5.Inheritance

- Concept of inheritance: defining derived and base classes, Class hierarchies, public, private, and protected derivations
- Types of Inheritance: Single Inheritance, Multilevel Inheritance, Multiple Inheritance, Hierarchical Inheritance, Hybrid Inheritance
- Virtual base class: Function overriding
- Constructors/Destructors in derived classes: Constructors invocation and data members initialization in derived classes
- Member classes: classes within classes

Inheritance:

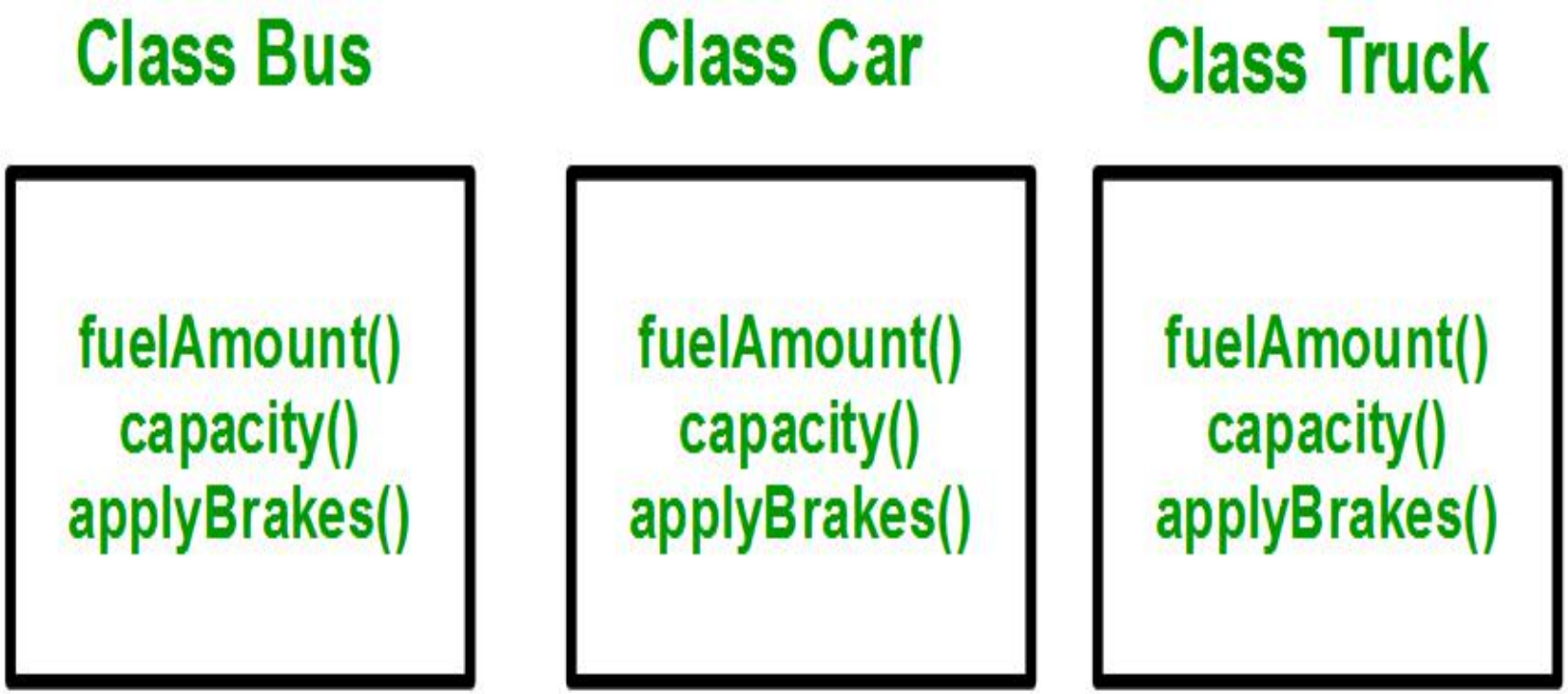
The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

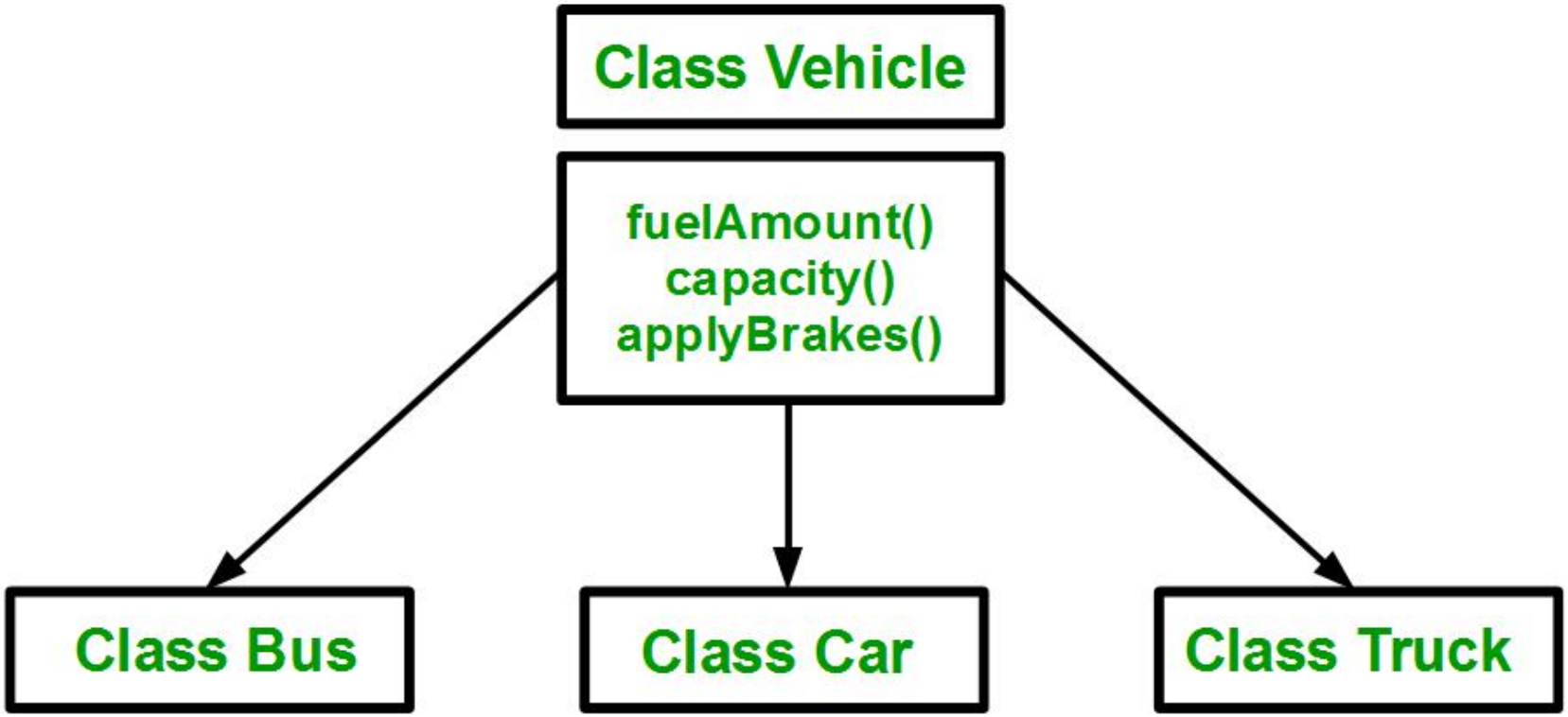
Super Class:The class whose properties are inherited by sub class is called Base Class or Super class.

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:



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 increas re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



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
};
```

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: private member of the base class will never get inherited in the sub class.

// C++ program to demonstrate implementation of Inheritance

```
 #include <iostream>
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;
};
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;
  }
```

Output:

```
Child id is 7
Parent id is 91
```

Explanation

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’.

Purpose of Inheritance

- ✓ Code Reusability
- ✓ Method Overriding (Hence, Runtime Polymorphism.)
- ✓ Use of Virtual Keyword

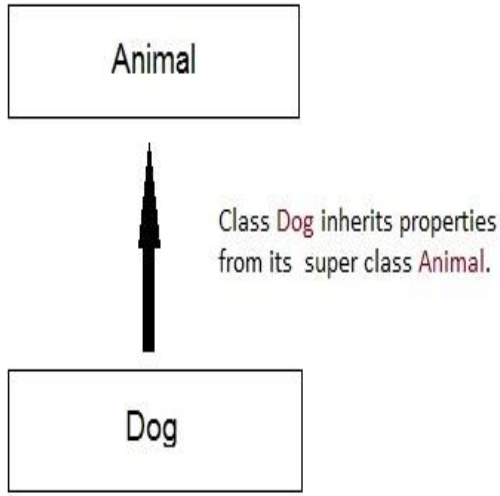
Example 2

```
#include<iostream>
using namespace std;
class Animal
{
  public:
  int legs;
};
int Animal::legs=4;
class Dog : public Animal
{
  public:
  int tail;
};
int Dog::tail=1;
```

```
int main()
{
 Dog d;
 cout << d.legs;
 cout << d.tail;
}
```

Output :

4 1



Consider a base class Shape and its derived class Rectangle as follows

Example 3:

```
#include <iostream>
 using namespace std;
// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }
   protected:
      int width;
      int height;
};
// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
int main(void) {
   Rectangle Rect;
   Rect.setWidth(5); // base class member function
   Rect.setHeight(7); //base class member function
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
}output:
Total area: 35
```

Modes of Inheritance

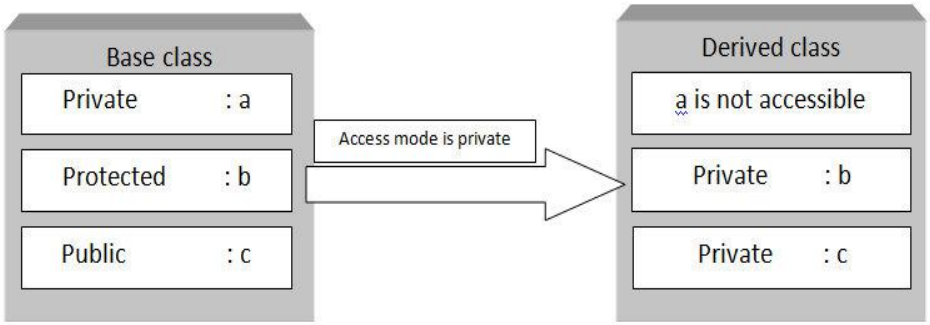
Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.

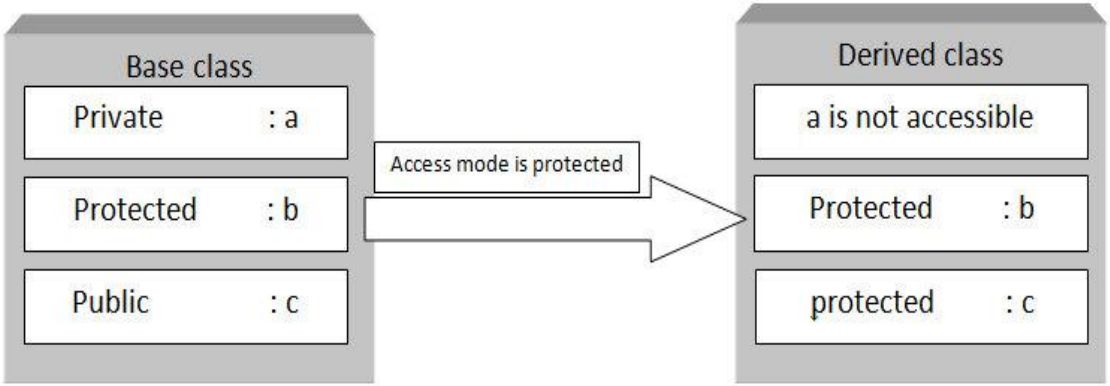
Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.

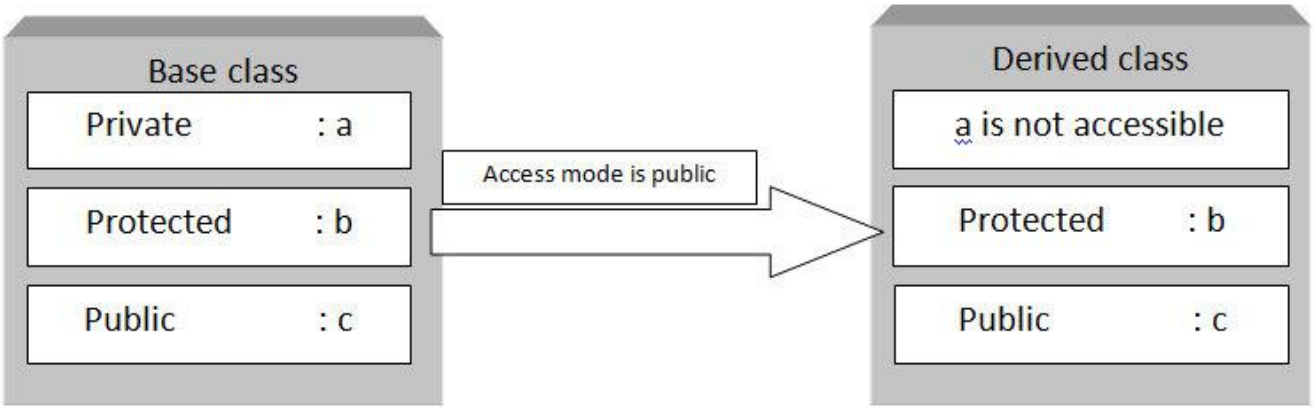
Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class. Private members of the base class will never get inherited in sub class.

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible
Protected	Protected	Private	Protected

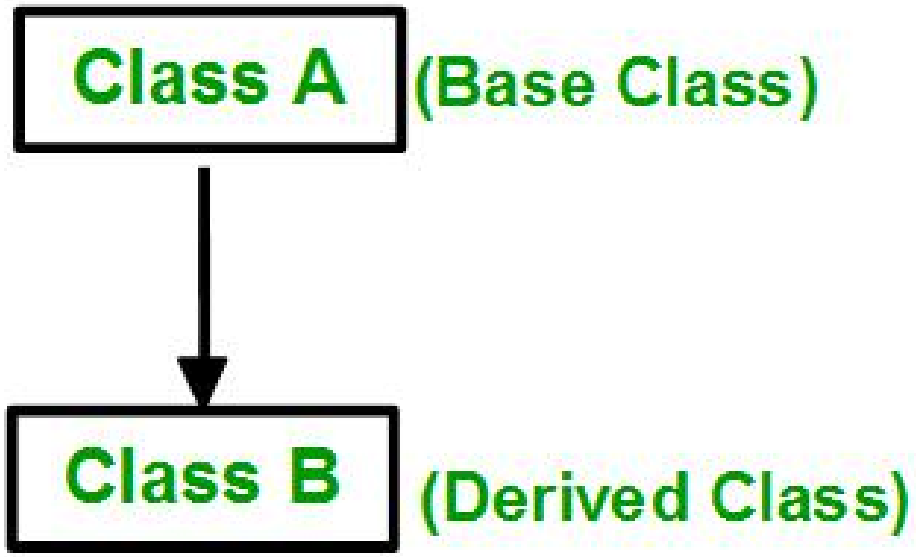

Base class
Private : a
Protected : b
Public : c
Access mode is private
Derived class
a is not accessible
Private : b
Private : c


Base class
Private : a
Protected : b
Public : c
Access mode is protected
Derived class
a is not accessible
Protected : b
protected : c


Base class
Private : a
Protected : b
Public : c
Access mode is public
Derived class
a is not accessible
Protected : b
Public : c

Types of 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.



Syntax:

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

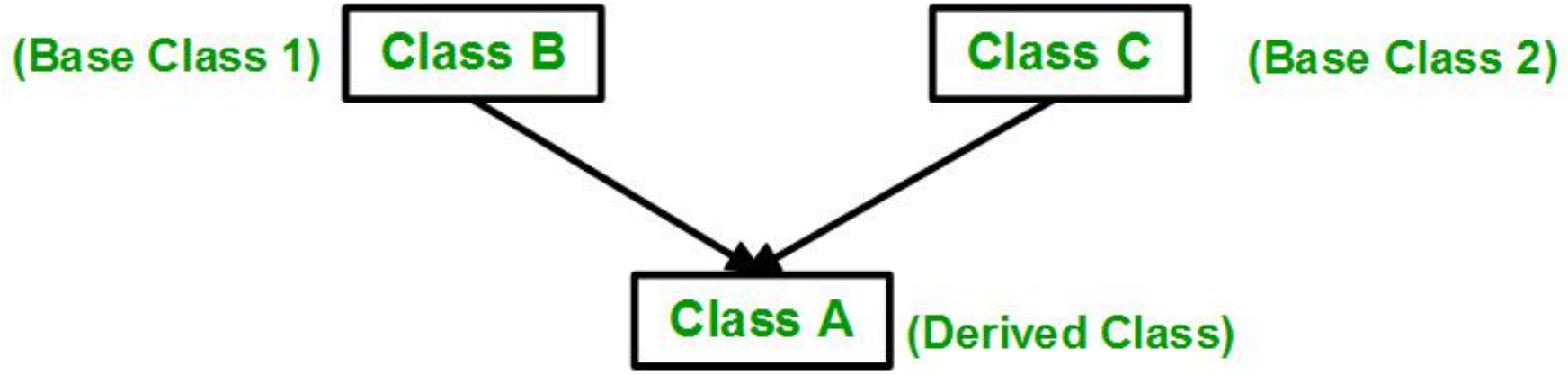
```
// C++ program to explain Single inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from base class
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: 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.



Syntax:

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

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

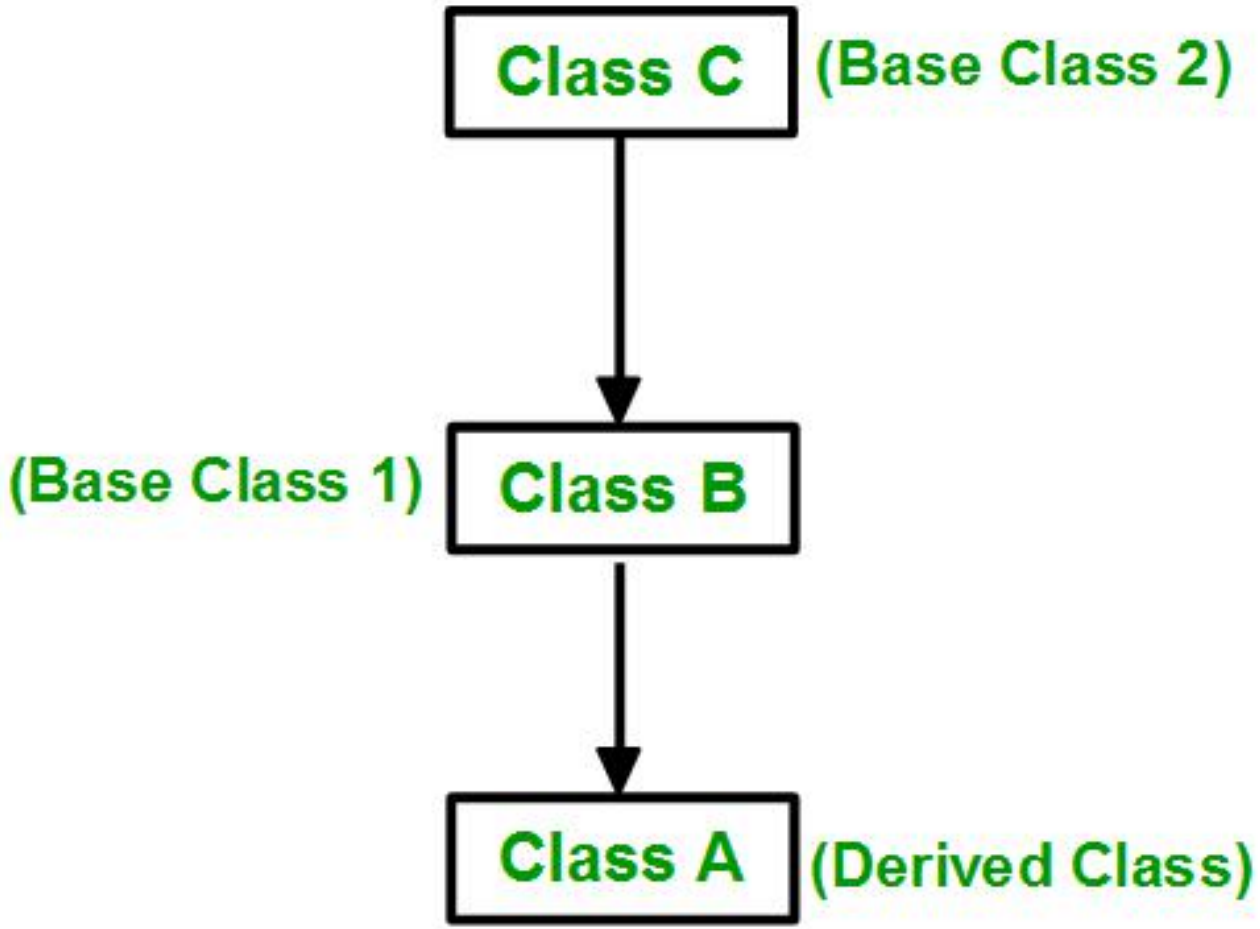
```
#include <iostream>
using namespace std;
 // first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" <<
endl;
    }
};
// second base class
class FourWheeler {
   public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler
Vehicle" << endl;
    }
```

```
// 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
```

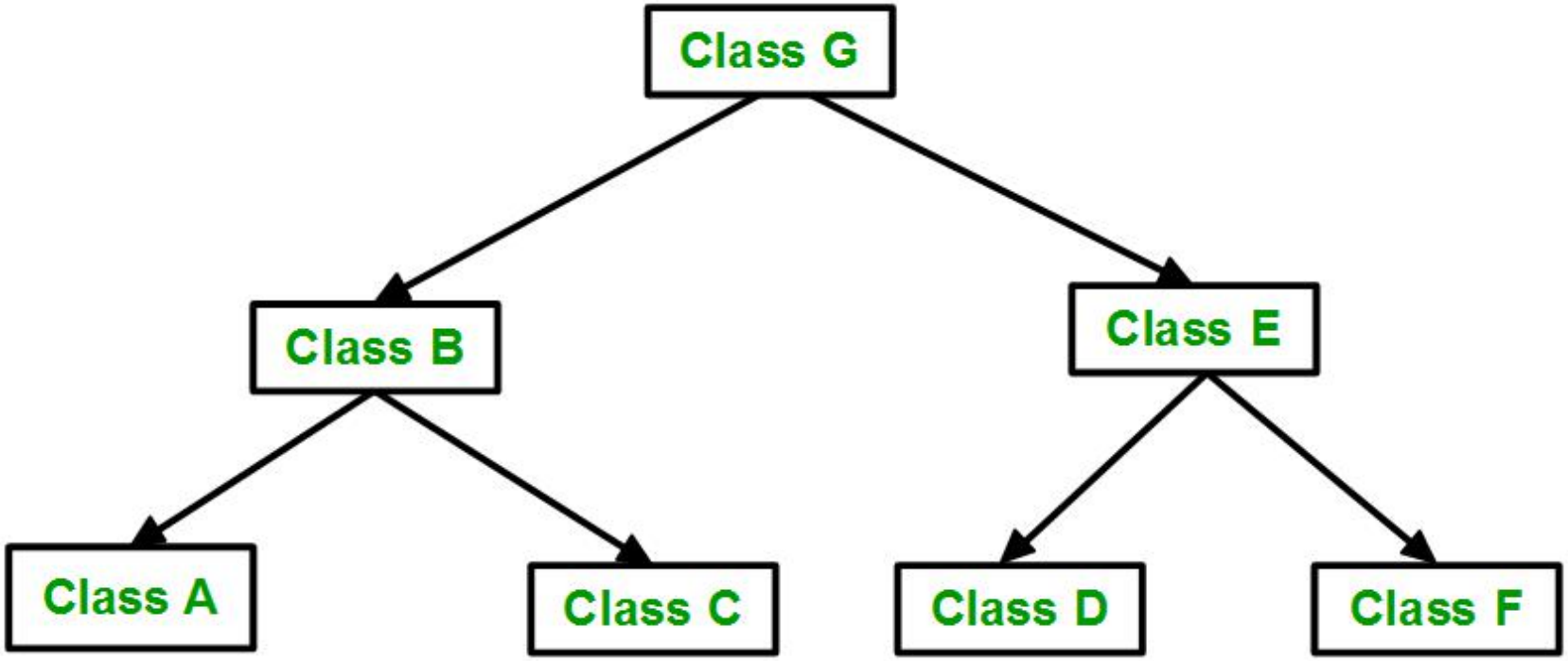
Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



```
#include <iostream>
using namespace std;
class Vehicle //base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are
vehicles"<<endl;
    }
};
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

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.



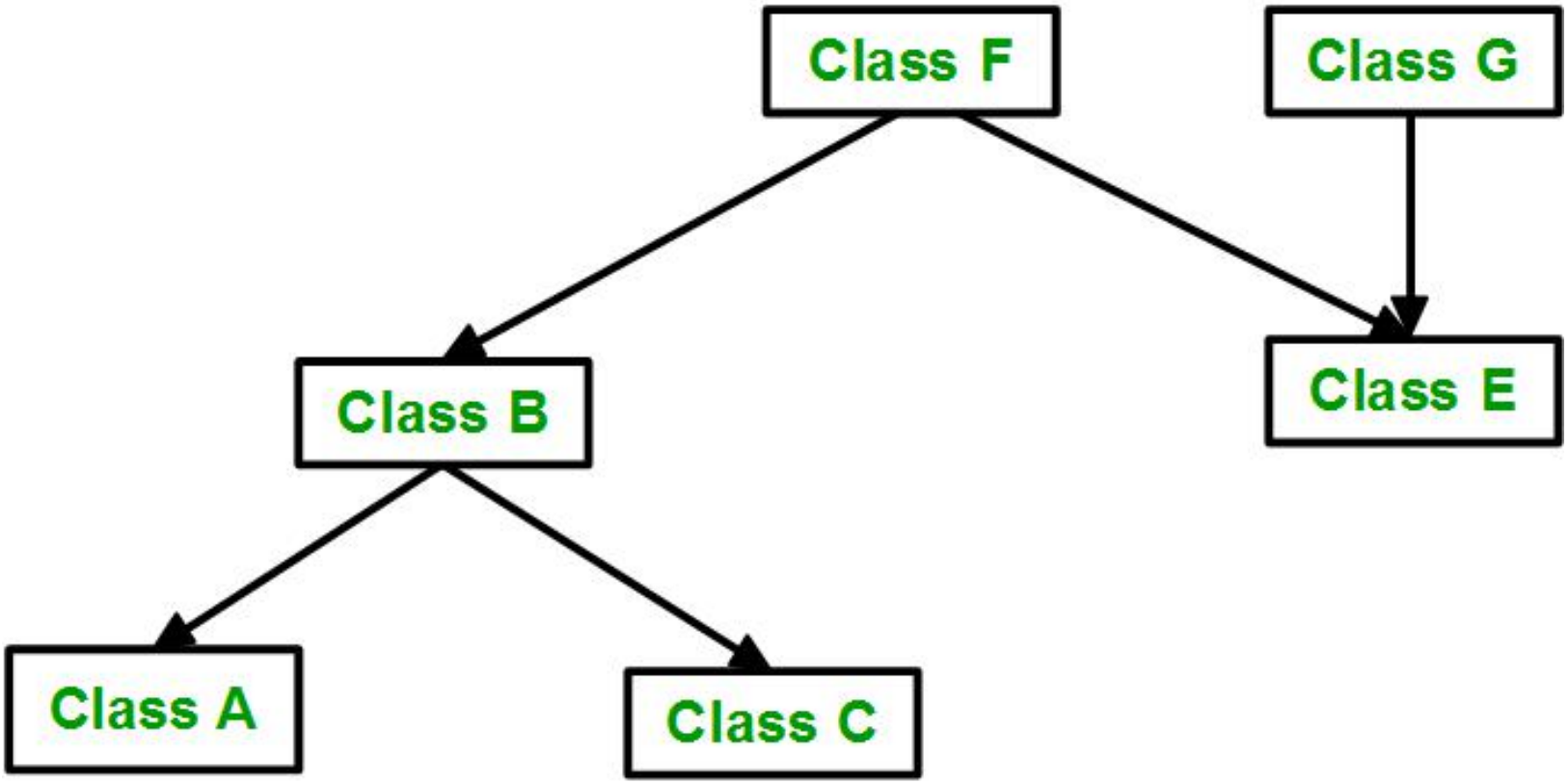
```
// C++ program to implement Hierarchical Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
// first sub class
class Car: public Vehicle
{

};
// 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;
}
```

Output:

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

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

Below image shows the combination of hierarchical and multiple inheritance:



```
#include <iostream>
using namespace std;
class Vehicle // base class
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" <<
endl;
    }
};
class Fare  //base class
{
public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
// first sub class
class Car: public Vehicle
{
 };
// second sub class
class Bus: public Vehicle, public Fare
{
};
int main() // main function
{
    // creating object of sub class will
    // invoke the constructor of base
class
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```

Constructors/Destructors in derived classes

- 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.

- Why the base class's constructor is called on creating an object of derived class?
- What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only. This is why the constructor of base class is called first to initialize all the inherited members.

```
// C++ program to show the order of constructor call in single inheritance
 #include <iostream>
using namespace std;

// base class
class Parent
{
    public:

    // base class constructor
    Parent()
    {
        cout << "Inside base class" << endl;
    }
};

// sub class
class Child : public Parent
{
    public:
        //sub class constructor
    Child()
    {
        cout << "Inside sub class" << endl;
    }
};
// main function
int main() {
     // creating object of sub class
    Child obj;
     return 0;
} Output:
Inside base class
Inside sub class
```

Order of constructor call for Multiple Inheritance

For multiple inheritance order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.

```
 #include <iostream>
using namespace std;
class Parent1 //base class1
{
   public:
   Parent1()
    {
        cout << "Inside first base class"
<< endl;
    }
};
class Parent2  //base class2
{
    public:
    Parent2()
    {
        cout << "Inside second base
class" << endl;
    }  };
/ /child class inherits Parent1 and
Parent2
class Child : public Parent1, public
Parent2
{
    public:
   Child()
    {
        cout << "Inside child class" <<
endl;
    }
};
int main() {
     Child obj1; //create child class obj
    return 0;
} Output:
Inside first base class
Inside second base class
Inside child class
```

Order of constructor and Destructor call for a given order of Inheritance

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)

How to call the parameterized constructor of base class in derived class constructor?

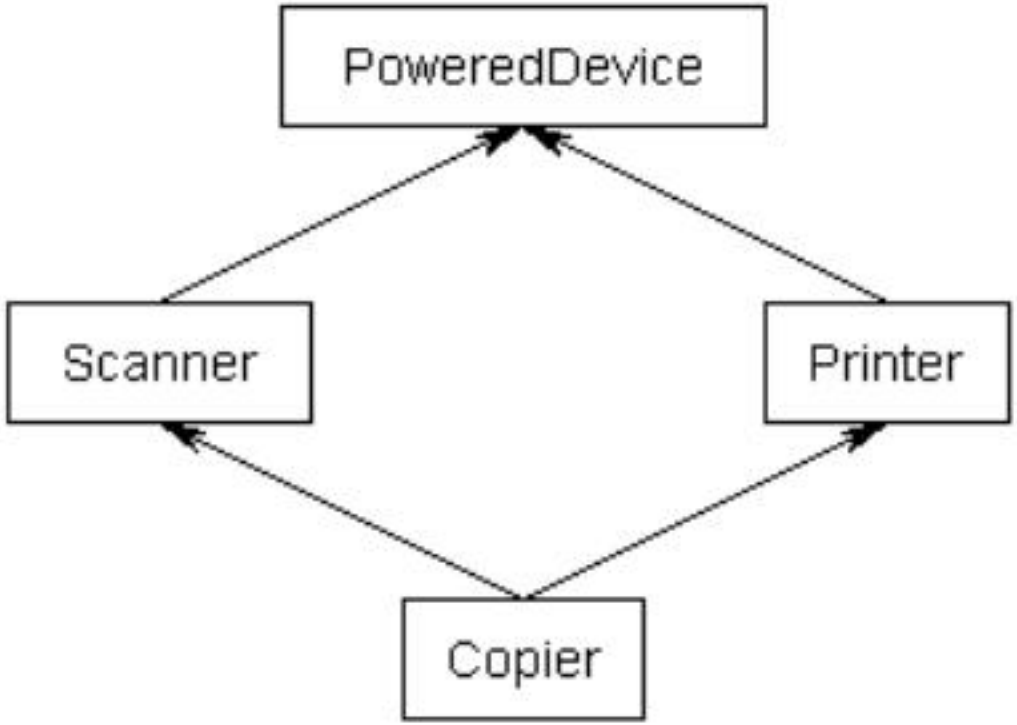
To call the parameterised constructor of base class when derived class's parameterised constructor is called, you have to explicitly specify the base class's parameterised constructor in derived class as shown in below program:

```
/* 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;
    }
};
class Child : public Parent
{
    public:
     // sub class's parameterised constructor
    Child(int j): Parent(j)
    {
        cout << "Inside sub class's parameterised constructor" << endl;
    }
};
int main() {
    Child obj1(10);
    return 0;
}Output:
Inside base class's parameterised constructor
Inside sub class's parameterised constructor
```

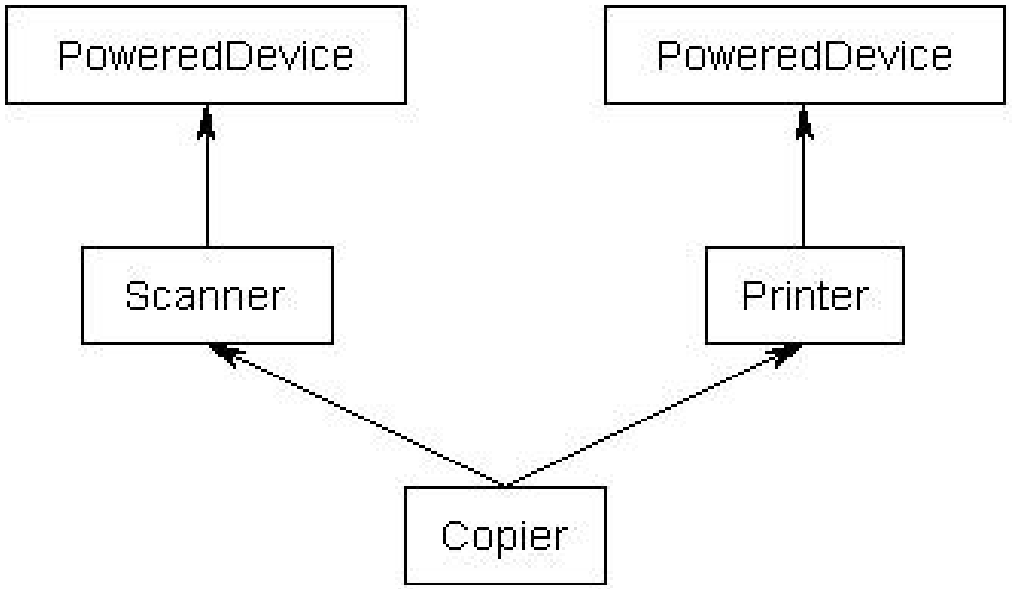
Important Points:

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

Virtual base classes: The diamond problem



If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class - one from Printer, and one from Scanner. This has the following structure:



```
#include<iostream>
using namespace std;
class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
                cout << "PoweredDevice: " << power << '\n';
    }
};
class Scanner: public PoweredDevice
{
public:
    Scanner(int scanner, int power): PoweredDevice(power)
    {
                cout << "Scanner: " << scanner << '\n';
    }
};
```

```
class Printer: public PoweredDevice
{
public:
    Printer(int printer, int power): PoweredDevice(power)
    {
                        cout << "Printer: " << printer << '\n';
    }
};
class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power): Scanner(scanner, power),
                                                 Printer(printer, power)
    {
    }
};
```

```
int main()
{
    Copier copier(1, 2, 3);
return 0;
}
```

output:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

Explanation:

As you can see, PoweredDevice got constructed twice. While this is often desired, other times you may want only one copy of PoweredDevice to be shared by both Scanner and Printer.

Virtual base classes

To share a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a virtual base class, which means there is only one base object that is shared.

```
class PoweredDevice
{
};
 class Scanner: virtual public PoweredDevice
{
};
 class Printer: virtual public PoweredDevice
{
};
 class Copier: public Scanner, public Printer
{
};
```

Example showing how to use the virtual keyword to create a shared base class:

Now, when you create a Copier class, you will get only one copy of PoweredDevice that will be shared by both Scanner and Printer.

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

Example:

```
#include <iostream>
using namespace std;
class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
                    std::cout << "PoweredDevice: " << power << '\n';
    }
};
```

```
class Scanner: virtual public PoweredDevice // PoweredDevice is now a virtual base class
{
public:
    Scanner(int scanner, int power) : PoweredDevice(power) // this line is required to create Scanner objects, but ignored in this case
    {
                    std::cout << "Scanner: " << scanner << '\n';
    }
};
 class Printer: virtual public PoweredDevice //PoweredDevice is now a virtual base class
{
public:
    Printer(int printer, int power): PoweredDevice(power) // this line is required to create Printer objects, but ignored in this case
    {               std::cout << "Printer: " << printer << '\n';
    }
};
```

```
class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power) : Scanner(scanner, power),
            Printer(printer, power),PoweredDevice(power) // PoweredDevice is
constructed here
    {
    }
};
int main()
{
    Copier copier(1, 2, 3);
}
```

Output:

```
PoweredDevice: 3
Scanner: 1
Printer: 2
```

As you can see, PoweredDevice only gets constructed once.

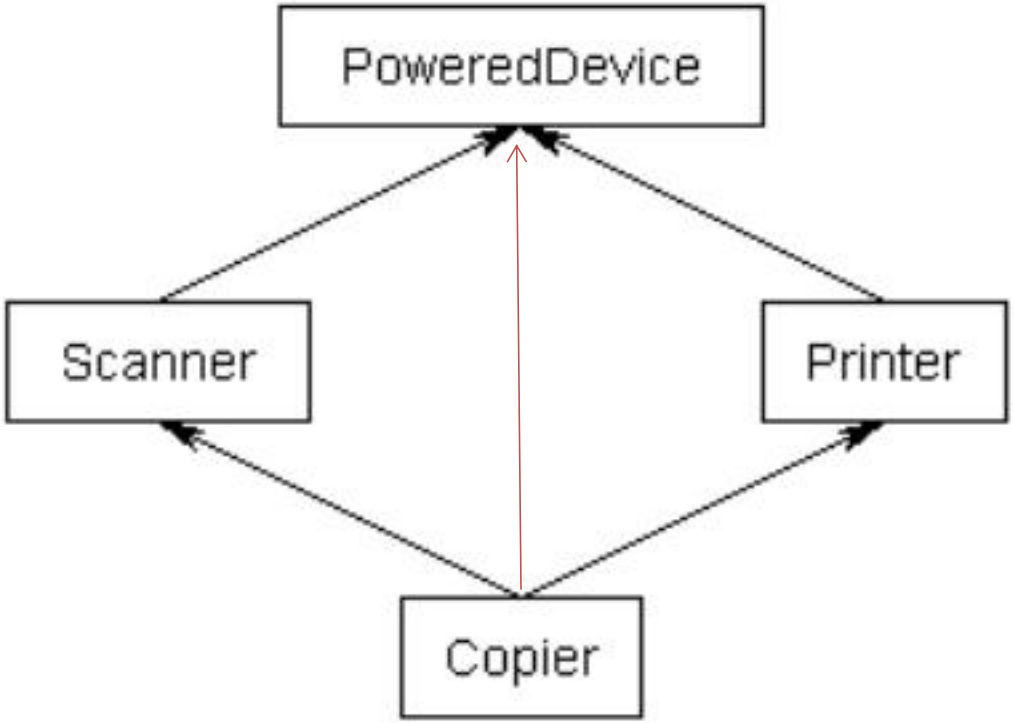
There are a few details that we would be remiss if we did not mention.

- First, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes.

- Second, note that the Scanner and Printer constructors still have calls to the PoweredDevice constructor. When creating an instance of Copier, these constructor calls are simply ignored because Copier is responsible for creating the PoweredDevice, not Scanner or Printer. However, if we were to create an instance of Scanner or Printer, those constructor calls would be used, and normal inheritance rules apply.

- Third, if a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class. In this case, Copier inherits Printer and Scanner, both of which have a PoweredDevice virtual base class. Copier, the most derived class, is responsible for creation of PoweredDevice. Note that this is true even in a single inheritance case: if Copier was singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier is still responsible for creating PoweredDevice.

- Fourth, a virtual base class is always considered a direct base of its most derived class (which is why the most derived class is responsible for its construction). But classes inheriting the virtual base still need access to it. So in order to facilitate this, the compiler creates a virtual table for each class directly inheriting the virtual class (Printer and Scanner). These virtual tables point to the functions in the most derived class. Because the derived classes have a virtual table, that also means they are now larger by a pointer (to the virtual table).

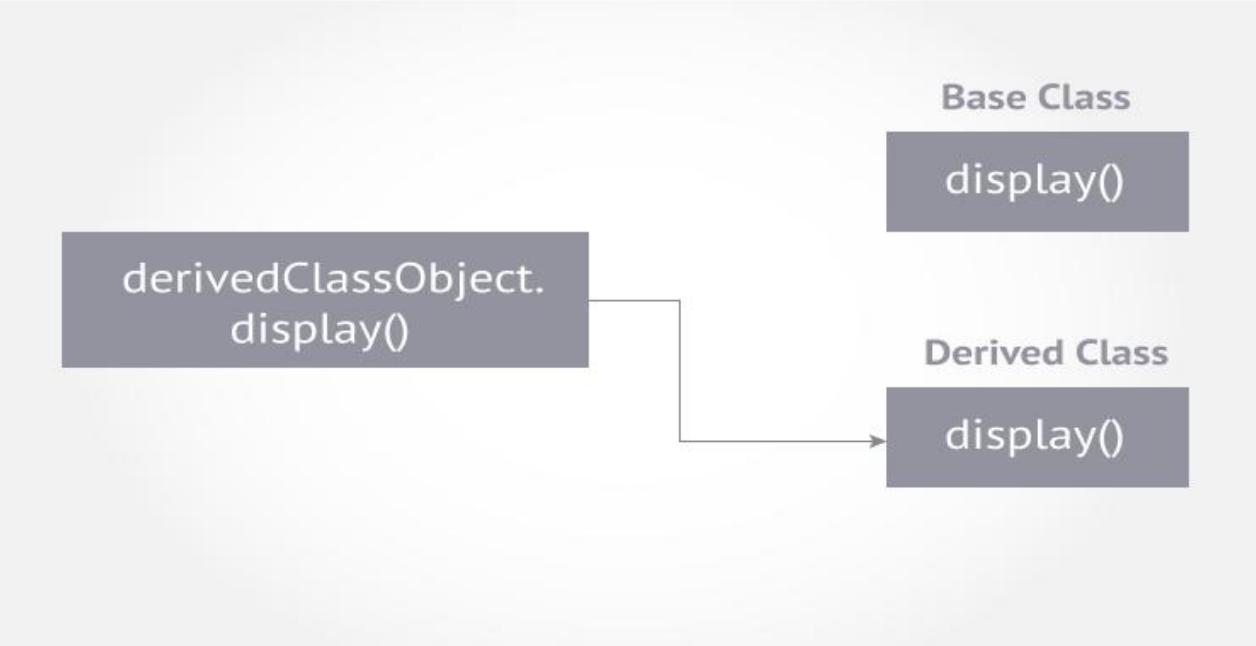


Funtion overriding:

- Suppose, both base class and derived class have a member function with same name and arguments (number and type of arguments).

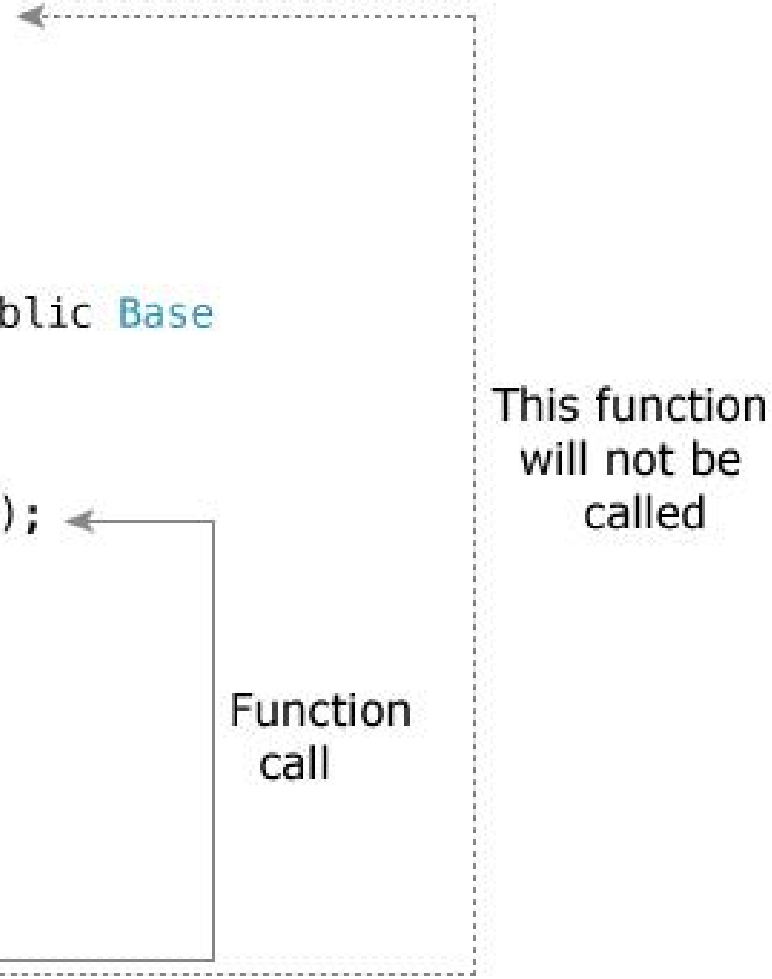
- If you create an object of the derived class and call the member function which exists in both classes (base and derived), the member function of the derived class is invoked and the function of the base class is ignored.

  This feature in C++ is known as function overriding.



Example:

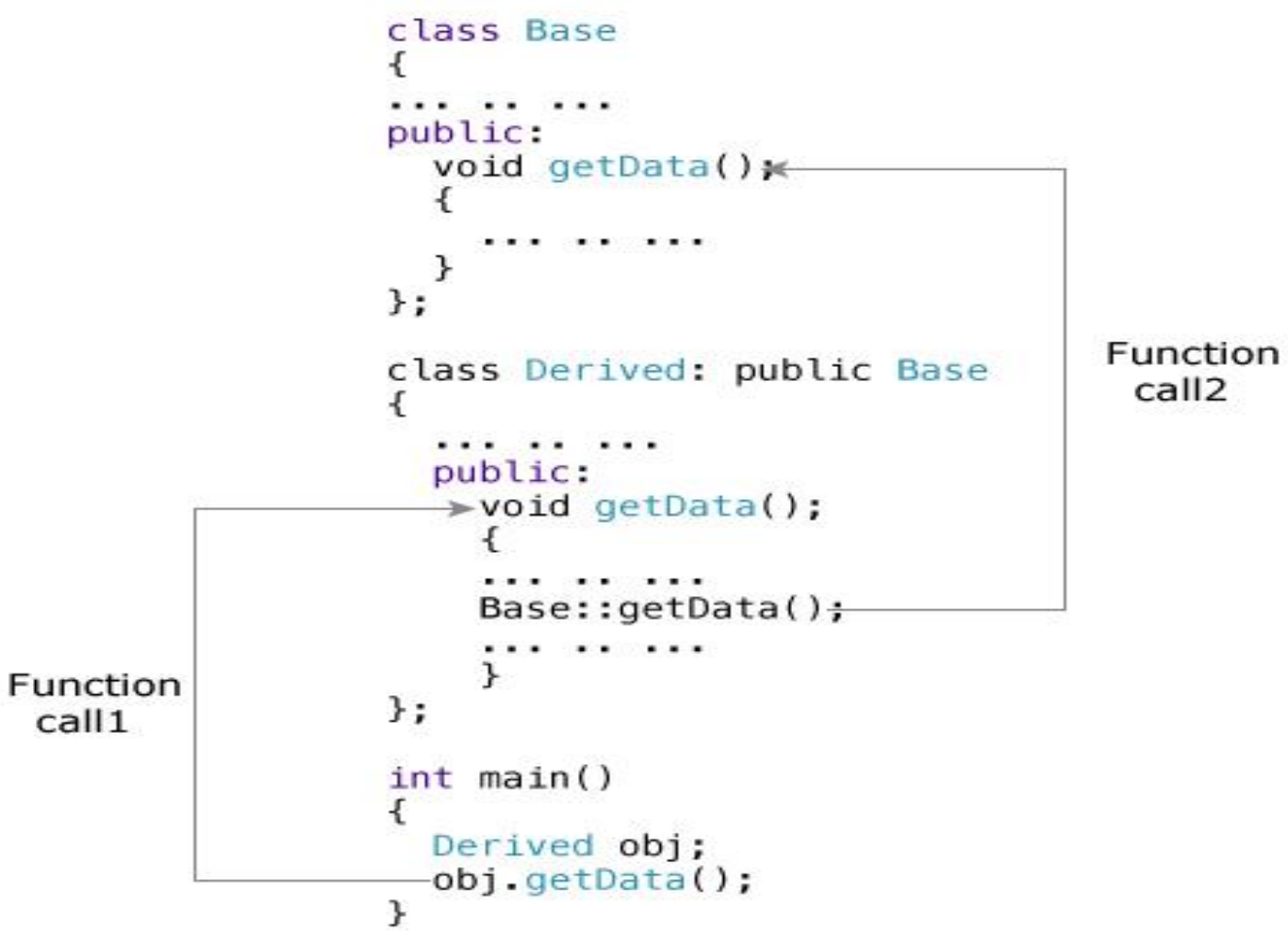
```
class Base
{
... .. ...
public:
  void getData();
  {
    ... .. ...
  }
};

class Derived: public Base
{
  ... .. ...
  public:
    void getData();
    {
    ... .. ...
    }
};

int main()
{
  Derived obj;
  obj.getData();
}
```



How to access the overridden function in the base class from the derived class?

To access the overridden function of the base class from the derived class, scope resolution operator :: is used.

For example,If you want to access getData() function of the base class, you can use the following statement in the derived class. i.e;Base::getData();



Nested Classes

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

```
#include<iostream>
using namespace std;
class Enclosing {
   int x;
  /* start of Nested class declaration */
   class Nested {
      int y;
      void NestedFun(Enclosing *e) {
        cout<<e->x;  /* works fine:
nested class can access
private members of Enclosing class*/
      }
   }; // declaration Nested class ends
here
}; // declaration Enclosing class ends
here
int main()
{
}
```

```
#include<iostream>
 using namespace std;
class Enclosing {
          int x;
 /* start of Nested class declaration */
   class Nested {
      int y;
   }; // declaration Nested class ends
here
    void EnclosingFun(Nested *n) {
        cout<<n->y;  // Compiler Error: y is
private in Nested
   }
}; // declaration Enclosing class ends
here
int main()
{
}
```

Nesting(containership) of classes

If an object of a class is becoming a member of another class,it is referred as member object.

```
class X { .....};
class Y { .....};
class Z {
X x1;
//x1 is an object of X class
Y y1;
//y1 is an object of Y class
..... };
```

In the above example, object of class Z contains the objects of class A & class B. This kind of relationship is called containership or nesting.

```
#include<iostream>
using namespace std;
class A
{
      public:
      void get() {
      cout<<“fun1”;
}
};
class B
{
      public:
      void show() {
      cout<<“fun2”;
}
};
```

```
class C
{
      A obj1;
      B obj2;
      public:
      void disp(){
      obj1.get();
      obj2.show();
     }
};
main(){
C c1;
c1.disp();
}
output:
fun1
fun2
```