Object-Oriented Programming (OOP)

Week – 06

Oct 06 2020

Instructor: **Talha Khan**

Email: talha.khan@nu.edu.pk

Object Relationship

- is-a relationship

- has-a relationship

has-a **Relationship**

- In a *has-a* relationship, an object contains one or more objects of other classes as members

A car ***has a*** steering

An office ***has a*** department

has-a **Relationship**

```
class Office
{
    Department d;
    // any other  members
};
```

is-a Relationship

- Sometimes, one class *is* an extension of another class

A car ***is a*** vehicle
Cricket ***is a*** sport

is-a Relationship

- The extended (or child) class contains all the features of its base (or parent) class, and may additionally have some unique features of its own

- The key idea behind inheritance

Inheritance

- Capability of a class to derive properties and characteristics from another class.

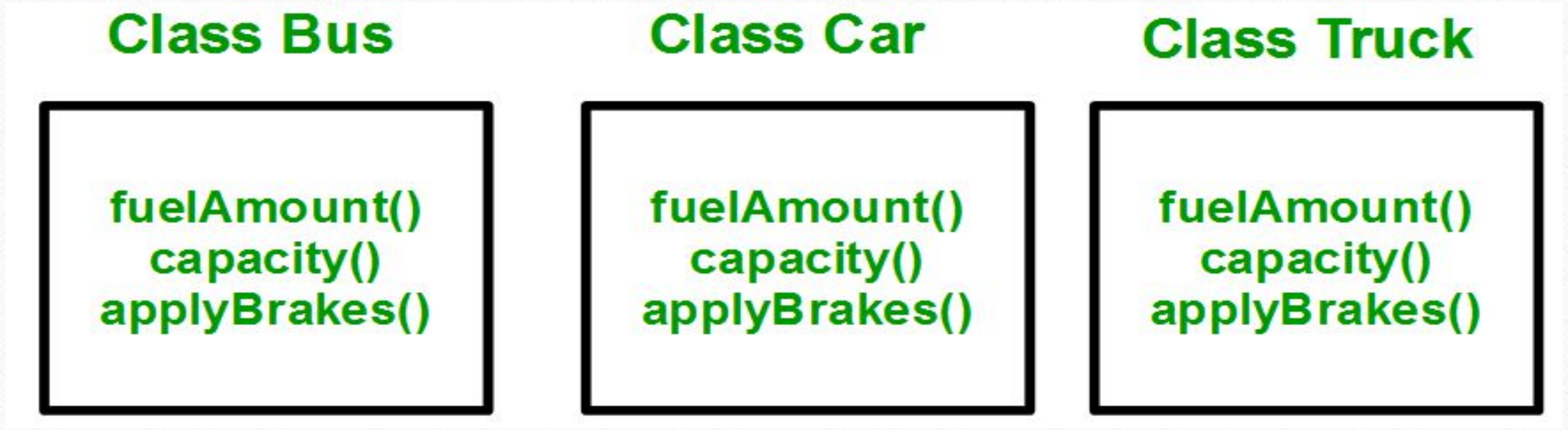
- One of the most important feature of Object Oriented Programming

Concepts

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class or Child Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class or Parent Class.

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.



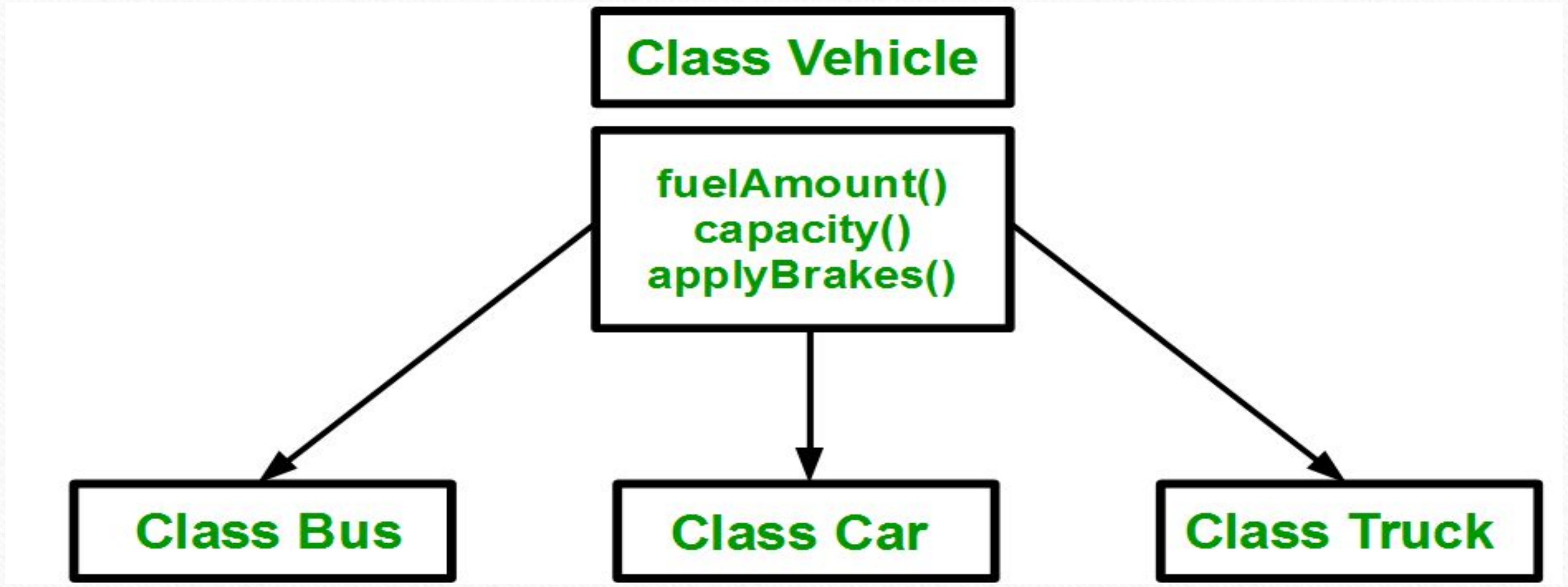
Why and when to use inheritance?

- 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.

Why and when to use inheritance?

- 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.

Why and when to use inheritance?



Implementing inheritance in C++

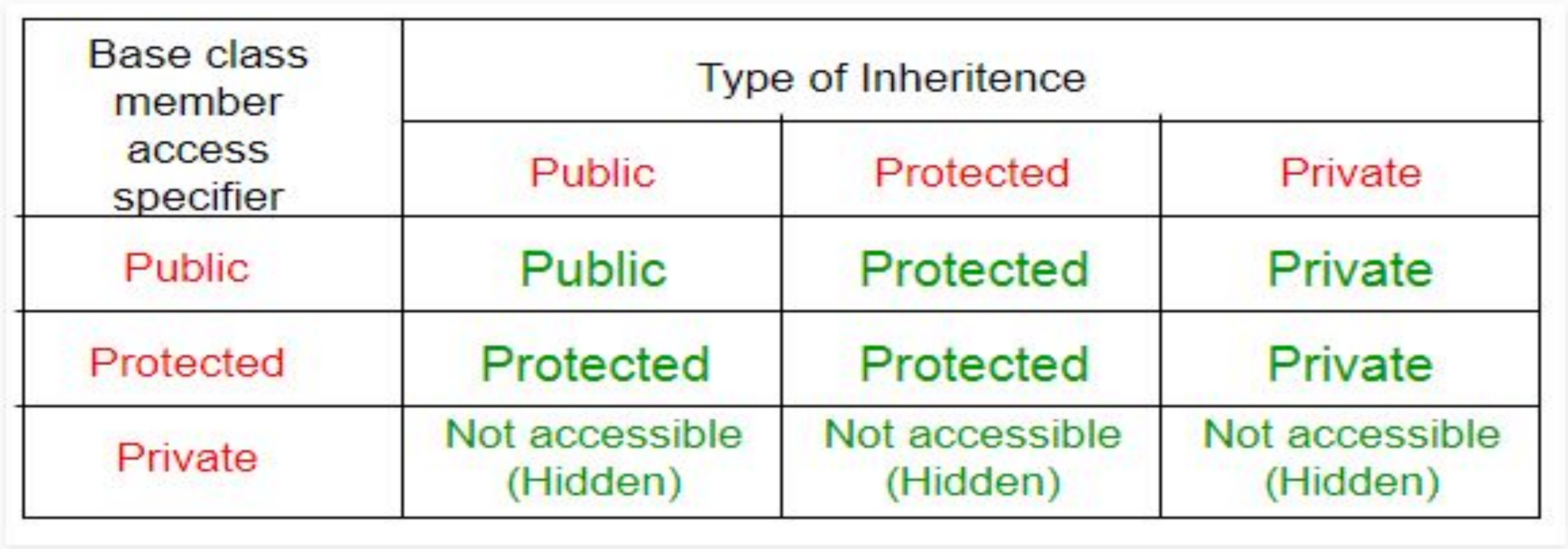
- Syntax:

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

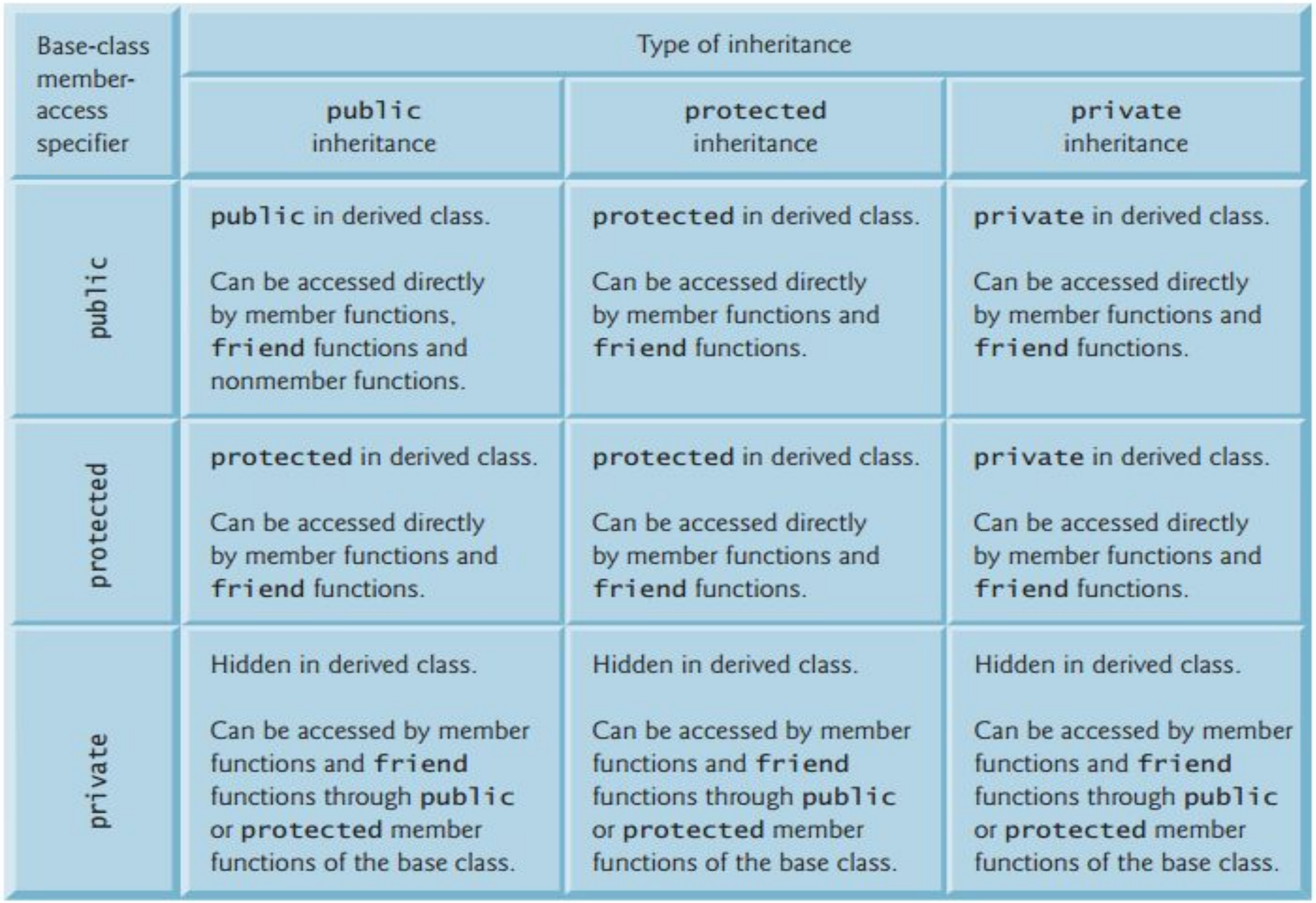
Modes of Inheritance

- **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.
- **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.
- **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.

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)



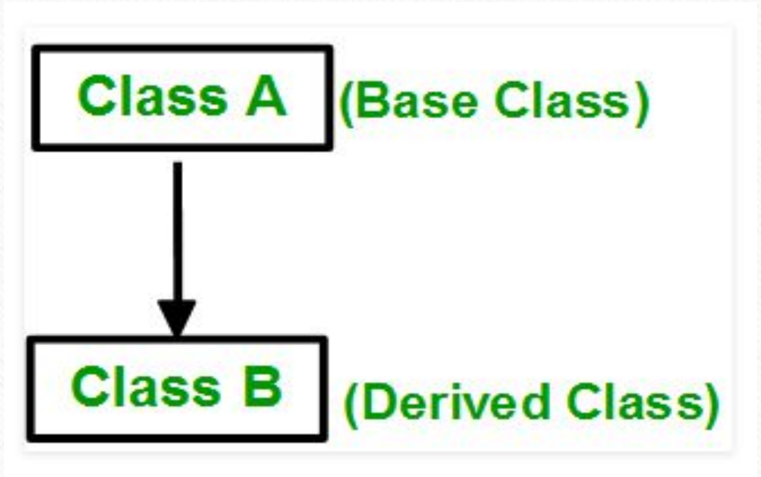
Base-class member-access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
public	**public** in derived class. Can be accessed directly by member functions, **friend** functions and nonmember functions.	**protected** in derived class. Can be accessed directly by member functions and **friend** functions.	**private** in derived class. Can be accessed directly by member functions and **friend** functions.
protected	**protected** in derived class. Can be accessed directly by member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by member functions and **friend** functions.	**private** in derived class. Can be accessed directly by member functions and **friend** functions.
private	Hidden in derived class. Can be accessed by member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by member functions and **friend** functions through **public** or **protected** member functions of the base class.

Types of Inheritance in C++

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid (Virtual) Inheritance

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.



Single Inheritance (Continue..)

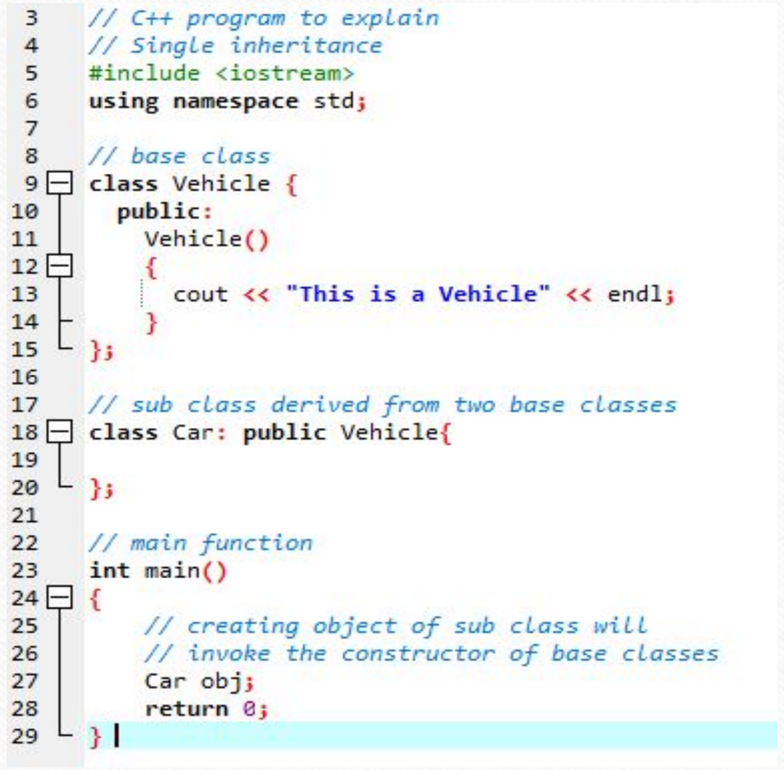
- Syntax:

class subclass_name : access_mode base_class

{

//body of subclass

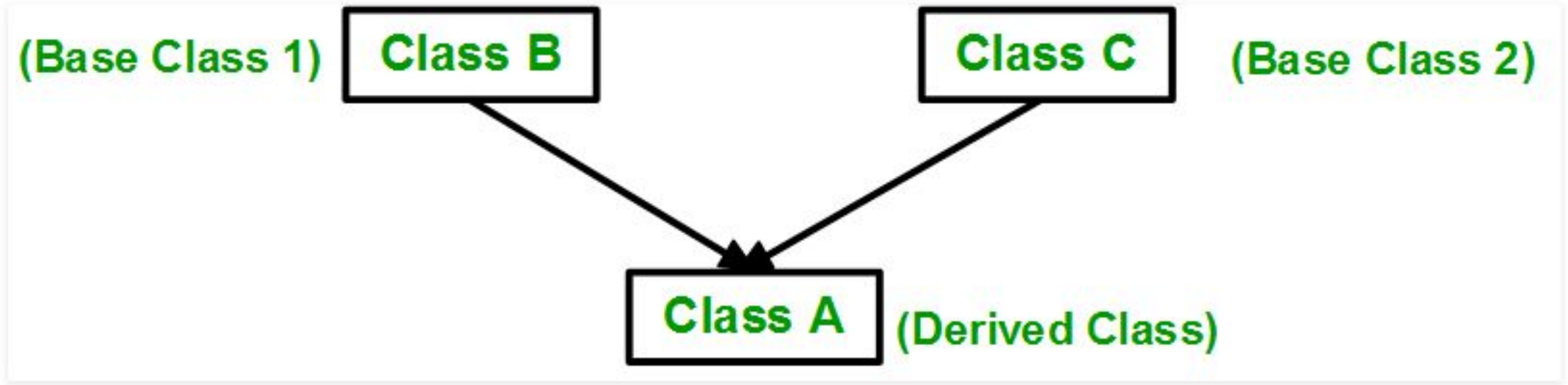
};



```
// 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;
}
```

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**.



Multiple Inheritance (Continue..)

- Syntax:

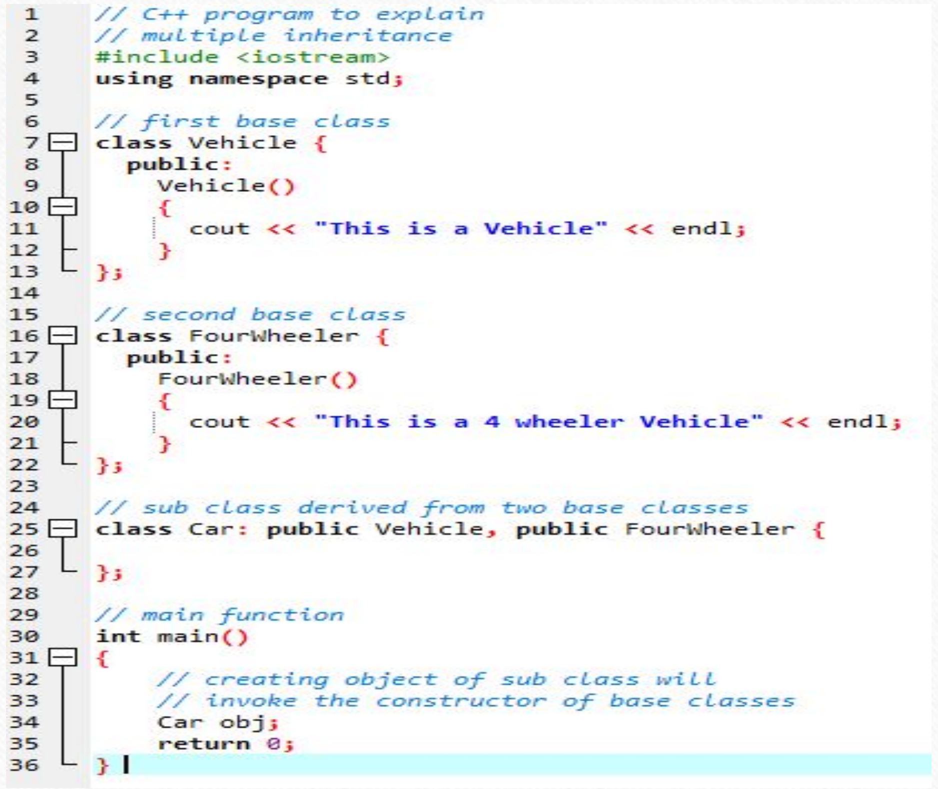
class subclass_name : access_mode base_class1,

access_mode base_class2, ...

{

 //body of subclass

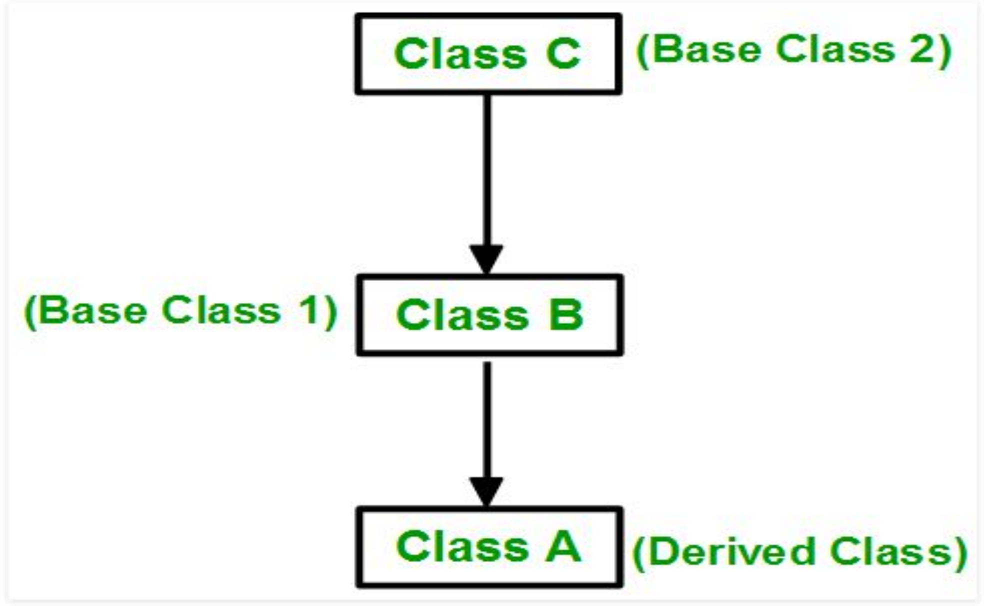
};



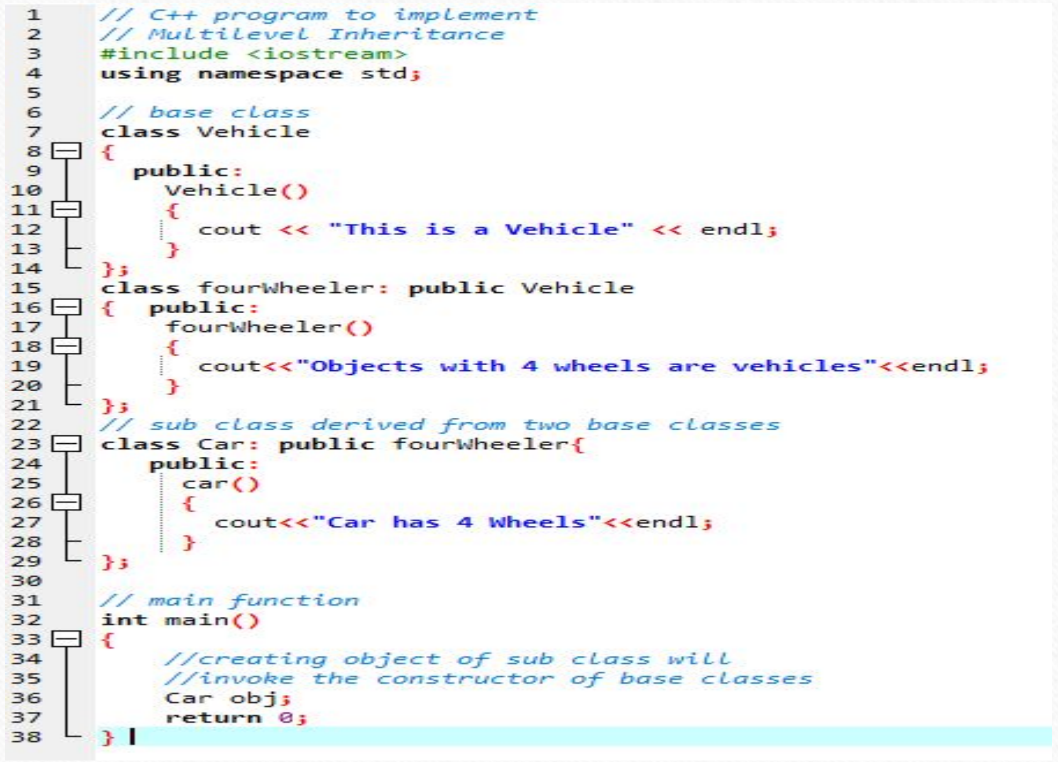
```cpp
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// 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;
}
```

Multilevel Inheritance

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



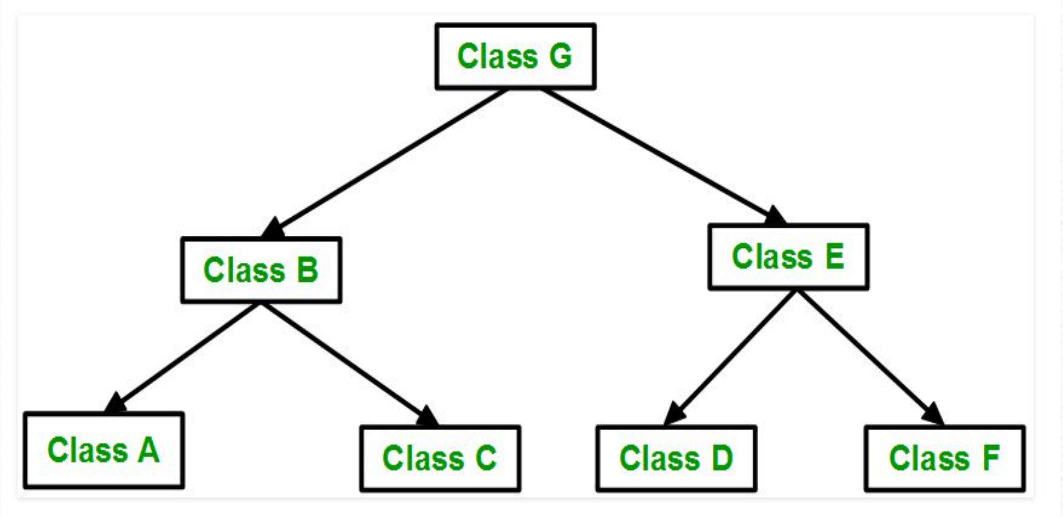
Multilevel Inheritance (Continue..)



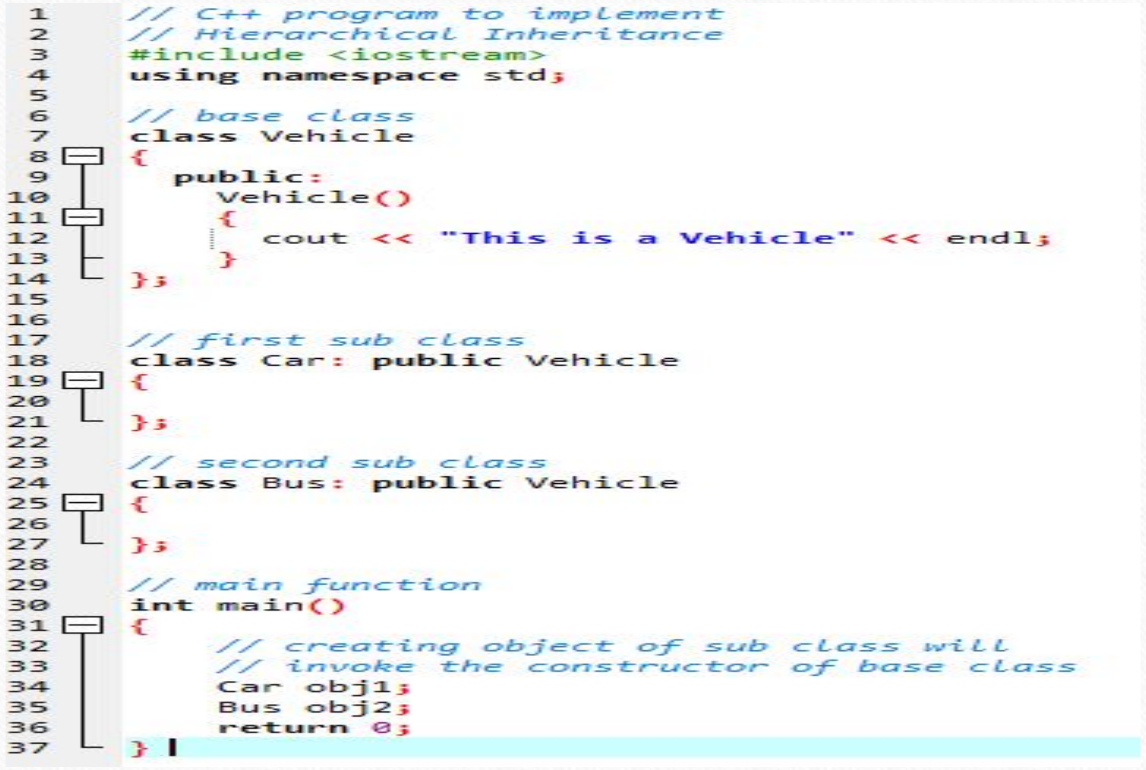
```cpp
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// 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;
}
```

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.



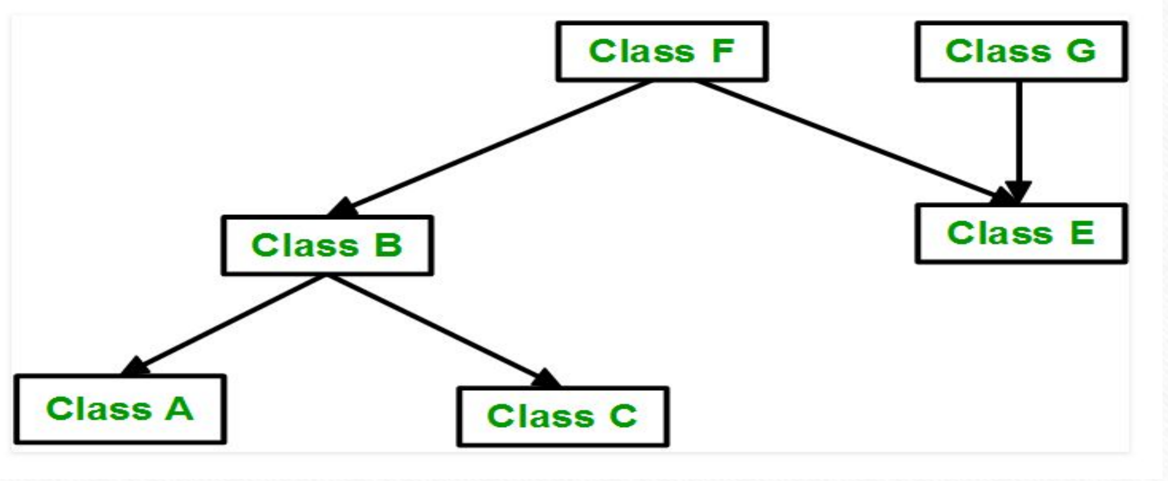
Hierarchical Inheritance (Continue..)



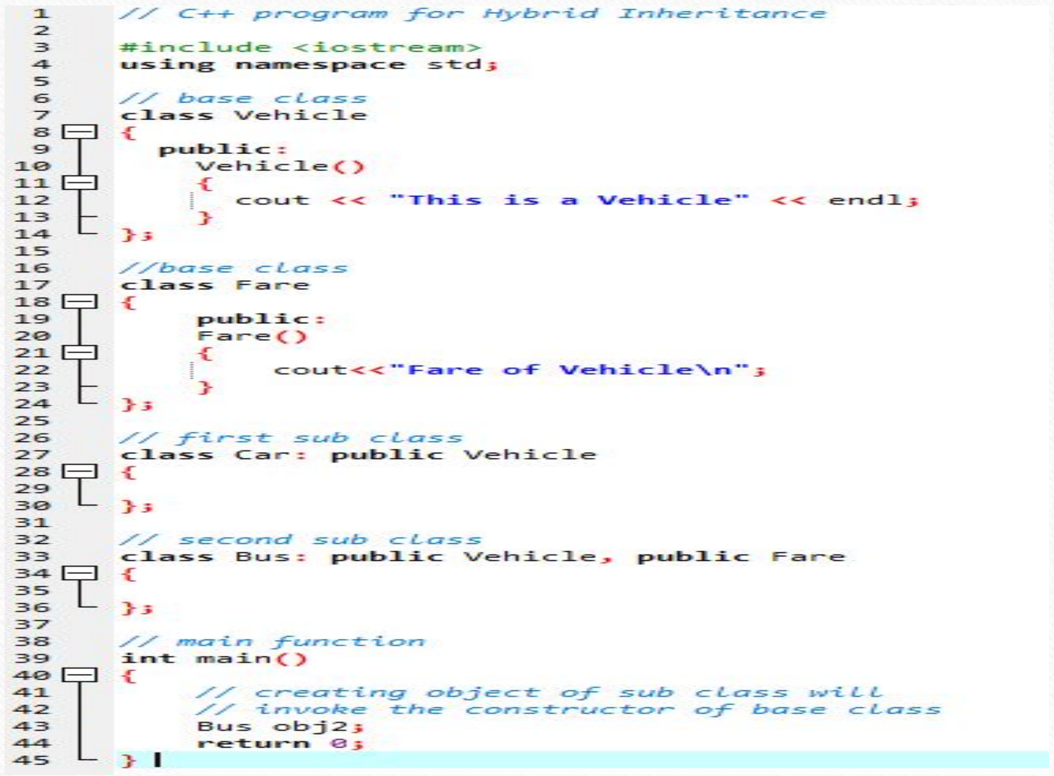
```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// 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;
}
```

Hybrid (Virtual) Inheritance

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



Hybrid Inheritance (Continue..)

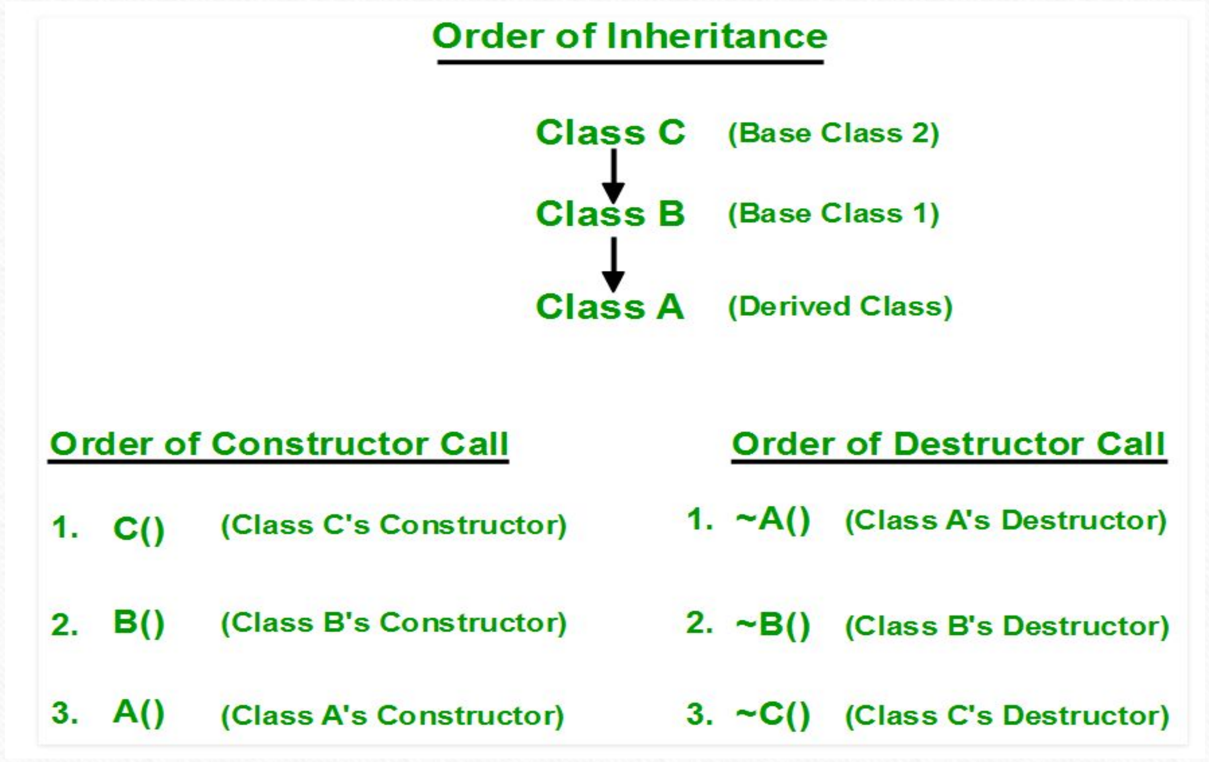


```
// C++ program for Hybrid Inheritance

#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;
}
```

Order of constructor and Destructor call

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
  If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.

Order of constructor and Destructor call



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)

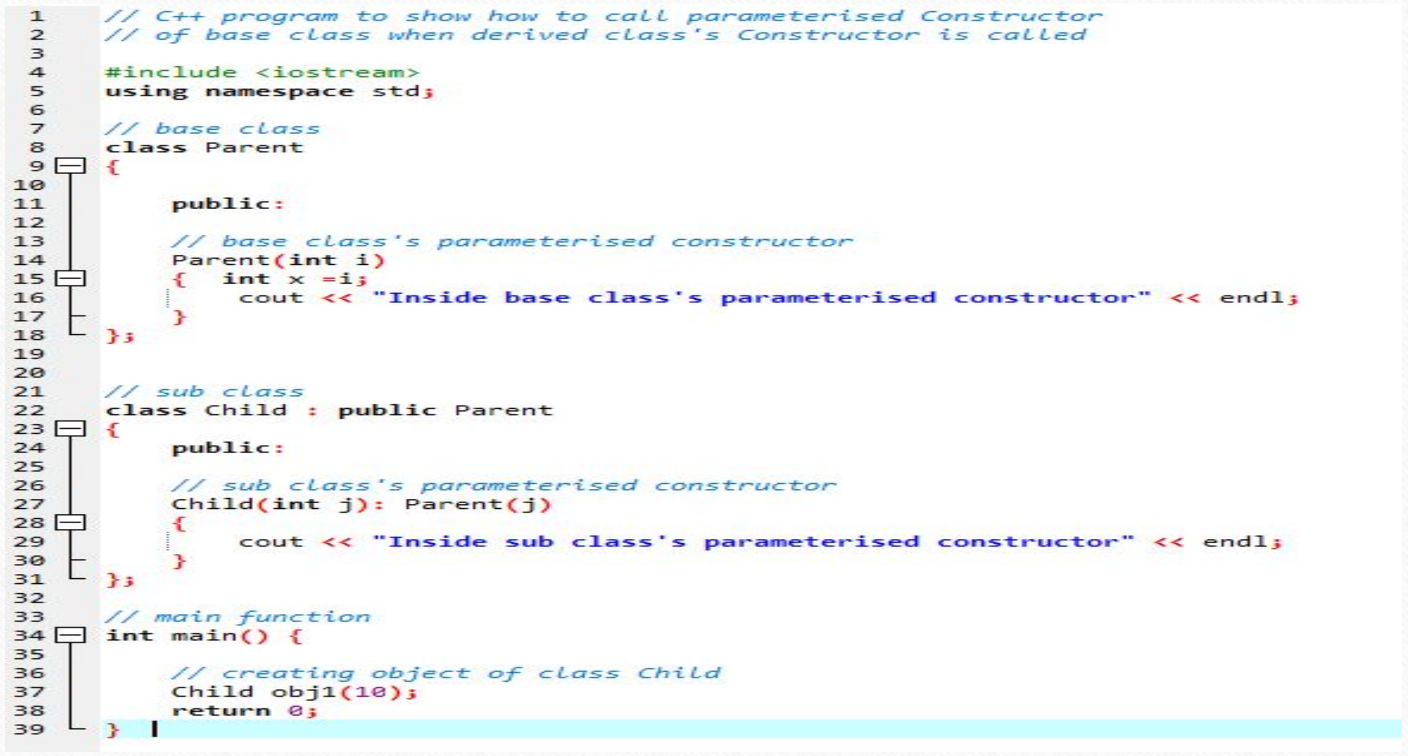
Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

Concept: Calling parameterized constructor of base class in derived class constructor!

- To call the parameterized constructor of base class when derived class’s parameterized constructor is called, you have to explicitly specify the base class’s parameterized constructor in derived class

Concept: Calling parameterized constructor of base class in derived class constructor!



```
// C++ program to show how to call parameterised Constructor
// of base class when derived class's Constructor is called

#include <iostream>
using namespace std;

// base class
class Parent
{

    public:

    // base class's parameterised constructor
    Parent(int i)
    {   int x =i;
        cout << "Inside base class's parameterised constructor" << endl;
    }
};


// sub class
class Child : public Parent
{
    public:

    // sub class's parameterised constructor
    Child(int j): Parent(j)
    {
        cout << "Inside sub class's parameterised constructor" << endl;
    }
};

// main function
int main() {

    // creating object of class Child
    Child obj1(10);
    return 0;
}
```

Important Points

- Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
- To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.
- The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.