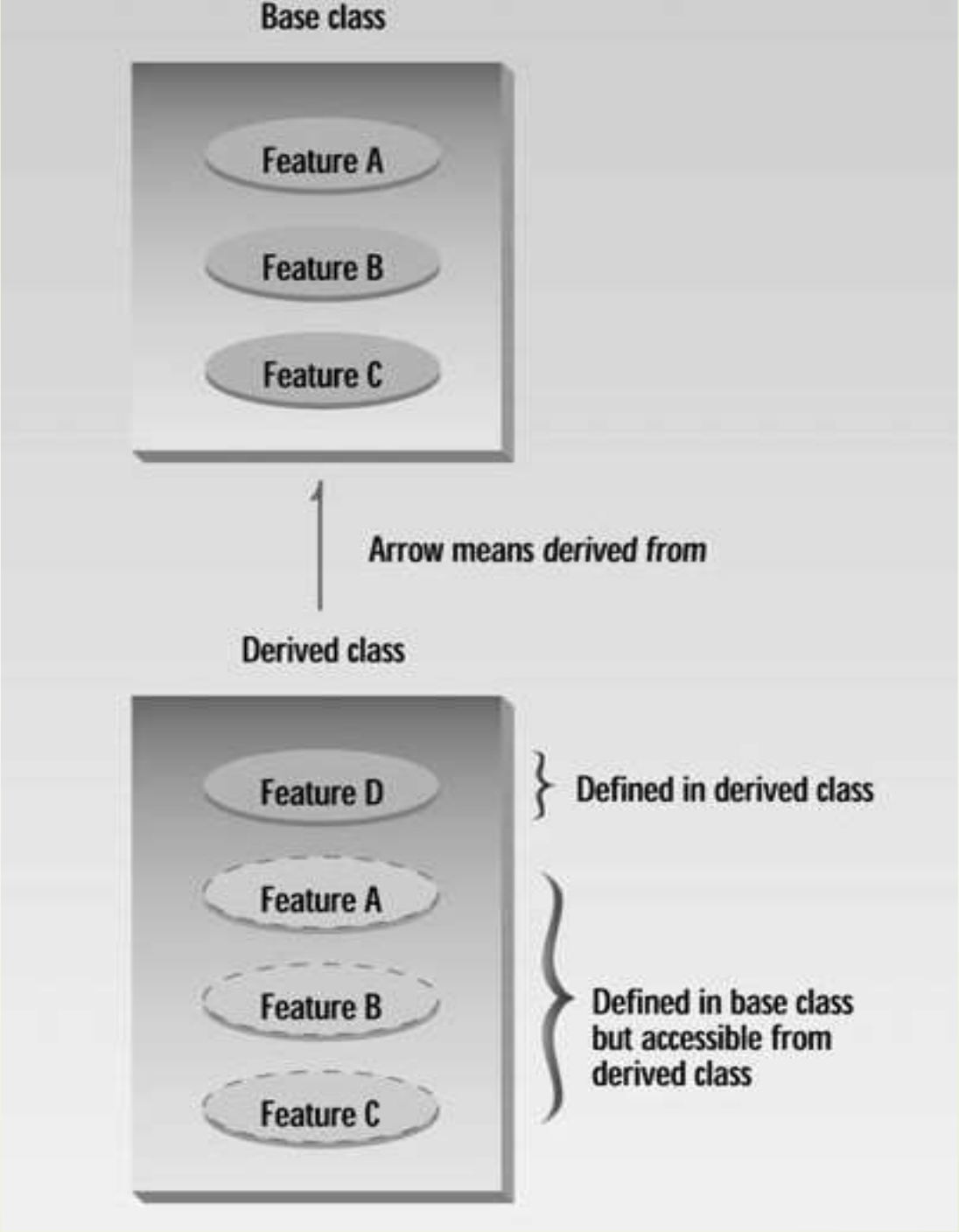
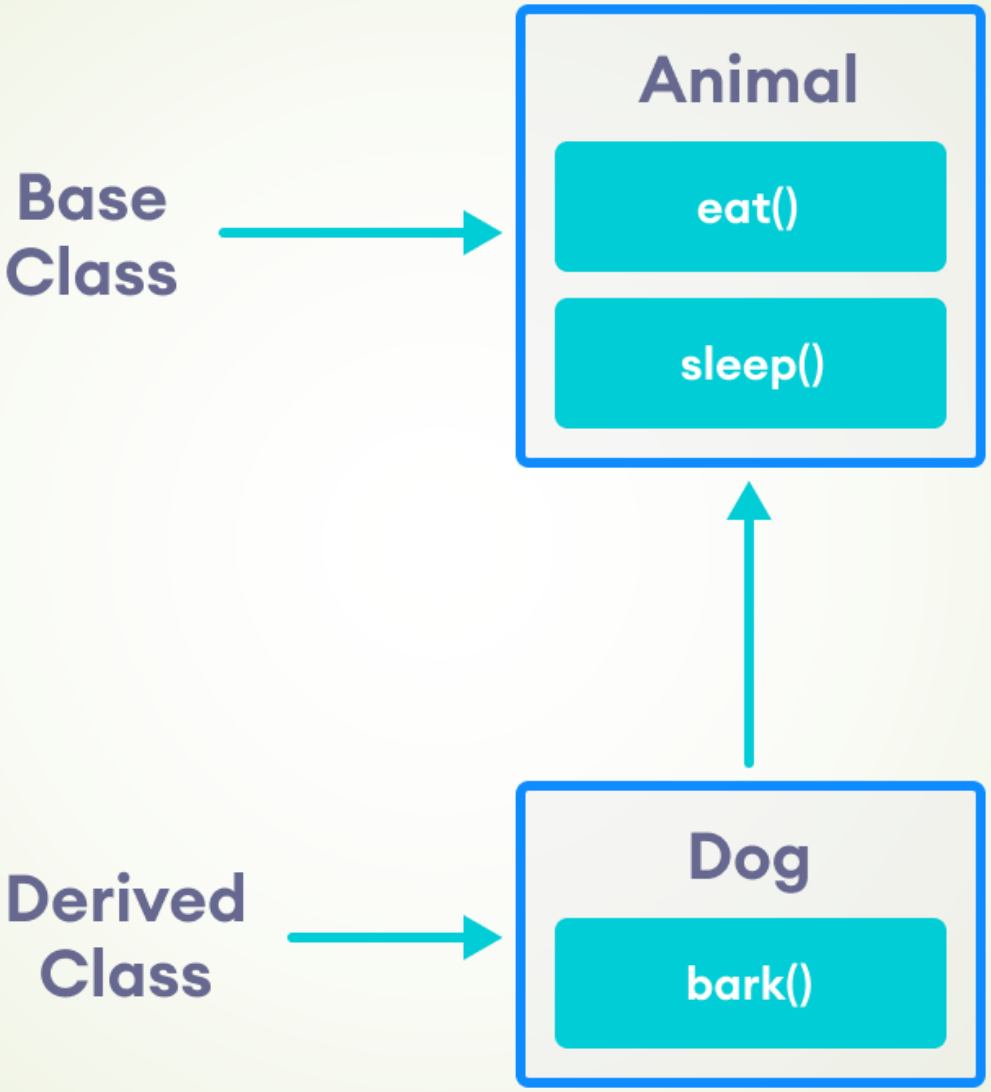
# Inheritance

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- **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.

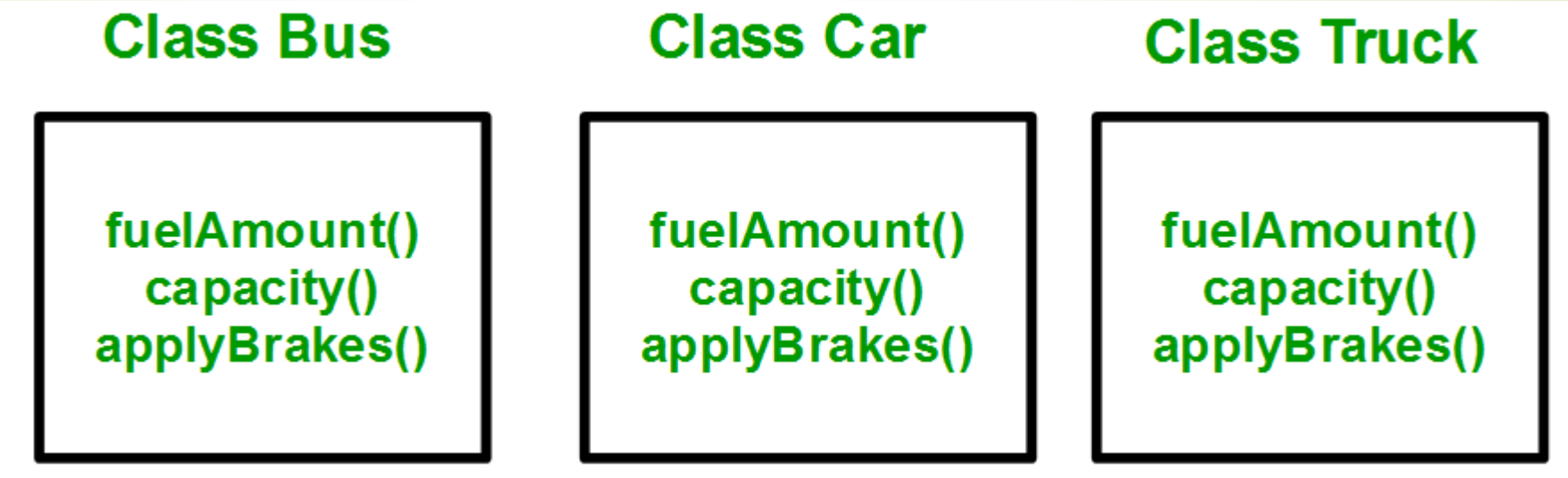
```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```


Base class
Feature A
Feature B
Feature C
Arrow means derived from
Derived class
Feature D
Feature A
Feature B
Feature C
Defined in derived class
Defined in base class
but accessible from
derived class

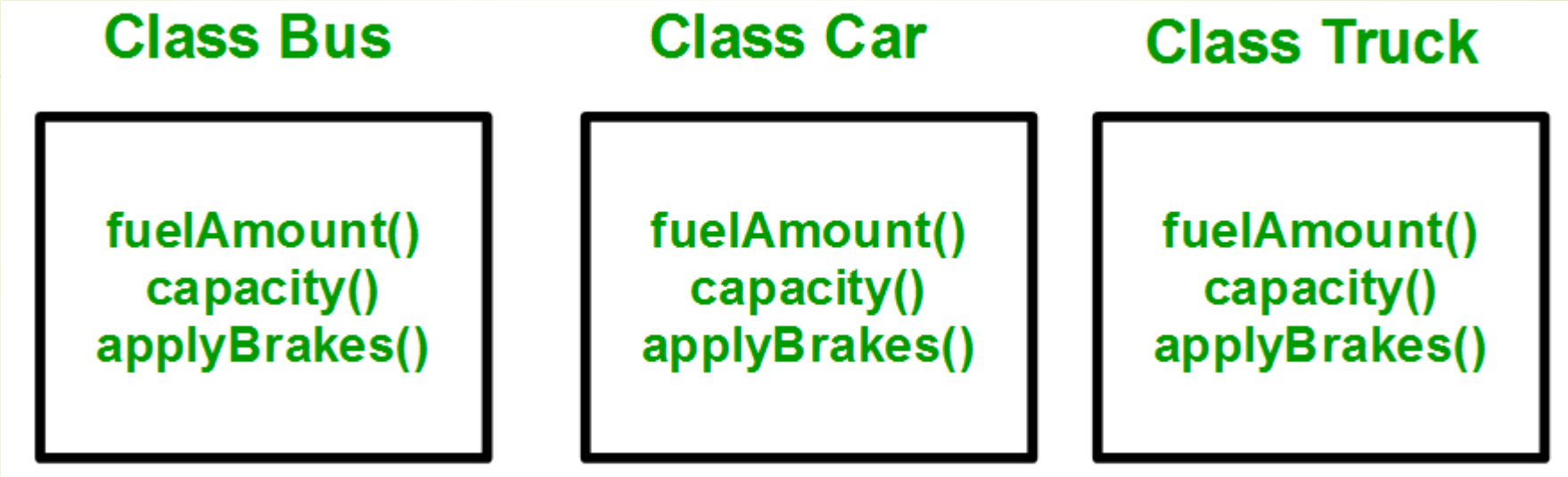

Base Class
Animal
eat()
sleep()
Derived Class
Dog
bark()

# 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:

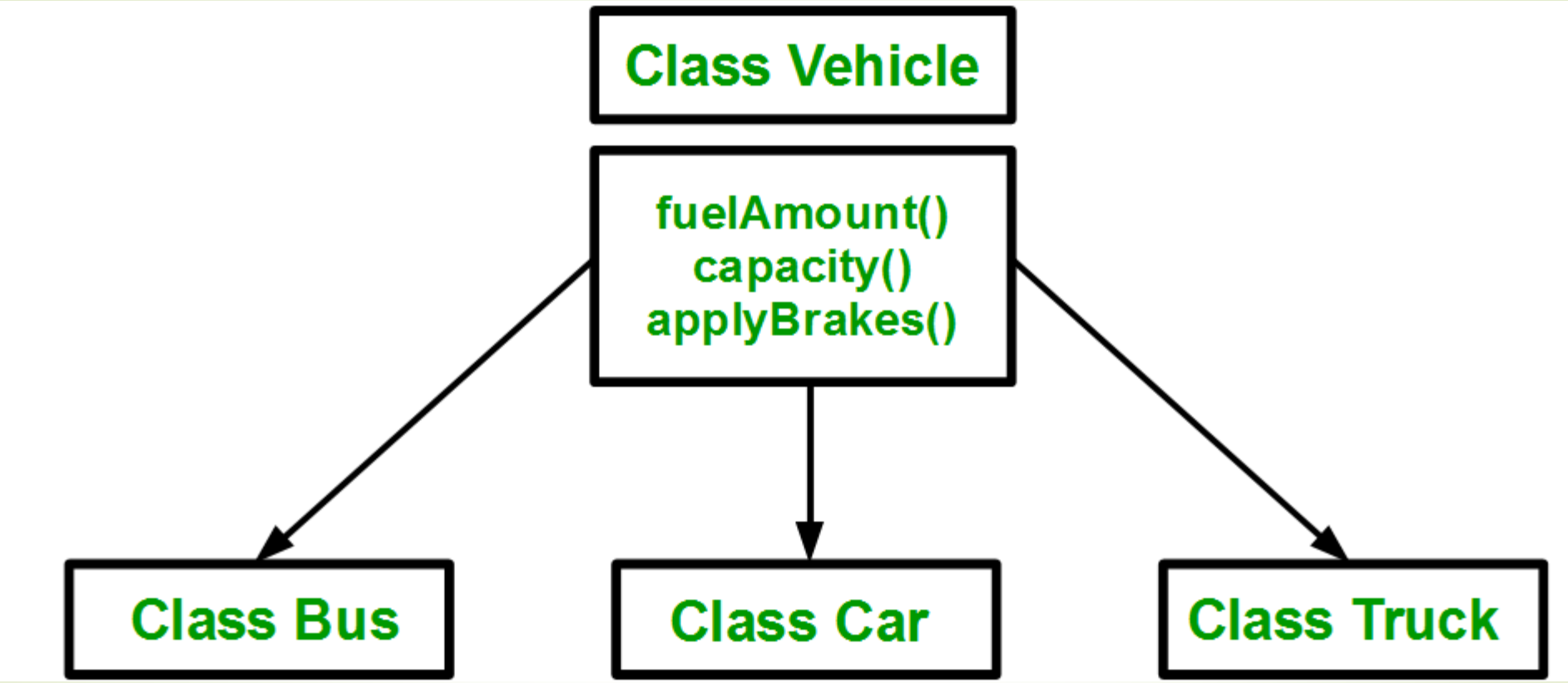


# Why and when to use inheritance?




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.

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.




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).

- Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.
- Here are some examples:
- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.

# Implementing inheritance in C++

```
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.

```
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;
   }
```

```
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.

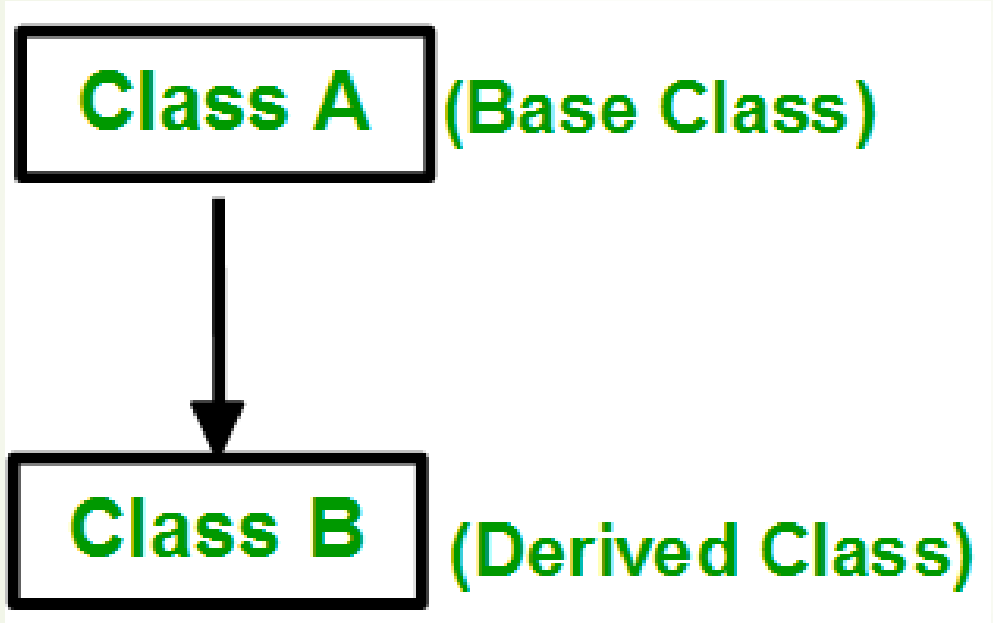
The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

```
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
};
```

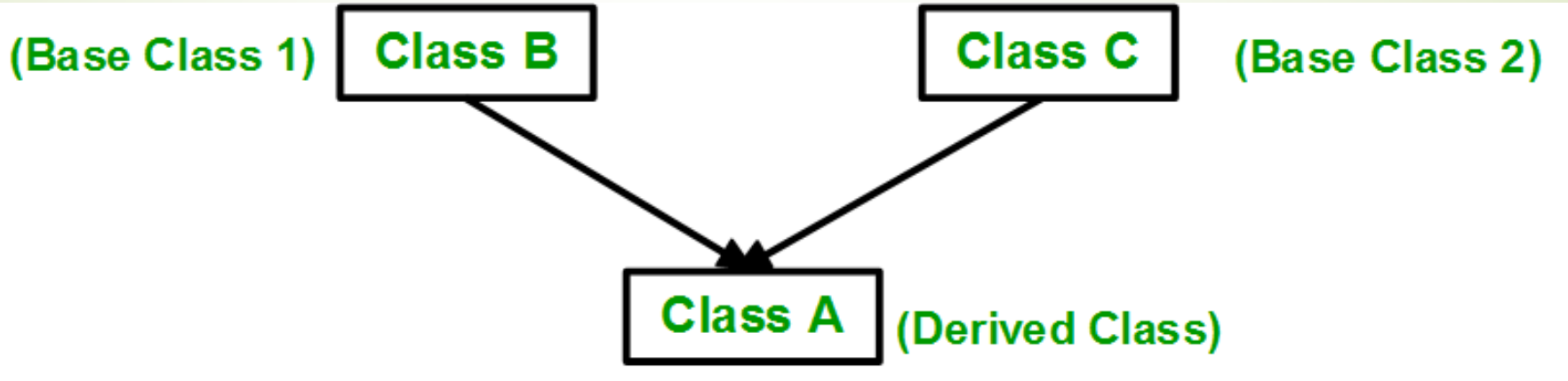
| Base class member access specifier | Type of Inheritence | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

```
class Declaration
{
        private:
        int a;
        public:
        int b;
        protected:
        int c;
        public:

        void show()
        {
                a=10;
                b=20;
                c=30;

                //Every members can be access
                here, same class
                cout<<"\nAccessing variable
                within the class"<<endl;

                cout<<"Value of a: "<<a<<endl;
                cout<<"Value of b: "<<b<<endl;
                cout<<"Value of c: "<<c<<endl;
                }
};

class Sub_class:public Declaration
{
public:
}

void show()
{
        b=5;
        c=6;
        cout<<"\nAccessing variable in sub the class"<<endl;

        //cout<<"Value of a: "<<a<<endl;
        //b is public so it is accessible any where
        cout<<"Value of b: "<<b<<endl;
        //'c' is declared as protected, so it is accessible in sub class
        cout<<"Value of c: "<<c<<endl;
}
};
void main()
{
Declaration d; // create object
d.show();

Sub_class s; // create object
s.show();    // Sub class show() function

cout<<"\nAccessing variable outside the class"<<endl;
//'a' cannot be accessed as it is private
//cout<<"value of a: "<<d.a<<endl;

//'b' is public as can be accessed from any where
cout<<"value of b: "<<d.b<<endl;

//'c' is protected and cannot be accesed here
//cout<<"value of c: "<<d.c<<endl;
```

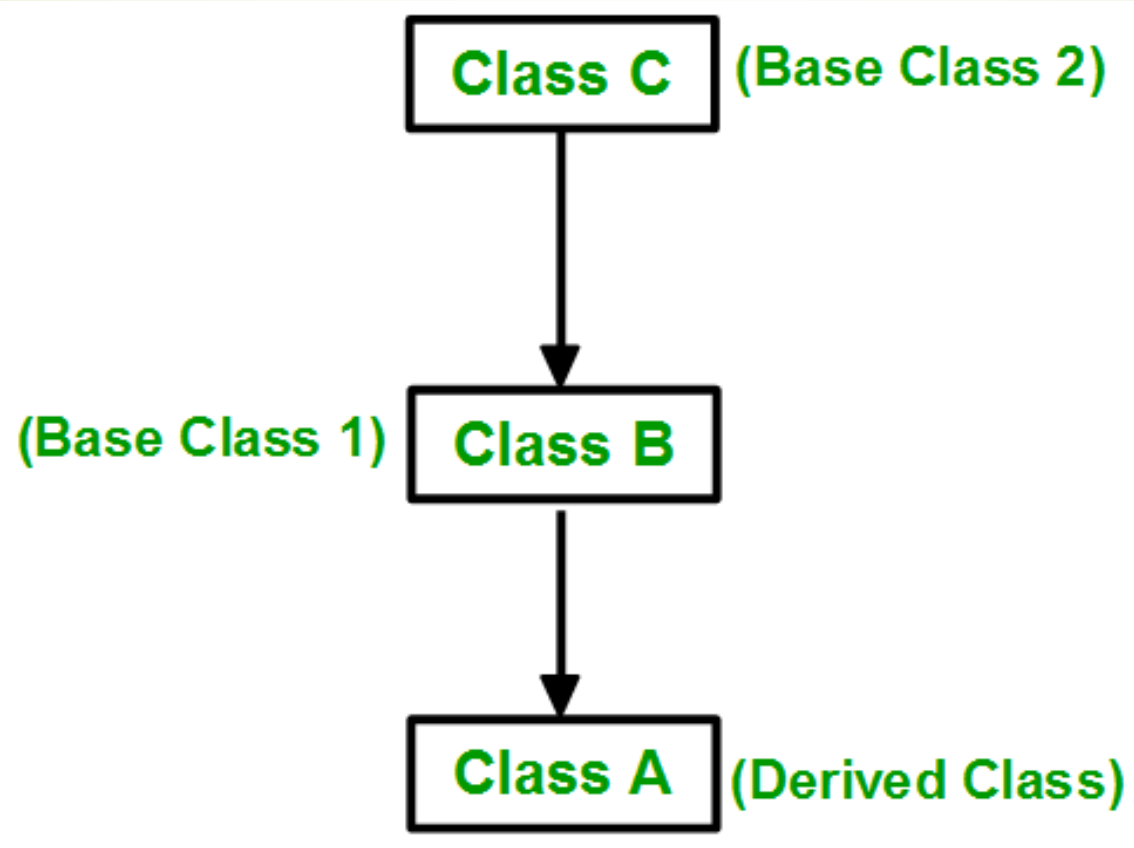
# Single Inheritance



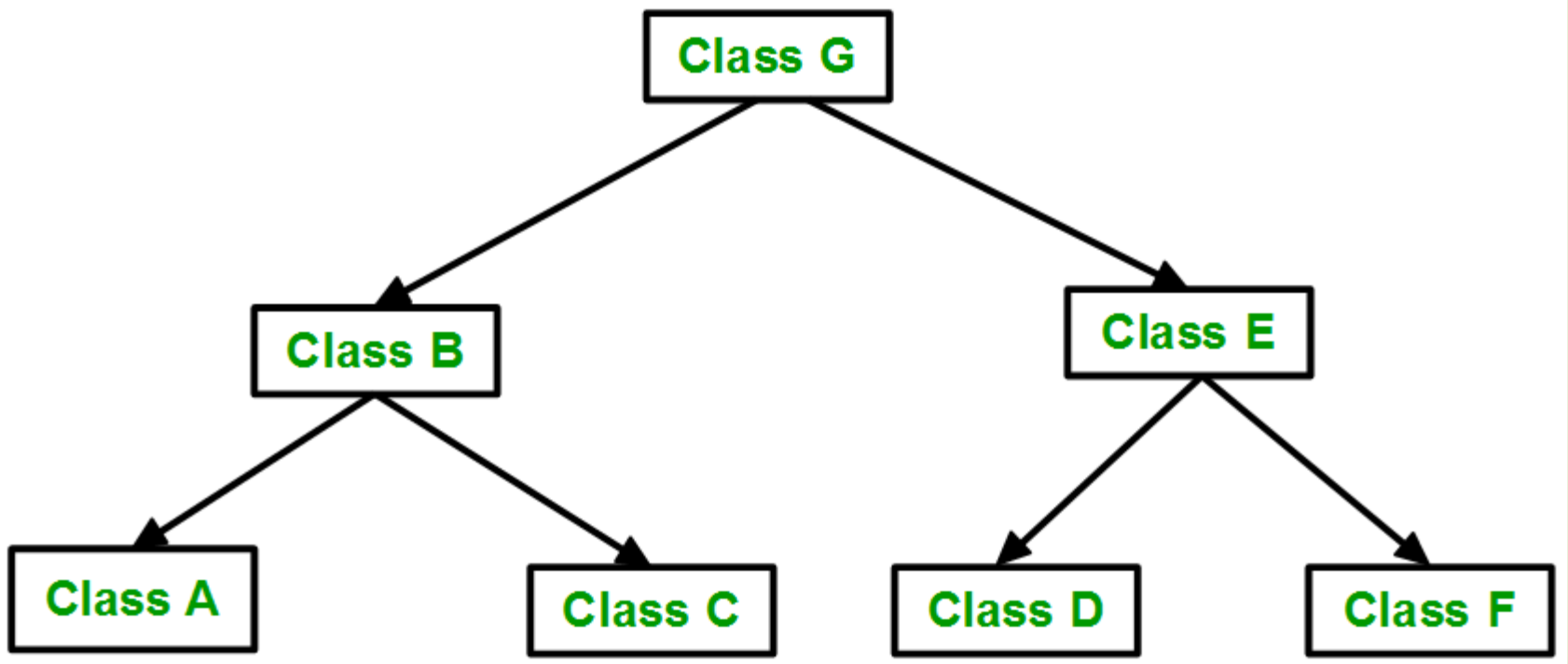
# Multiple Inheritance:



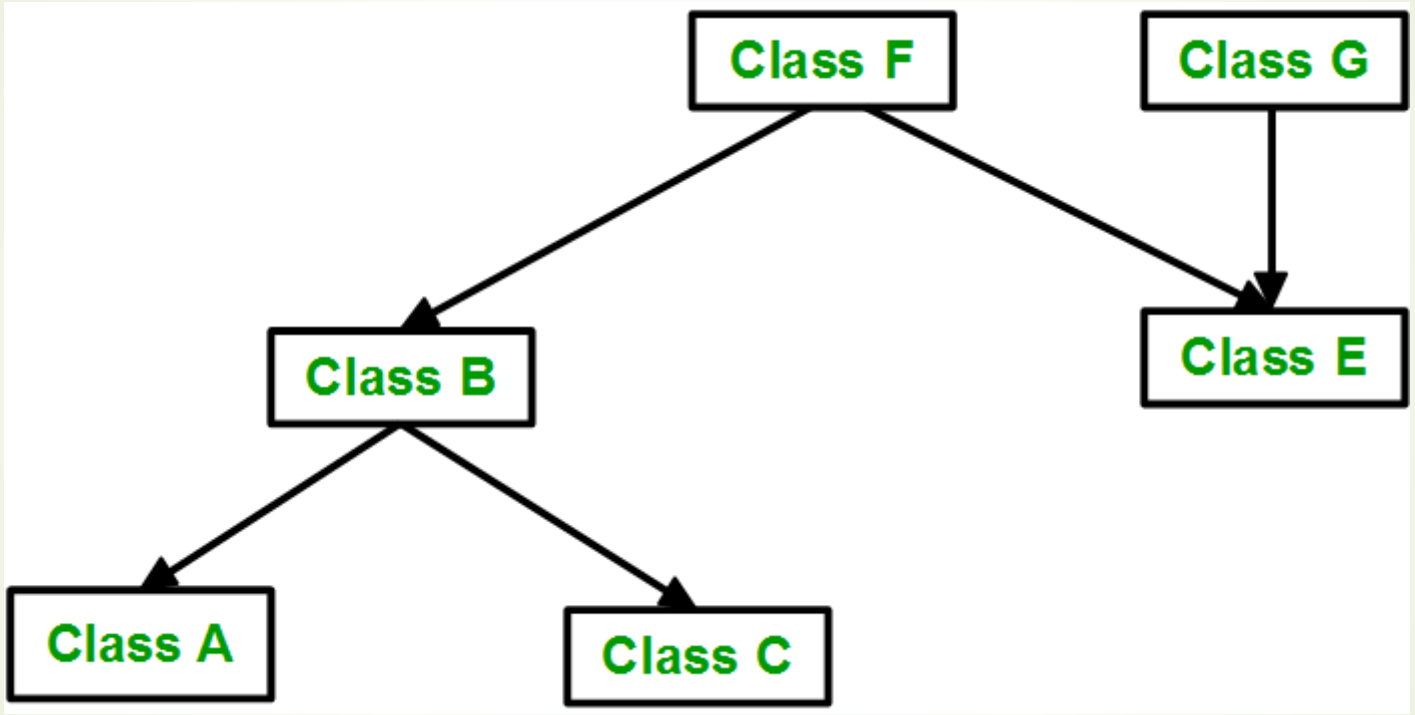
# Multilevel Inheritance



# Hierarchical Inheritance:



# Hybrid (Virtual) Inheritance:



# Multipath inheritance: