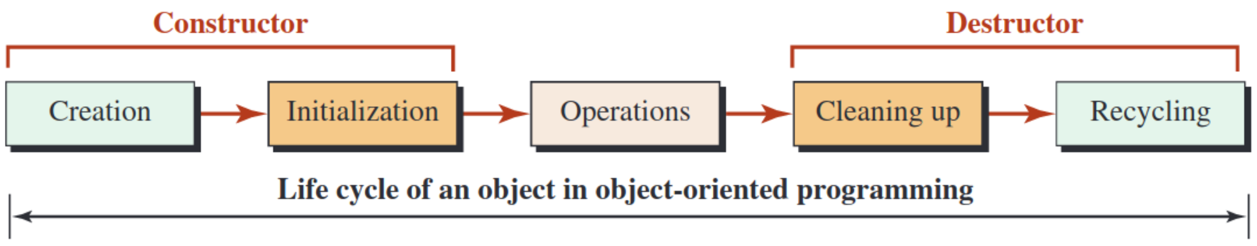
CONSTRUCTORS AND DESTRUCTORS

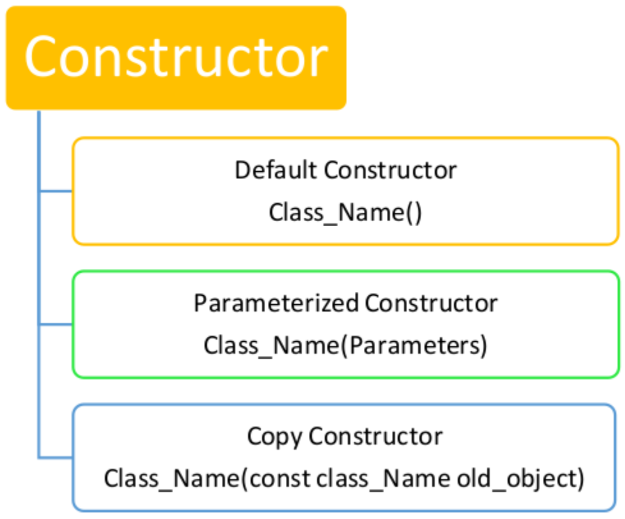
A constructor is a special type of member function of a class which initializes objects of a class. Constructor has same name as the class . Constructors don’t have return type. A constructor is automatically called when an object is created. It must be placed in public section of class. If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body). On the other hand, when we do not need an object anymore, the object should be cleaned up and the memory occupied by the object should be recycled. Cleanup is automatically done when another special member function named a destructor is called, the object goes out of scope, and the body of the destructor is executed; recycling is done when the program is terminated.

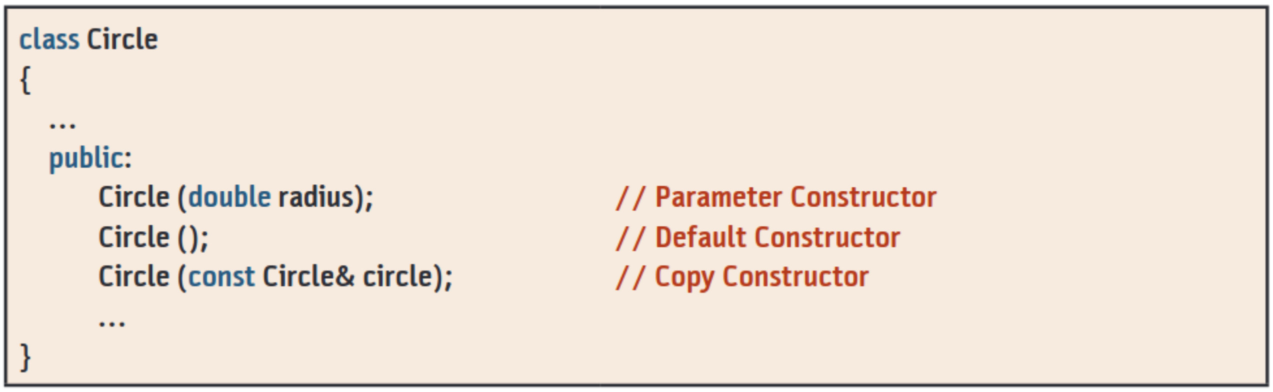
A constructor is a special member function that creates and initializes an object.
A destructor is a special member function that cleans and destroys an object.



CONSTRUCTORS

A constructor is a member function that creates an object when it is called and initializes the data members of an object when it is executed. The declaration of the data members in the class definition does not initialize the data members; the declaration just gives the names and the types of the data members. A constructor has two characteristics: It does not have a return value, and its name is the same as the name of the class. A constructor cannot have a return value (not even void) because it is not designed to return anything; its purpose is different. It creates an object and initializes the data members.





```
class Circle
{
   ...
   public:
        Circle (double radius);              // Parameter Constructor
        Circle ();                           // Default Constructor
        Circle (const Circle& circle);       // Copy Constructor
        ...
}
```

Default Constructor: The default constructor is a constructor with no parameters. It is used to create objects with each data member for all objects set to some literal values or default values. we cannot overload the default constructor because it has no parameter list.

The default constructor cannot be overloaded for a class.

```
#include <iostream>
using namespace std;
class construct{
public:
int a, b;
// Default Constructor
construct()
          {
          a = 10;
          b = 20;
          }
};
int main()
{

// Default constructor called automatically
// when the object is created
construct c;
cout << "a: " << c.a << endl << "b: " << c.b;
return 1;
}
```

```
Output: a: 10
B: 20
```

Parameterized Constructor: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. The parameter constructor can be overloaded, which means that we can have several parameter constructors each with a different signature. The advantage of the parameter constructor is that we can initialize the data members of each object with a specific value.

The parameter constructor can be overloaded for a class.

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

class Point
{
        private:
                int x, y;
        public:
                Point(int x1, int y1) // Parameterized
                Constructor {
                x = x1;
                y = y1;
}

int getX(){
return x;
}
int getY()
{
return y;
}
};

int main()
{
// Constructor called
Point p1(10, 15);

// Access values assigned by constructor
cout << "p1.x = " << p1.getX() << ", p1.y
= " << p1.getY();

return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

COPY CONSTRUCTOR: A copy constructor copies the data member values of the given object to the new object just created. The copy constructor has only one parameter that receives the source object by reference. we cannot overload the copy constructor because the parameter list is fixed and we cannot have an alternative form.

```
// Definition of a parameter constructor
Circle :: Circle (double rds)
: radius (rds)           // Initialization list
{
   // Any other statements
}
// Definition of a default constructor
Circle :: Circle ()
: radius (1.0)  // Initialization list. If it is missing, radius is set to garbage values
{
   // Any other statements
}
// Definition of a copy constructor
Circle :: Circle (const Circle& cr)
: radius (cr.radius)      // Initialization list
{
   // Any other statements
}
```

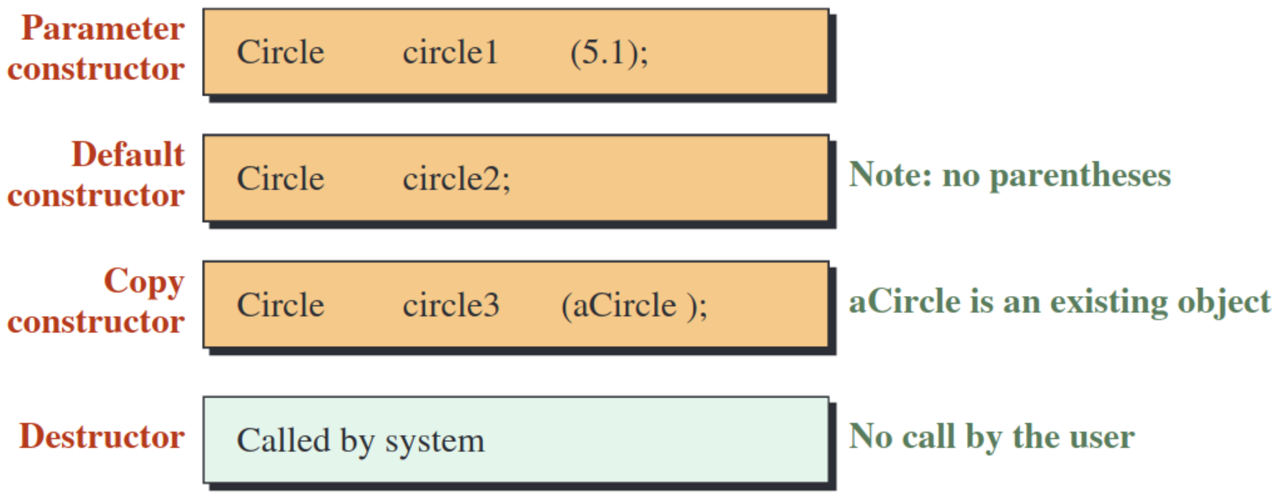
DESTRUCTOR: It is used to destroy the objects that have been created by a constructor. The destructor is a member function whose name is the same as the class name. Syntax- ~class_Name ( ) ; It never takes any argument nor does it return any value.

- the name of the destructor is the name of the class preceded by a tilde symbol (~), but the tilde is added to the first name, not the last name (the last name is the same for all member functions).
- destructor cannot have a return value
- A destructor can take no arguments, which means it cannot be overloaded.

A destructor is a special-purpose member function with no parameter and is designed to clean up and recycle an object.

Destructor Declaration

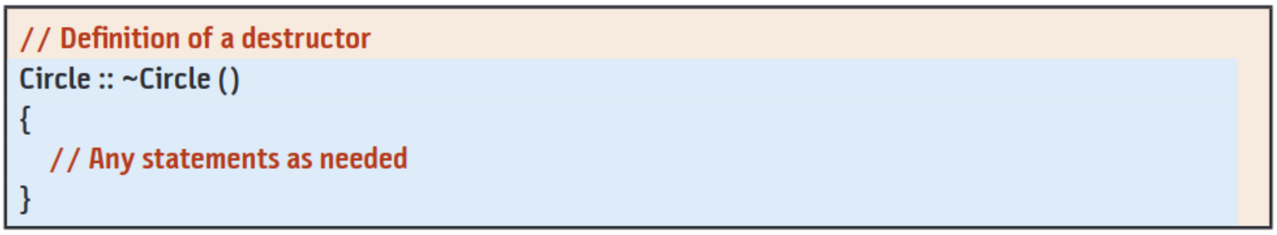
```
class Circle
{
   ...
   public:
        ...
        ~Circle ();                              // Destructor
}
```



Object construction and destruction for a class type

Destructor Definition

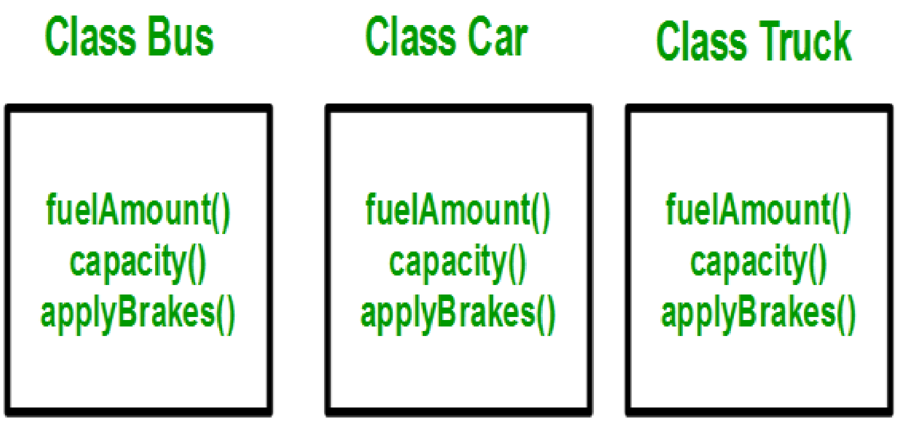
The definition of a destructor is similar to the definition of the other three member functions, but it must have a tilde (~) in front of the first name. A destructor should be public like all constructors.



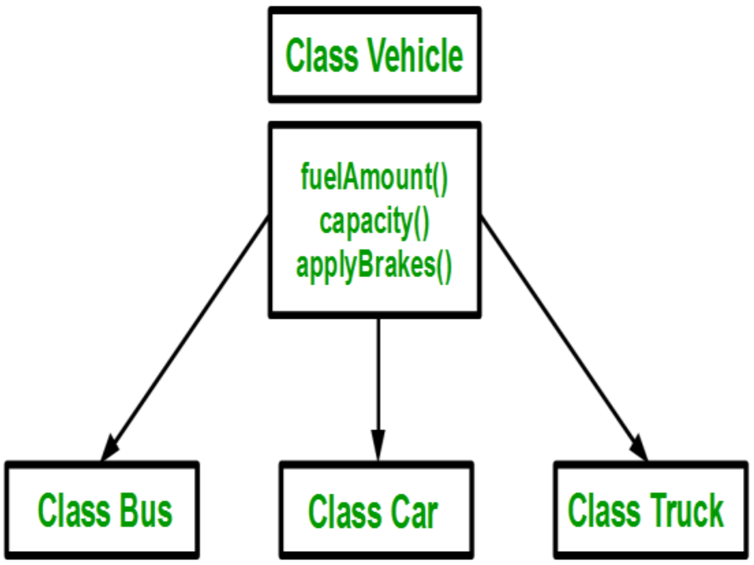
```
// Definition of a destructor
Circle :: ~Circle ()
{
   // Any statements as needed
}
```

INHERITANCE : 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.

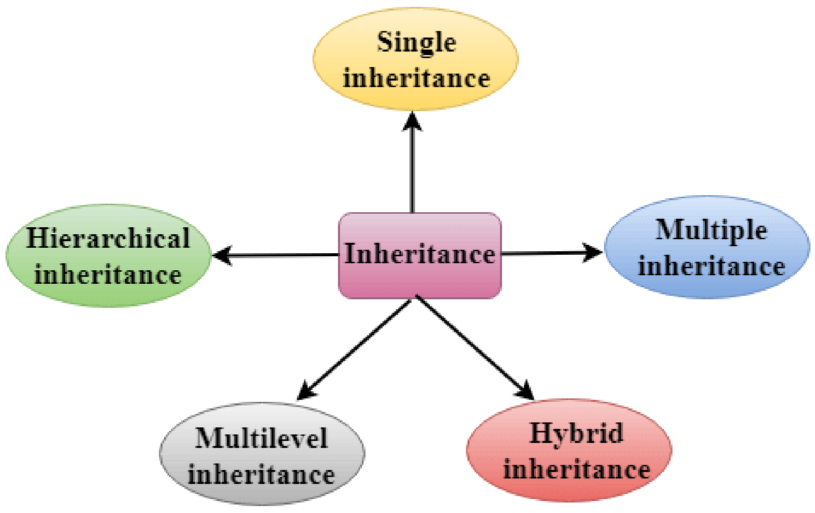
SUB CLASS: The class that inherits properties from another class is called Sub class or Derived Class.
SUPER CLASS: The class whose properties are inherited by sub class is called Base Class or Super class.
Let’s consider a group of vehicles named Bus, Car and Truck. The methods like fuelAmount(), applyBrakes() and capacity() will be the same for all three classes.



Using Inheritance:



TYPES OF INHERITANCE:



IMPLEMENTING INHERITANCE:

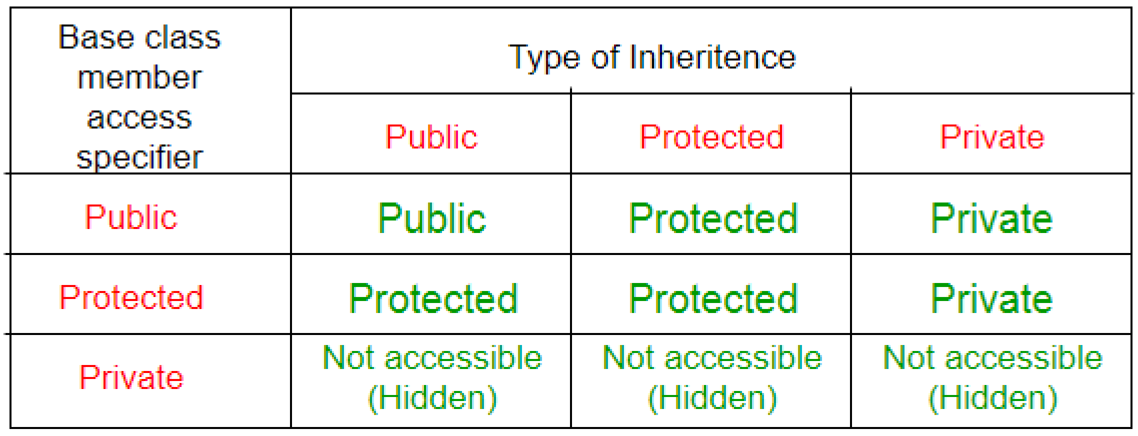
Syntax:

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

Subclass_name – It is the name of the sub class.

- Access_mode – It is the mode in which you want to inherit this sub class such as Private/Public/Protected.
- Base_class_name - It is the name of the base class from which you want to inherit the sub class.

MODES OF INHERITANCE:



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)

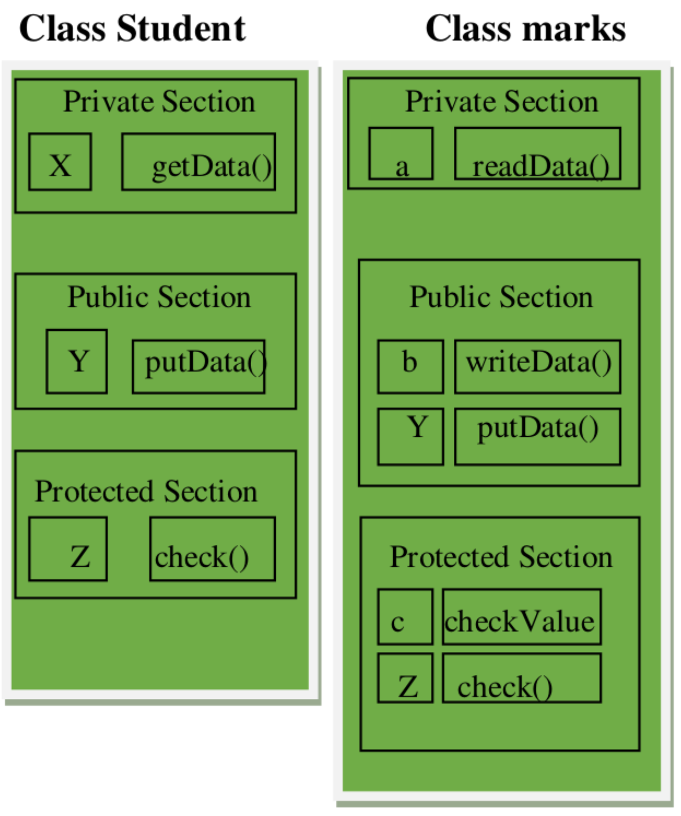
PROGRAM:

```
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
};
```

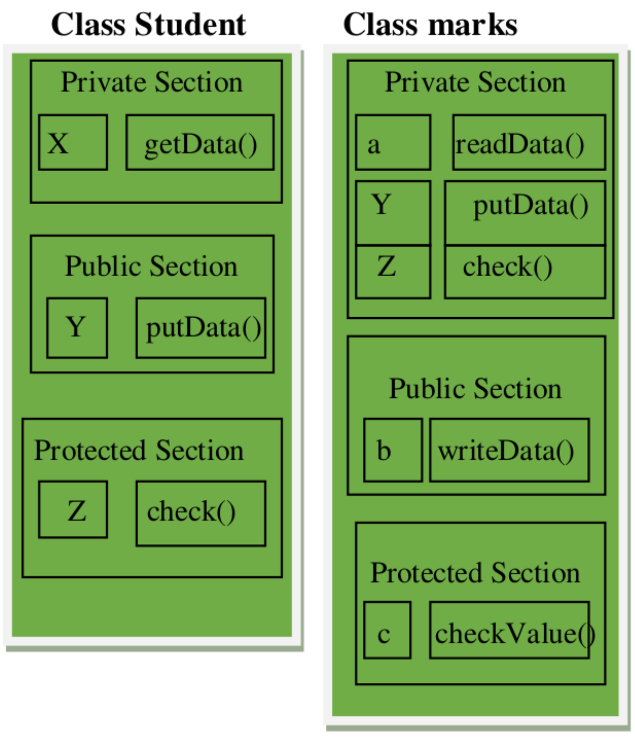
PUBLIC VISIBILITY MODE:

```
class student // base class
{
        private :
                int x; // base class private
members
                void getdata ( );
        public: //base class public members
                int y;
                void putdata();
        protected: //base class public members
                int z;
                void check ( );
};


class marks : public student // class marks
derived class
{
        private :
                int a ;
                void readdata ( );
        public :
                int b;
                void writedata ( );
```



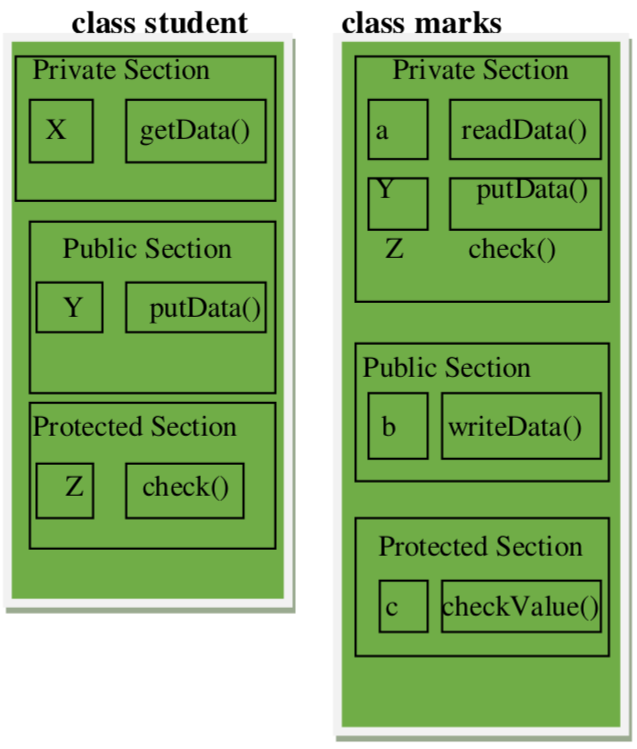
```
          protected :
                    int c;
                    void checkvalue ( );
};
```

PRIVATE VISIBILITY MODE:

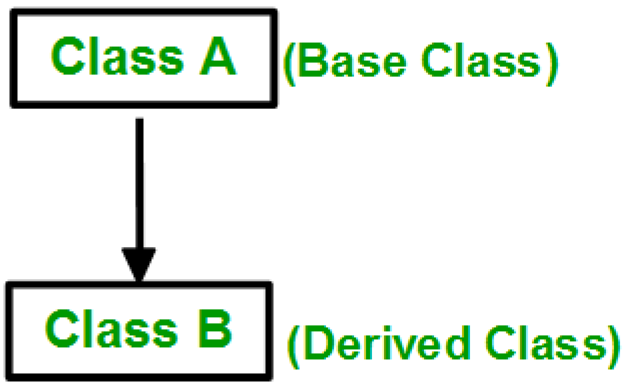


PROTECTED VISIBILITY MODE:

A member declared as protected is accessible by the member functions of the class and its derived classes. It cannot be accessed by the member functions other than these classes.



SINGLE INHERITANCE:



SYNTAX:

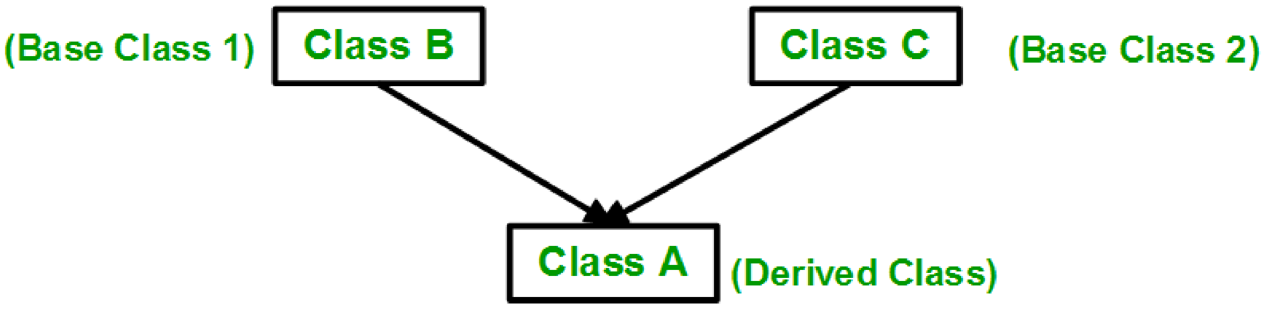
```
class subclass_name : access_mode
base_class
{
        //body of subclass
};
```

PROGRAM:

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

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

};

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

OUTPUT:
```
This is a Vehicle
```

MULTIPLE INHERITANCE:



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

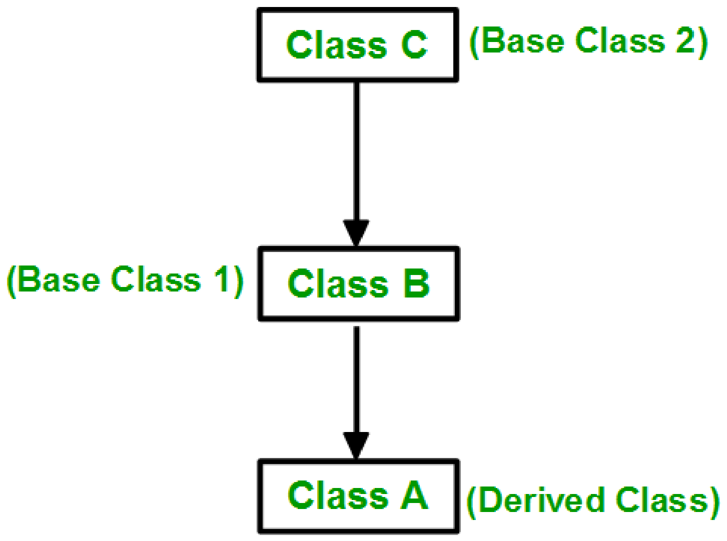
PROGRAM:
```
#include <iostream>
using namespace std;

// 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
```

MULTI-LEVEL INHERITANCE:



PROGRAM:

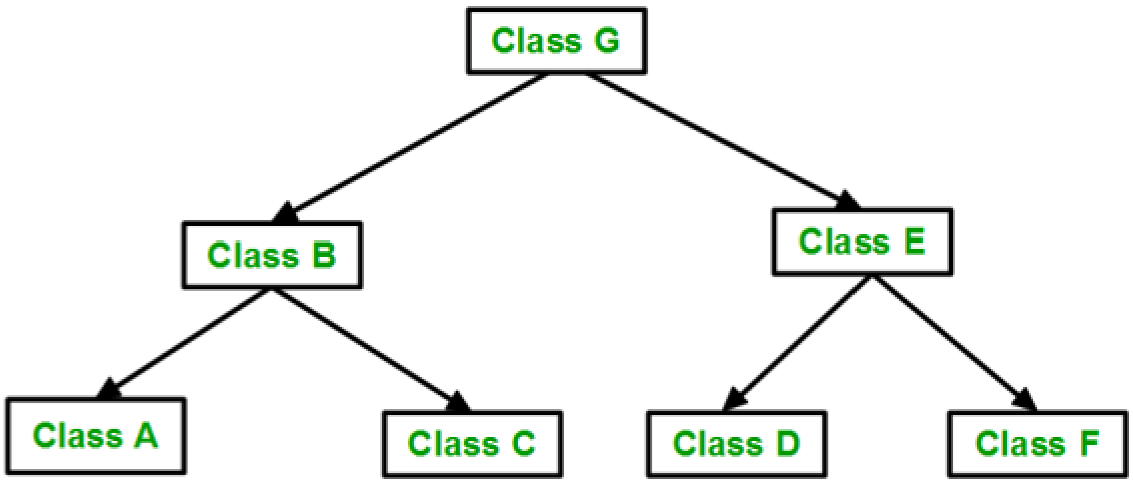
```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
public:
Vehicle()
{
cout << "This is a Vehicle" << endl;
}
};

// 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;
}
```

OUTPUT:

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

HIERACHICAL INHERITANCE:



PROGRAM :

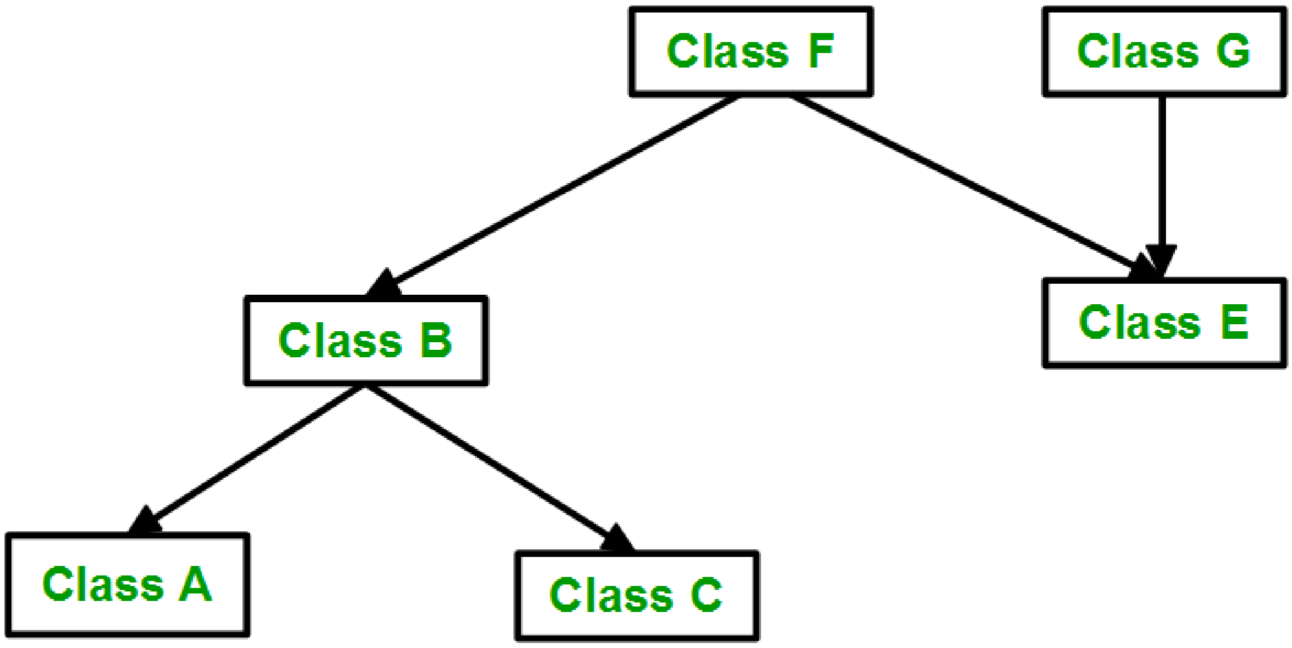
```
#include <iostream>
```

```
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
```

HYBRID INHERITANCE:



PROGRAM:

```
#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
{

};
```

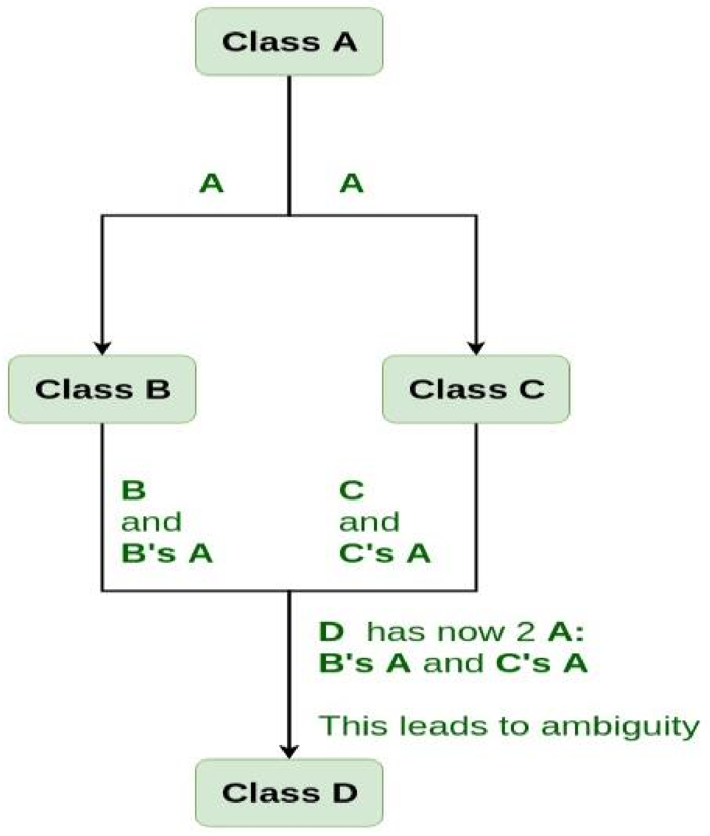
```
// second sub class
class Bus: public Vehicle, public Fare
{

};


// 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
```

VIRTUAL BASE CLASS: 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.

Let's consider the following situation.


Class A
A
A
Class B
Class C
B and B's A
C and C's A
D has now 2 A: B's A and C's A
This leads to ambiguity
Class D