Object Oriented Programming using C++

Inheritance

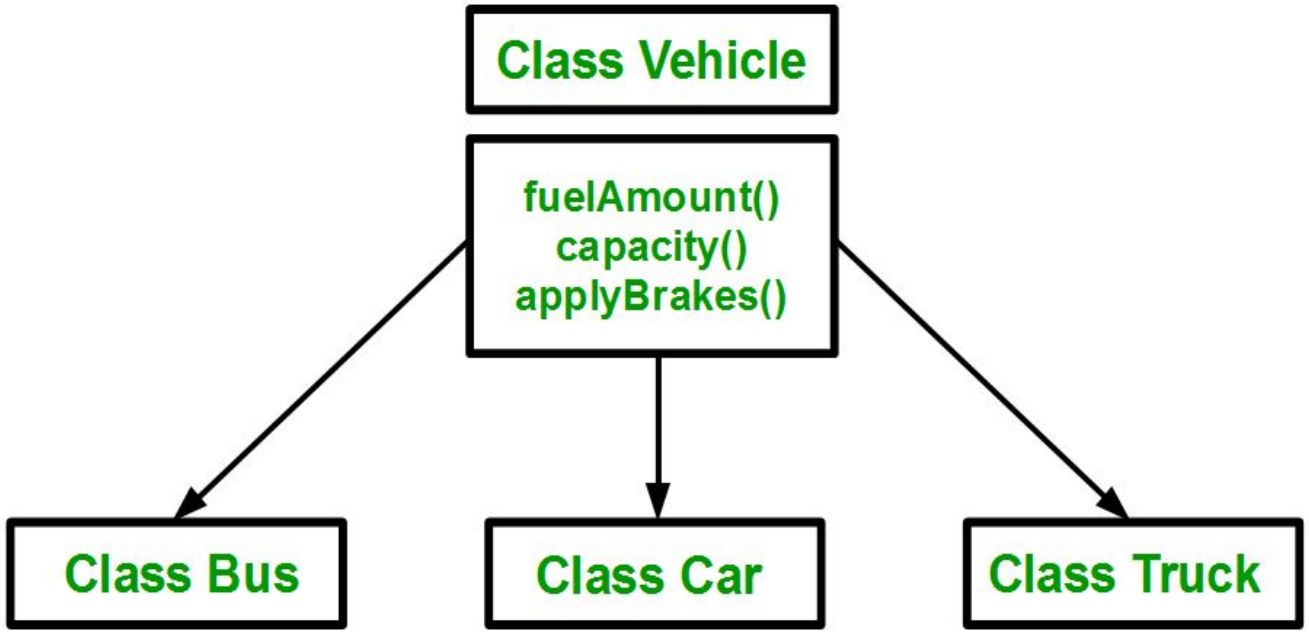
Introduction

- The mechanism of deriving properties of a class to another class is called Inheritance.

- The new class or the class that inherits properties from another class is called sub class or derived class.

- The old class or the class whose properties are inherited by new class is called Base Class or Super Class.

Contd...



- The derived class inherits some or all of the properties from the base class.
- A class can inherits properties from more than one class or from more than one level.

Advantages of Inheritance:

- **Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

- In the previous example, 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
};
```

subclass_name: It is the name of the derived class.

access mode: When deriving a class from a base class, the base class may be inherited through ***public***, ***protected*** or ***private*** inheritance.

- protected. Data members or Member functions which are declared as protected can be accessed in the derived class or within the same class
- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

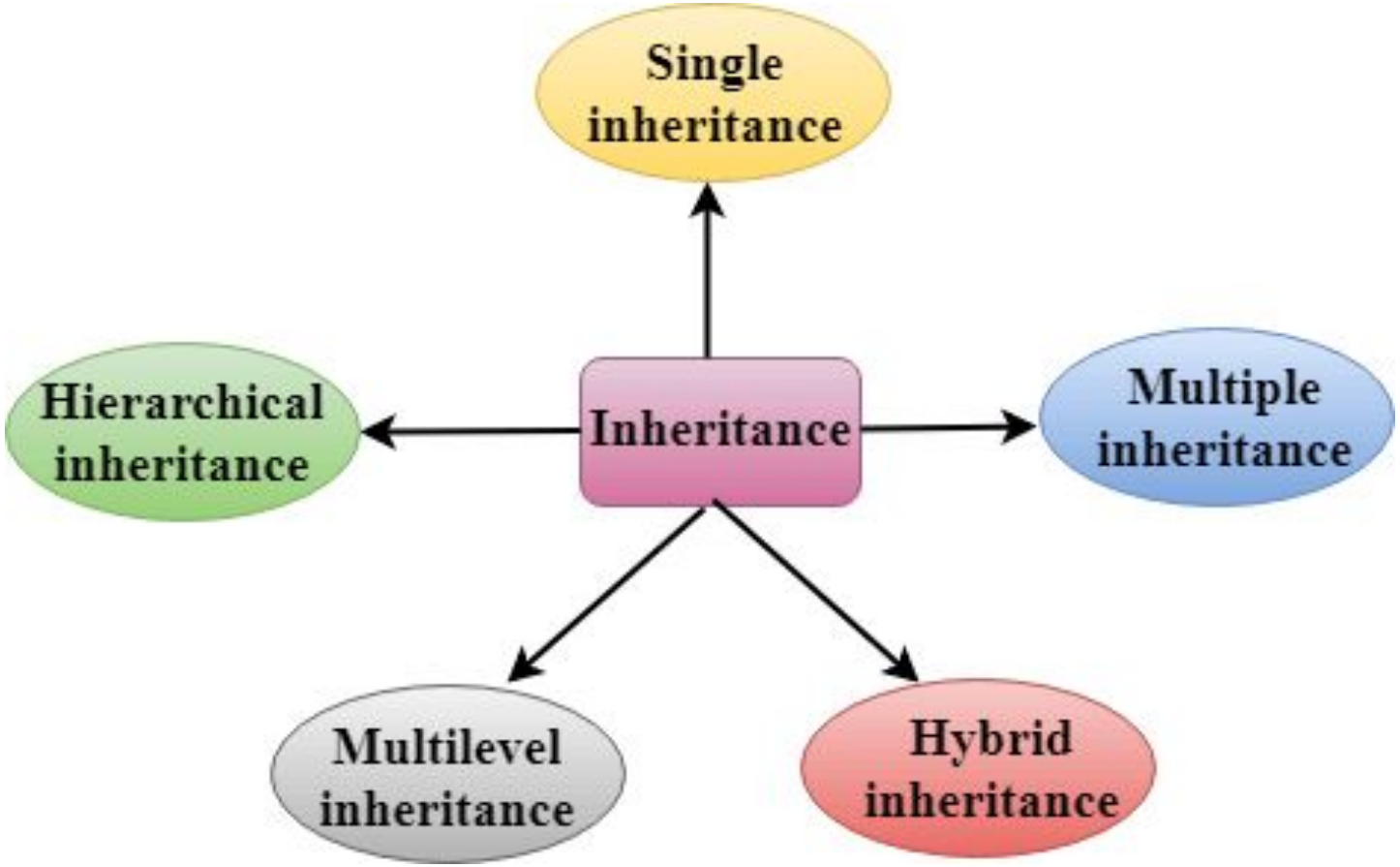
Contd…

base_class_name: It is the name of the base class.

- *When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class.*
- Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- *When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class.*
- Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.
- *When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** /private members of the derived class.*

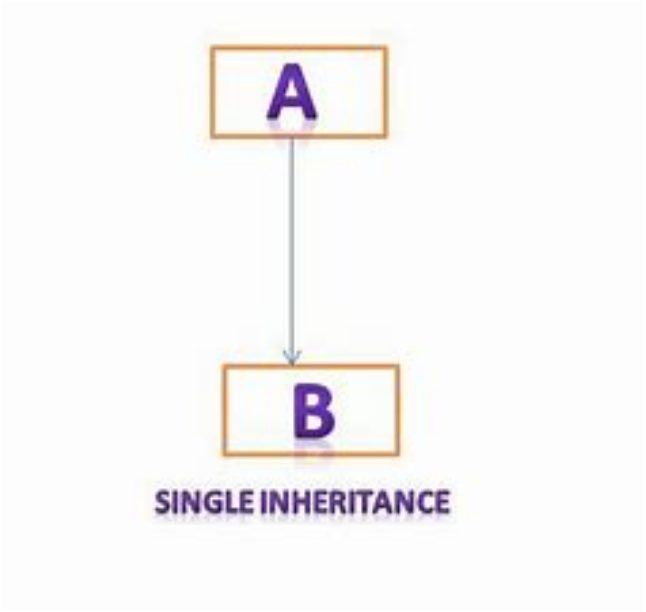
Types Of Inheritance

- **C++ supports five types of inheritance:**



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.



Example:

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

 class Account
{
  public:
  float salary = 60000;
 };

  class Programmer: public Account
{
  public:
  float bonus = 5000;
  };
  int main(void)
{
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
}
```

O/P:
Salary:60000
Bonus:5000

C++ Single Level Inheritance Example: Inheriting Methods

```
#include <iostream>
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};
```

```
class B : private A
{
   public:
   void display()
   {
      int result = mul();
      std::cout <<"Multiplication of a and b is : "<<result<< std::endl;

   }
};
int main()
{
  B b;
  b.display();

   return 0;
}
```

O/P:

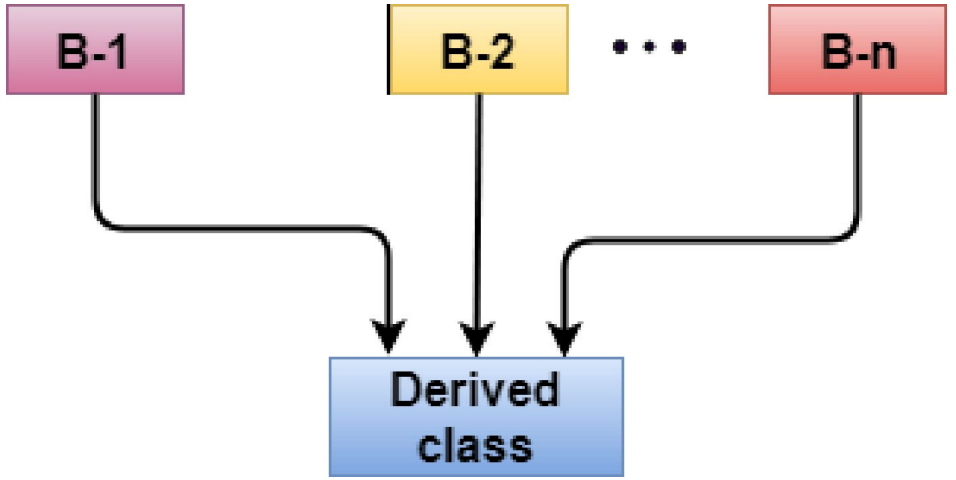
Multiplication of a and b is : 20

2. Multiple Inheritance

- **Multiple inheritance** is the process of deriving a new class that inherits the properties from two or more classes.

Syntax:

```
class D : visibility B-1, visibility B-2,….,B-n
{
   // Body of the class;
}
```



Example:

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

class stud //base class-1
{
 protected:
int roll, m1, m2;
 public:
void get()
{
cout << "Enter the Roll No.: ";
cin >> roll;
cout << "Enter the two highest marks: ";
 cin >> m1 >> m2;
}
};
class extracurriculum //Base class-2
{
protected:
int xm;
 public:
void getsm()
{
cout << "\nEnter the mark for Extra Curriculum Activities: ";
cin >> xm;
 }
 };
class output : public stud, public extracurriculum
{
int tot, avg;
public:
void display()
{
tot = (m1 + m2 + xm);
avg = tot / 3;
cout << "\n\n\tRoll No : " << roll << "\n\tTotal : " << tot;
cout << "\n\tAverage : " << avg;
}
 };
int main()
{
output O;
O.get();
 O.getsm();
O.display();
}
```

O/P:

```
Enter the Roll No.:26
Enter the two highest Marks:35 40
Enter the mark for Extra Curriculum
activities:36
Roll No:26
Total:111
Average:37
```

Ambiguity in Multiple Inheritance

- Ambiguity in multiple inheritance occurs when two or more base classes have same named member function.

- If the object of derived class needs to access one of the same named member function then it results in ambiguity.

```
#include <iostream>

class Animal // Base Class 1
{
  public:
int weight;
void info()
{
std::cout << "Weight:" << weight << "cm\n";
}
};
```

```
class Wing // Base Class 2
{
  public:
int wingwidth;
void info()
{
std::cout << "Wing width:" << wingwidth << "cm\n";
}
};

class Bird: public Animal, public Wing  // Derived Class 2
{

};


int main()
{
Bird eagle;
eagle.weight = 3;   // in kgf
eagle.wingwidth = 280;  // in cm

eagle.info();

getchar();
return 0;

}
```

Contd…

- C++ allows to use same function names in different classes.
- To avoid this error in Multiple Inheritance, use their class names with :: operator. So first we should write object then dot with base class name and :: operator with the method name in that base class.

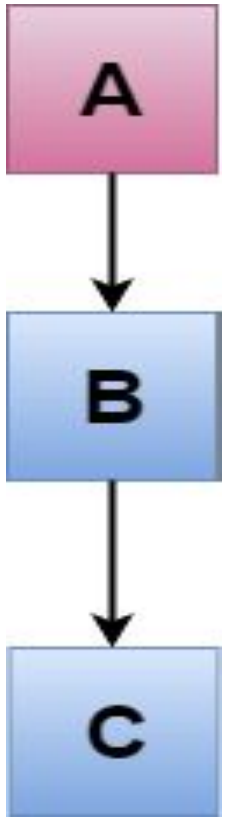
Ex:
- eagle.Wing::info();
- eagle.Animal::info();

Program:

```
#include <iostream>

class Animal // Base Class 1
{
  public:
int weight;

void info()
{
std::cout << "Weight:" << weight << "cm\n";
}
};

class Wing // Base Class 2
{
  public:
int wingwidth;


void info()
{
std::cout << "Wing width:" << wingwidth << "cm\n";
}

};

class Bird: public Animal, public Wing  // Derived Class 2
{

};


int main()
{
Bird eagle;
eagle.weight = 3;   // in kgf
eagle.wingwidth = 280;  // in cm

eagle.Wing::info();
eagle.Animal::info();

getchar();
return 0;
}
```

3.Multilevel inheritance

- It is a process of deriving a class from another derived class.



Ex: Multilevel Inheritance

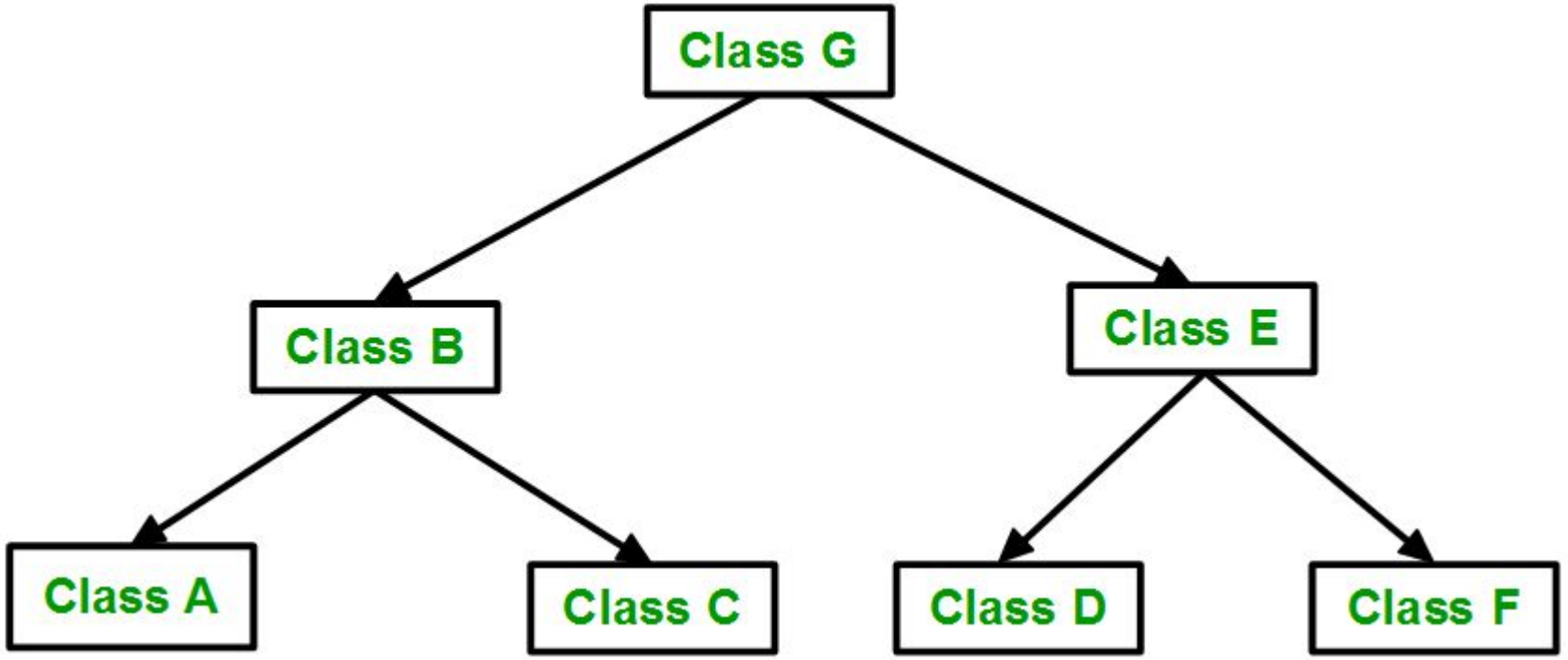
```
#include <iostream>
using namespace std;

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

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};

// sub class derived from the derived base class
fourWheeler
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;
}
```

O/P:

This is a vehicle
Objects with 4 wheels are vehicle
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.



The above image shows the combination of hierarchical and multiple inheritance

Ex: Hierarchical Inheritance

```
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;
}
```

O/P:

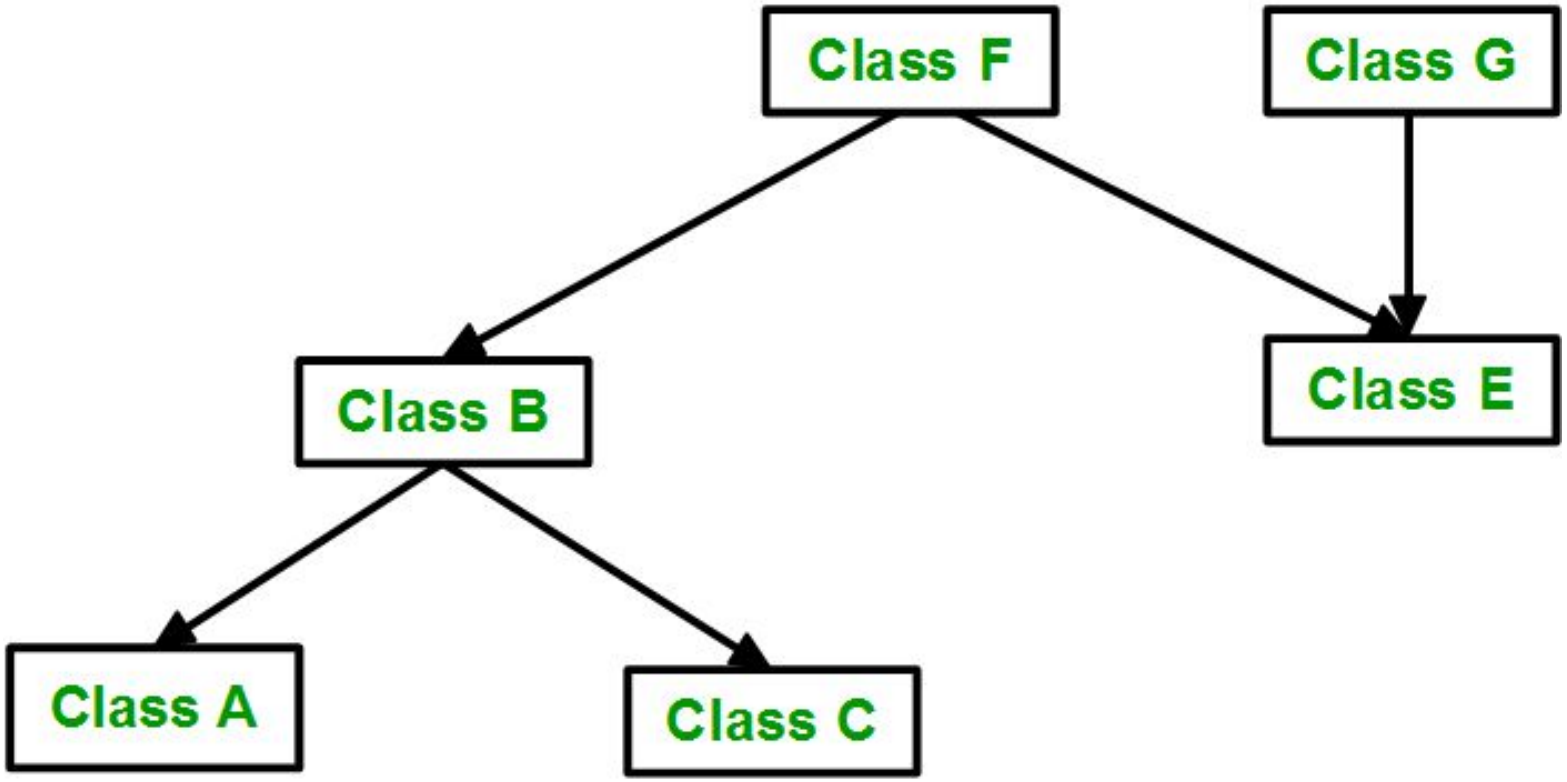
```
This is a Vehicle
This is a Vehicle
```

Where hierarchical inheritance does is used?

It is used in the following cases where hierarchy is to be maintained. For instance, the database of an organization is stored in the hierarchical format. There are different sections of an organization such as IT, computer science, Civil, Mechanical, etc. Each organization has the same attributes such as student name, roll number, year, etc. which comes under a class Student. Hence all the sections inherit the student properties and thus following the format of hierarchical inheritance.

5. Hybrid (Virtual) Inheritance

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



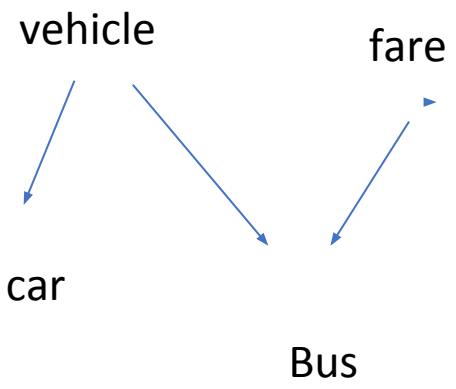
Example: Hybrid Inheritance

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

// base class 1
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

//base class 2
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;
}
```

O/P:

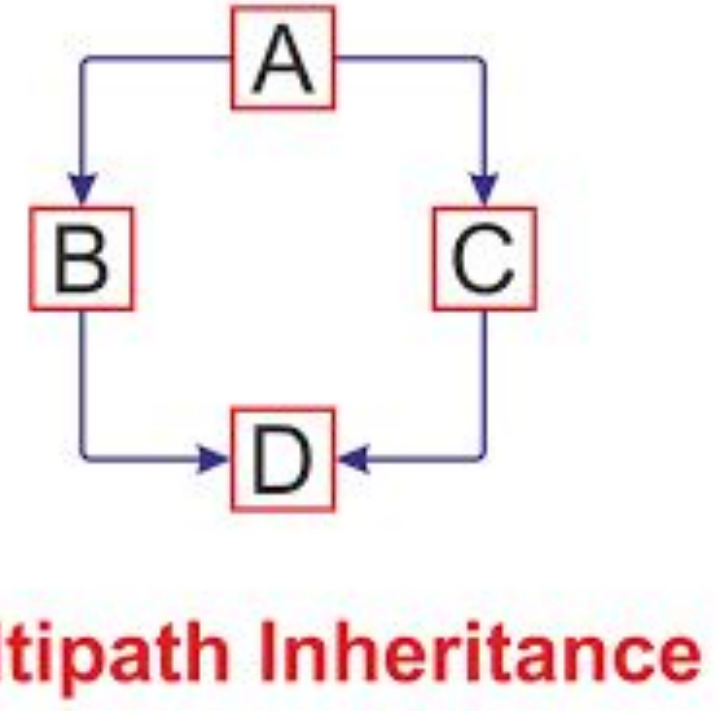
This is a Vehicle

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 arise in this type of inheritance.*



Multipath Inheritance

There are 2 ways to avoid this ambiguity:

1. Avoiding ambiguity using scope resolution operator:

- *Using scope resolution operator we can manually specify the path from which data member a will be accessed.*

2. Avoiding ambiguity using virtual base class:

- *To overcome the ambiguity occurring due to multipath inheritance, C++ provide a keyword virtual.*
- *The key word virtual declares the specified classes virtual so that only one copy of the base class will come to the derived class.*

Example: **Avoiding ambiguity using scope resolution operator**

```
#include <conio.h>
#include <iostream.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
c: 40
```

- In 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.

Example: **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;



     }
```

```
O/P:
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.

Constructor and Destructor in Inheritance

- When we are using the constructors and destructors in the inheritance, parent class constructors and destructors are accessible to the child class hence when we create an object for the child class, constructors and destructors of both parent and child class get executed

```
#include<iostream.h>
using namespace std;
class parent //parent class
{
public:
parent()//constructor
{
cout<<"Parent class Constructor\n";
}
~parent()//destructor
{
cout<<"Parent class Destructor\n";
}};
```

```
class child : public parent//child class
{
public:
child() //constructor
{
cout<<"Child class Constructor\n";
}
~ child() //destructor
{
cout<<"Child class Destructor\n";
}
};
int main()
{
child c;//automatically executes both child and parent class //constructors  and
destructors because of inheritance
}
```

O/P:

Parent class Constructor

Child class Constructor

Child class Destructor

Parent class Destructor

Inheritance in Parametrized Constructor/ Destructor

- In the case of the default constructor, it is implicitly accessible from parent to the child class but parametrized constructors are not accessible to the derived class automatically, for this reason, explicit call has to be made in the child class constructor for accessing the parameterized constructor of the parent class to the child class.

Syntax

<class_name>:: constructor(arguments)

- Whenever you are using the parameterized constructor in parent class it is mandatory to define a default constructor explicitly.

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

class parent
{

int x;
public:
// parameterized constructor
parent(int i)
{
x = i;
cout << "Parent class Parameterized Constructor\n";
}
};
```

```
class child: public parent
{
int y;
public:
// parameterized constructor
child(int j) : parent(j)  //Explicitly calling
{
y = j;
cout << "Child class Parameterized Constructor\n";
}
};
int main()
{
child c(10);

return 0;
}
```

O/P:

Parent class Parameterized Constructor

Child class Parameterized Constructor

Abstract Class

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class.

Characteristics of Abstract Class:

1. *A class is abstract if it has at least one pure virtual function.*

Virtual Function:

A **virtual** function is a *member* function which is declared in the *base* class using the keyword virtual and is re-defined (Overriden) by the *derived* class.

- This especially applies to cases where a pointer of base class points to an object of a derived class.

- A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class