class have same function] then on accessing the same function with the object of derived class, always derived class version of that function will be called, as it will overrid (or hid) the base class function.

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

```
void show()
{
    cout<<"Derived class B";
}
};
int main()
{
    B obj1;
    obj1.show();
// obj1.A::.show();
    return 0;
}
```

Constructor in Derived class

- We can use constructor in derived classes in C++.
- In base constructor does not have any argument there is no need of any constructor in derived class.
- But if there is are one or more argument in the base class constructor, derived class need to pass argument to the base class constructor.
- If both base and derived class have constructors, base class constructor is executed first.

Constructor in multiple Inheritance

In multiple Inheritance, base class are constructed in order in which they appear in class declaration.
In multilevel Inheritance, the constructor are executed in the order of Inheritance.

Special Syntax:

C++ support an special syntax for passing argument to multiple class base class.
The constructor of the derived class receives all the arguments at once and then will pass the value Calls to the respective base class.
The body is called after all the constructor are finished executing.

Derived-Constructor (arg1, arg2, arg3 ...); Base1-Constructor (arg1, arg2), Base2-Constructor

Special Case of virtual Base class:

The Constructor for virtual base class are invoked before an nonvirtual base class.
If there are multiple virtual base class, they are invoked in the order declared.
Any non-virtual base class are constructed before the derived class constructor is

⇒ Case 1:

class B : public A {
//order of execution of constructor → first A() then
};

⇒ Case 1:

class B : public A
{

order of execution of constructor :
→ first A() then B()

};

⇒ Case 2:

class A : public B, public C
{

order of execution of constructor :
→ B() then C() and A()

};

Case 3:

class A: public B, virtual public C
{

Order of execution of constructor
-> C() then B() and A()

class Base1
{
int data1;
public:
Base1(int i)
{
data1=i;
cout<<"Base class constructor called";
}
void printdatabase1(void)
{
cout<<"the value of data1 is "<<data1;
}
};

class Base2
{
int data2;
public:
Base2(int i)
{

```
    data2 = i;
    cout << "Base2 class constructed called";
    }
void printdatabase2(void)
    {
    cout << "the value of data2 is" << data2;
    }
    };

class Derived : public Base2, public Base1
    {
    int derived1, derived2;
    public:
Derived( int a, int b, int c, int d) : Base2(b), Base1(a)
    {
        derived1 = c;
        derived2 = d;
        cout << "Derived class constructor called";
    }

void printdataderived(void)
    {
    cout << "the value of derived1 is : " << derived1;
    cout << "the value of derived2 is : " << derived2;
    } };
int main() {
Derived harry(1, 2, 3, 4);
harry.printdatabase1();
harry.print database2();
harry.printdatabase dataDerived();
    return 0;
    }
```