UNIT-3 Operator Overloading and Inheritance

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different tasks.

Syntax:

```
Return_Type classname :: operator op(Argument list)
{
   Function Body
}
```

Return_Type is the value type to be returned to another object.
operator op is the function where the operator is a keyword.
op is the operator to be overloaded.

- **OVERLOADING UNARY OPERATOR**

- The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.
- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Example of overloading minus operator

```
#include <iostream>
using namespace std;
class Distance {
  private:
    int feet;    // 0 to infinite
    int inches;  // 0 to 12
 public:
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
// method to display distance
    void displayDistance()  {
      cout << "F: " << feet << " I: " << inches << endl;
    }
// overloaded minus (-) operator
    Distance operator- ()  {
feet = -feet;
inches = -inches
      return Distance(feet, inches);
    }
};
int main() {
  Distance D1(11, 10), D2(-5, 11);
 D1.displayDistance();
D2.displayDistance();
return 0;}
```

Example of Binary operator overoading

```
#include <iostream>
using namespace std;
 class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
     public:
 double getVolume(void) {
      return length * breadth * height;
   }
   void setLength( double len ) {
      length = len;
   }
 void setBreadth( double bre ) {
      breadth = bre;
  }
 void setHeight( double hei ) {
      height = hei;
   }
 // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
```

```
// Main function for the program
int main() {
   Box Box1;     // Declare Box1 of type Box
   Box Box2; // Declare Box2 of type Box
   Box Box3; // Declare Box3 of type Box
   double volume = 0.0; // Store the volume of a box here
    // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
  // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
```

Continued…

```
 // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
 // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
  // Add two object as follows:
   Box3 = Box1 + Box2;
 // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
return 0;
}
```

Data conversion

- There can be 3 types of situations that may come in the data conversion between incompatible data types:
- **Conversion of primitive data type to user-defined type:** To perform this conversion, the idea is to use the constructor to perform type conversion during the object creation.
- **Conversion of class object to primitive data type :** In this conversion, the **from** type is a class object and the **to** type is primitive data type. The normal form of an overloaded casting operator function, also known as a conversion function.
- **Conversion of one class type to another class type:** In this type, one class type is converted into another class type. It can be done in 2 ways :
- 1.Using constructor
- 2.Using Overloading casting operator

Below is the example to convert primary data type to user-defined data type:

```
#include <bits/stdc++.h>
using namespace std;
// Time Class
class Time {
        int hour;
        int mins;
public:
        // Default Constructor
        Time()
        {
                hour = 0; mins = 0;
        }
        Time(int t)
        {
                hour = t / 60;
                mins = t % 60;
        }
                void Display()
        {
                cout << "Time = " << hour<< " hrs and "<< mins << " mins\n";
        }
};
```

```
// Driver Code
int main()
{
        // Object of Time class
        Time T1;
        int dur = 95;

        // Conversion of int type to
        // class type
        T1 = dur;
        T1.Display();

        return 0;
}
```

Conversion of class object to primitive data type:

- **Syntax:**

```
operator typename()
{
   // Code
}
```

Typename()- data type in which we want to convert it

```
// C++ program to illustrate the type-conversion
#include <iostream>
using namespace std;
class Time {
    int hour;
    int mins;
public:
  Time() {
        hour = 0;
        mins = 0;
    }
Time(int t) {
        hour = t / 60;  // Convert total minutes to hours
        mins = t % 60;  // Remaining minutes after full hours
    }
    // Type conversion operator to convert Time object to int
    operator int() const {
        return hour * 60 + mins;  // Convert hours and minutes to
total minutes
    }
    // Function to print the value of class variables
    void Display() {
        cout << "Time = " << hour << " hrs and " << mins << "
mins\n";
    }
};int main() {
    // Object of Time class
    Time T1;
    int dur = 95;  // Duration in minutes

    // Conversion of int type to class type
    T1 = dur;
    T1.Display();

    // Conversion of class type to int type
    int totalMinutes = T1;
    cout << "Total minutes = " << totalMinutes << " mins\n";

    return 0;
            }
```

Conversion of one class type to another class type:

In this type, one class type is converted into another class type. It can be done in 2 ways :

1.Using constructor

2.Using Overloading casting operator

1.Using constructor :

- In the Destination class we use the constructor method
- //Objects of different types
- ObjectX=ObjectY;
- Here ObjectX is Destination object and ObjectY is source object

Example of conversion of one class type to another class type:

```
#include<iostream>
using namespace std;
class CGS
{
        int mts; //meters
        int cms; //centimeters
        public:
        void showdata()
{
        cout<<"Meters and centimeters in CGS system:";
        std::cout << mts<<" meters "<<cms<<" centimeters" << std::endl;
}
CGS(int x,int y) // parameterized constructor
{
        mts=x;
        cms=y;
}
int getcms()
{
        return cms;
}
int getmts()
{
        return mts;
}
};
```

```
class FPS
{
int feet;
int inches;
public:
FPS() // default constructor
{
        feet=0;
        inches=0;
}
FPS(CGS d2)
{
        int x;
        x=d2.getcms()+d2.getmts()*100;
        x=x/2.5;
        feet=x/12;
        inches=x%12;
        }
void showdata()
{
        cout<<"feet and inches in FPS system:";
        std::cout << feet<<" feet "<<inches<<"
inches" << std::endl;
}
};

int main()
{
        CGS d1(9,10);
        FPS d2;
        d2=d1;
        d1.showdata(); //to display CGS values
        d2.showdata(); //to display FPS values
        return 0;
}
```

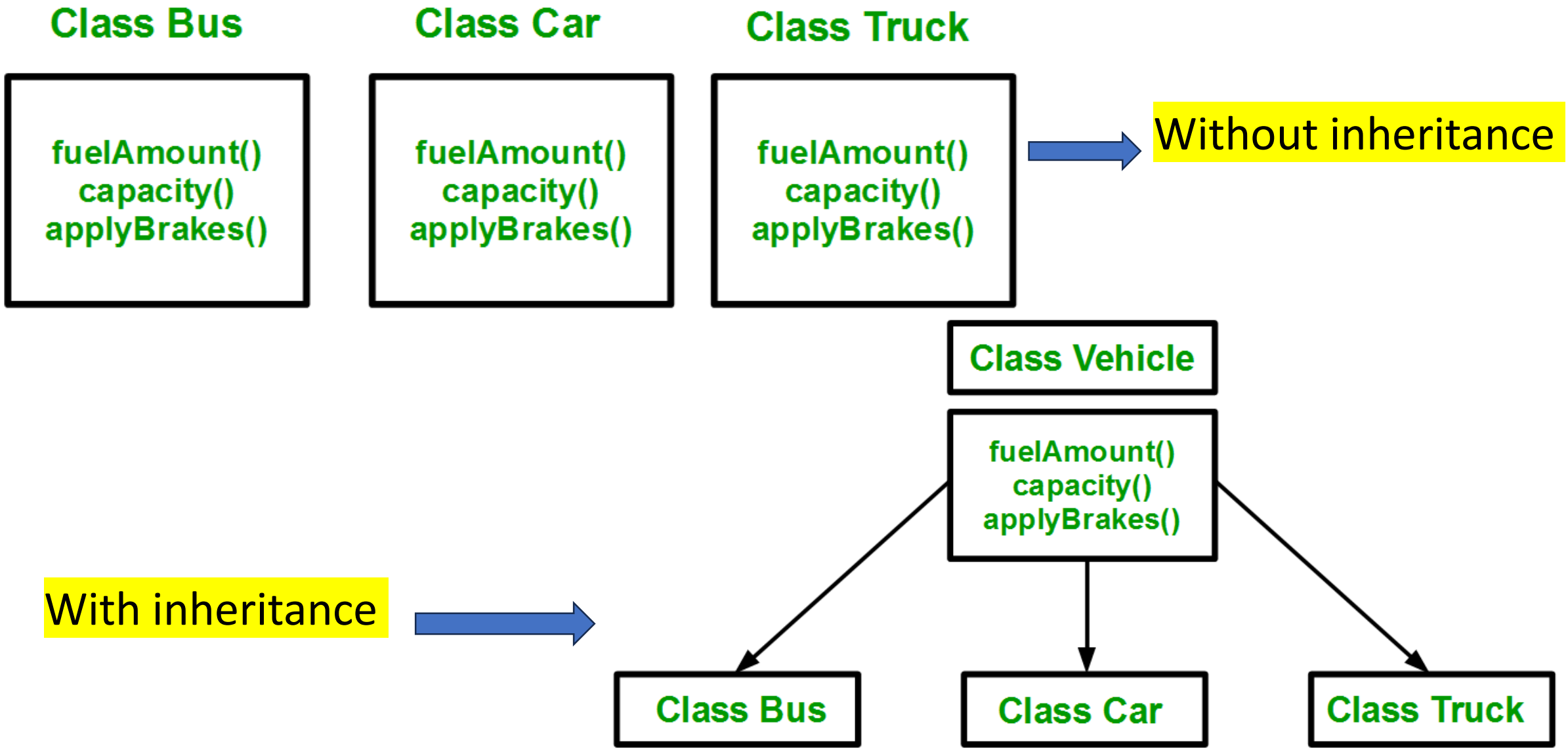
Output: Meters and centimeters in CGS system:9 meters 10 centimeters
feet and inches in FPS system:30 feet 4 inches

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 features of Object-Oriented Programming.

- Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class.
- ***Sub Class***: The class that inherits properties from another class is called Subclass or Derived Class.
- ***Super Class***: The class whose properties are inherited by a subclass is called Base Class or Superclass

Why do we use need inheritance????????



Creating derived class:

Syntax:

class <derived_class_name> : <access-specifier> <base_class_name>

{

    //body

}

Where

class — keyword to create a new class

derived_class_name — name of the new class, which will inherit the base class

access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default

base-class-name — name of the base class

Example:

1. class ABC : private XYZ //private derivation
   { }
2. class ABC : public XYZ //public derivation
   { }
3. class ABC : protected XYZ //protected derivation
   { }
4. class ABC: XYZ //private derivation by default
{ }

Note:

o When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

o On the other hand, 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 derived class.

Example: define member function without argument within the class and outside the class

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

class Person {
        int id;
        char name[100];

public:
        void set_p()
        {
                cout << "Enter the Id:";
                cin >> id;
                cout << "Enter the Name:";
                cin >> name;
        }

        void display_p()
        {
                cout << endl <<"Id: "<< id <<
"\nName: " << name <<endl;
        }
};
```

```
class Student : private Person {
        char course[50];
        int fee;
public:
        void set_s()
        {
                set_p();
                cout << "Enter the Course Name:";
                cin >> course;
                cout << "Enter the Course Fee:";
                cin >> fee;
        }

        void display_s()
        {
                display_p();
                cout <<"Course: "<< course <<
"\nFee: " << fee << endl;
}
};

int main()
        {
Student s;
s.set_s();
s.display_s();
return 0;
}
```

Output:
Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: GCS
Enter the Course Fee:70000

Id: 101
Name: Dev
Course: GCS
Fee: 70000

define member function with argument outside the class

```
#include<iostream>
#include<string.h>
using namespace std;
class Person
{
        int id;
        char name[100];

        public:
                void set_p(int,char[]);
                void display_p();
};
void Person::set_p(int id,char n[])
{
        this->id=id;
                strcpy(this->name,n);
}
void Person::display_p()
{
        cout<<endl<<id<<"\t"<<name;
}
class Student: private Person
{
        char course[50];
        int fee;
        public:
        void set_s(int,char[],char[],int);
        void display_s();
};
```

```
void Student::set_s(int id,char n[],char c[],int f)
{
        set_p(id,n);
        strcpy(course,c);
        fee=f;
}
void Student::display_s()
{
        display_p();
        cout<<"t"<<course<<"\t"<<fee;
}
Int main()
{
        Student s;
        s.set_s(1001,"Ram","B.Tech",2000);
        s.display_s();
        return 0;
}
```

C++ program to demonstrate implementation of Inheritance

Output: Child id is: 7
Parent id is: 91

```
#include <bits/stdc++.h>
using namespace std;
// Base class
class Parent {
public:
        int id_p;
};
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;
}
```

Modes of Inheritance: There are 3 modes of inheritance.

- Public Mode: If we derive a subclass 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 the derived class.
- Protected Mode: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
- Private Mode: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

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
};
```

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

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single Inheritance:

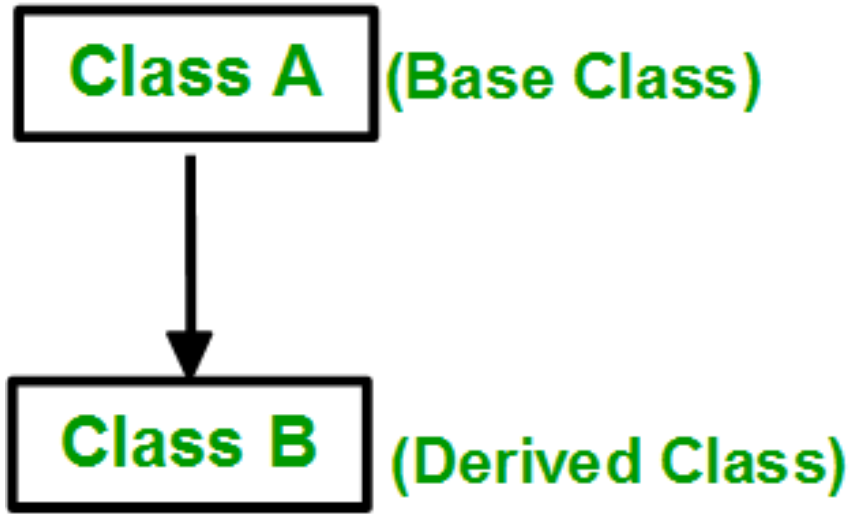
- In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

Syntax:

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

OR

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



Syntax:

```
class subclass_name : access_mode base_class
{
  // body of subclass
};
OR
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
```

1. 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;
}
```

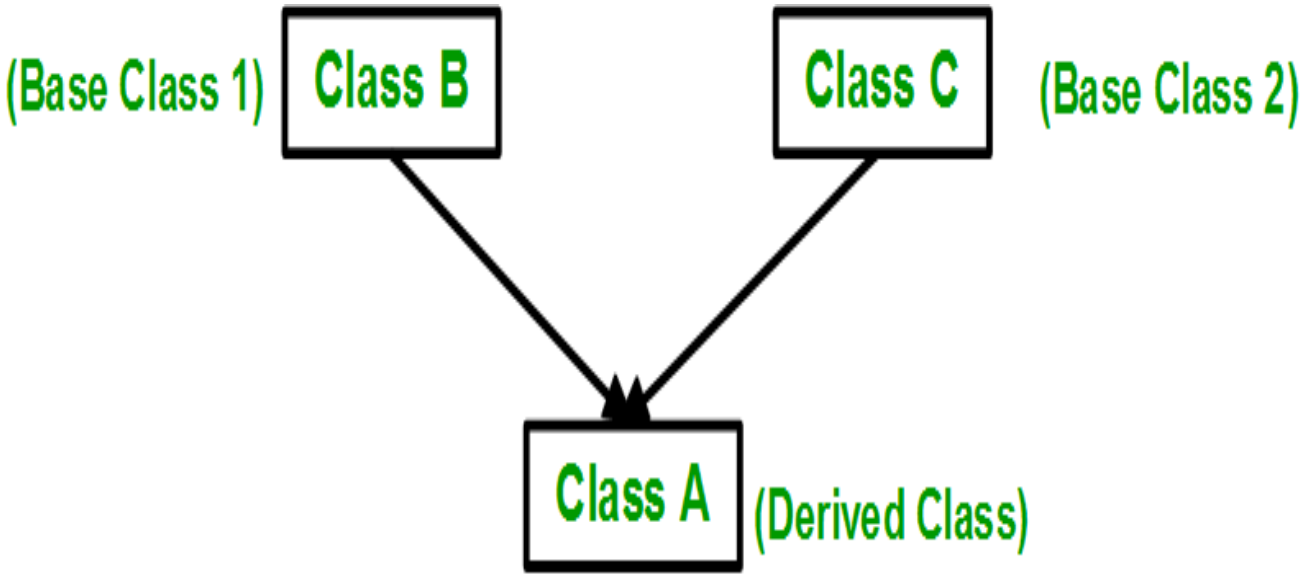
```
#include<iostream>
using namespace std;
class A
{       protected:
        int a;
        public:
                void set_A()
                {       cout<<"Enter the Value of A=";
                        cin>>a;

                }
                void disp_A()
                {       cout<<endl<<"Value of A="<<a;
                }
};
class B: public A
{       int b,p;
        public:
                void set_B()
                {       set_A();
                        cout<<"Enter the Value of B=";
                        cin>>b;
                }
                void disp_B()
                {
                        disp_A();
                        cout<<endl<<"Value of B="<<b;
                }
                void cal_product()
                {
                        p=a*b;
                        cout<<endl<<"Product of "<<a<<" *
"<<b<<" = "<<p;
                }

};

main()
{

   B _b;
   _b.set_B();
   _b.cal_product();

   return 0;

}
```

Multiple Inheritance

- 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. The constructors of inherited classes are called in the same order in which they are inherited.

Syntax:

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



Example:

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

// 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;
}
```

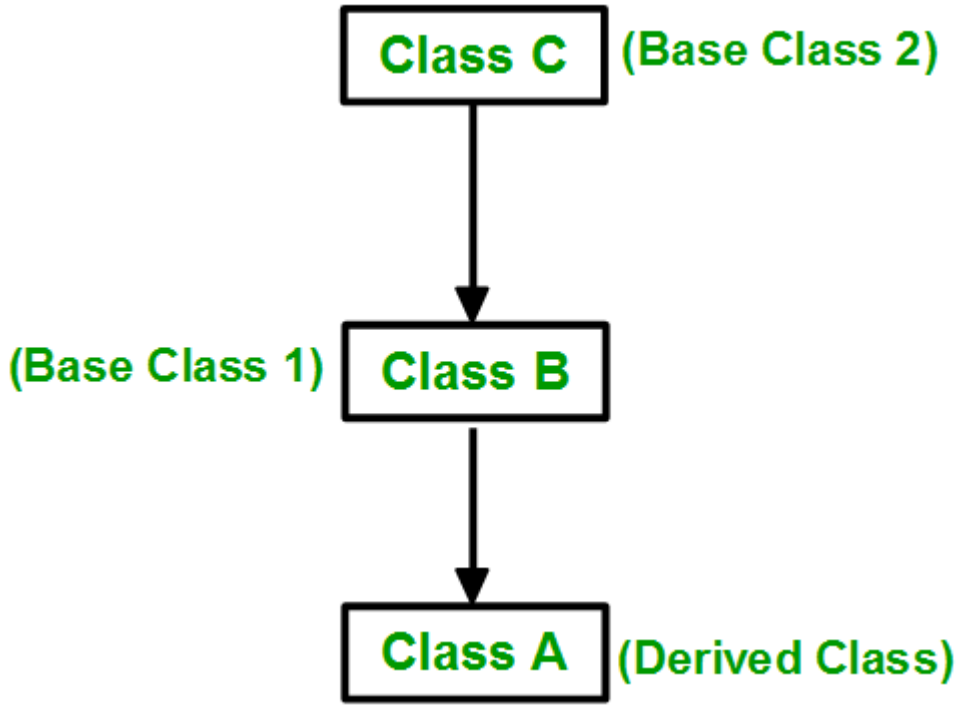
Multilevel Inheritance

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

Multilevel Inheritance in C++

Syntax:-

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



// C++ program to implement Multilevel Inheritance

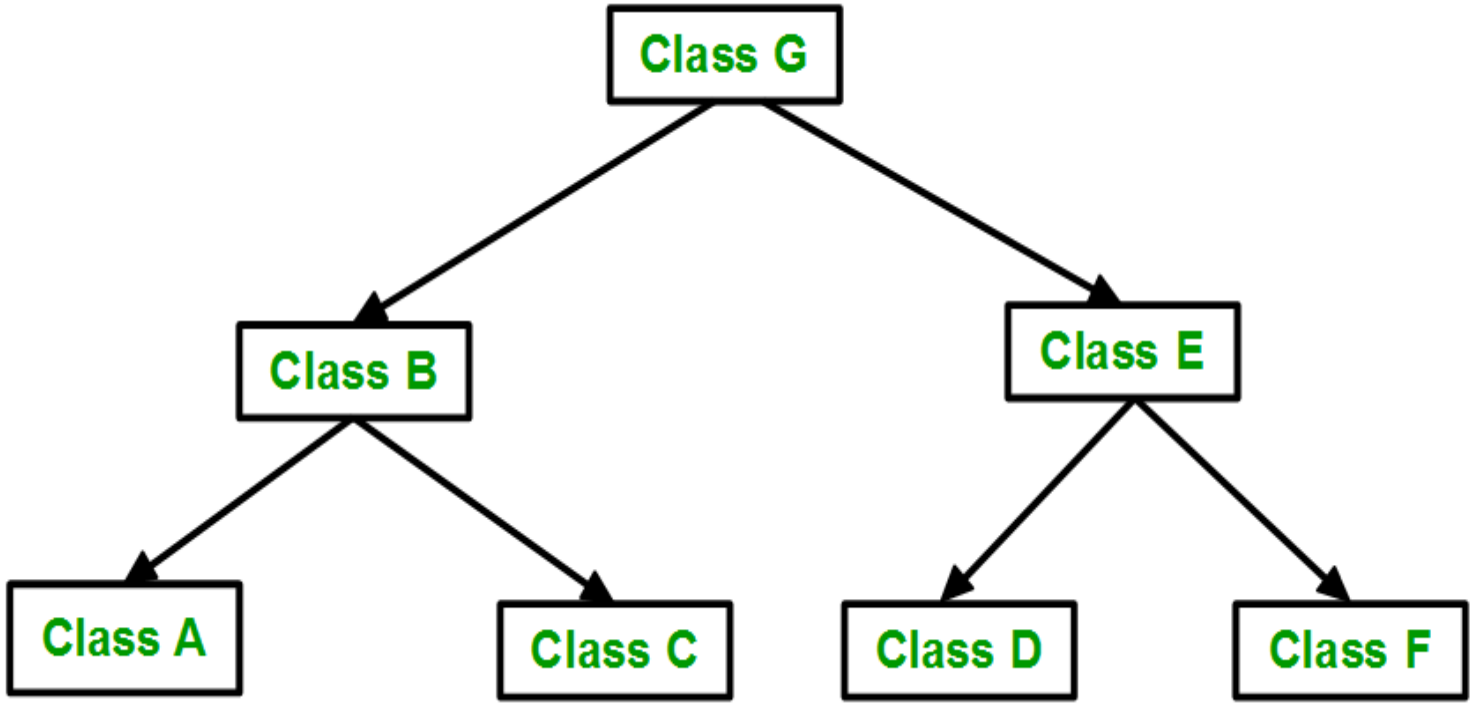
```
#include <iostream>
using namespace std;

// 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;
}
```

Hierarchical Inheritance in C++

- 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.

```
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.
}
```

Syntax

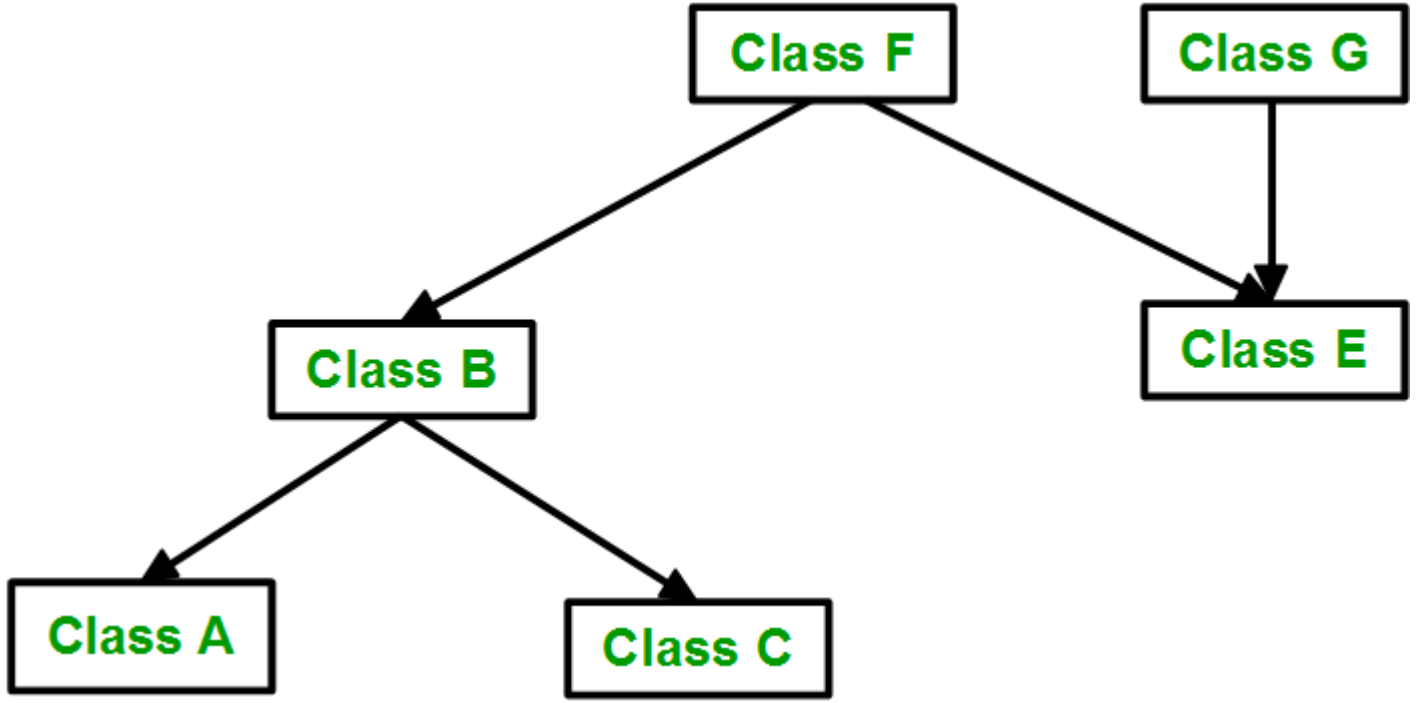


// C++ program to implement Hierarchical Inheritance

```
#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()
{
        Car obj1;
        Bus obj2;
        return 0;
}
```

Hybrid (Virtual) Inheritance:

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



// C++ program for Hybrid Inheritance

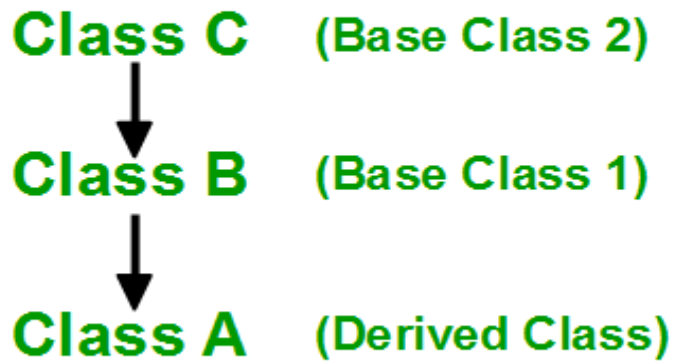
```
#include <iostream>
using namespace std;

// 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()
{

        Bus obj2;
        return 0;
}
```

Derived class constructor

// C++ program to show the order of constructor calls in Multiple Inheritance

```
#include <iostream>
using namespace std;
// first base class
class Parent1
{       public:
        // first base class's Constructor
        Parent1()
        {
                cout << "Inside first base class" << endl;
        }
};
// second base class
class Parent2
{
        public:
        // second base class's Constructor
        Parent2()
        {
                cout << "Inside second base class" << endl;
                }
};
class Child : public Parent1, public Parent2
{
        public:

        // child class's Constructor
        Child()
        {
                cout << "Inside child class" << endl;
        }
};
int main() {
        Child obj1;
        return 0;
}
```

Order of Inheritance



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)