Inheritance

Access Modifier

1. Public
2. Private
3. Protected

Public Class A
{
Int i , j;
 	
void abc ()
{
S=u+t;
}
Int x , y
 	
}
	1. Particular member access modifier
	2. Access modifier block
	3. Class declaration as a public

Concept Of Inheritance

- Way-1 : calling parent class component from main method using child class object reference
- Way-2 : calling parent class member from child class

	Mode of inheritance		
parent class access modifier	Public	Protected	private
Public	w1 ,w 2 Publicly	~~W1~~ , w2 Protectedly	~~W1~~ , w2 Privately (2nd generation is locked in multi level)
Protected	~~W1~~ , w2 protectedly	~~W1~~ , w2 Protectedly	~~W1~~ , w2 Privately (2nd generation is locked in multi level)
Private	~~W1 , W2~~ Not possible	~~W1 , W2~~ Not possible	~~W1 , W2~~ Not possible

Mode Of inheritance

- Public Mode
- Private Mode
- Protected Mode

Public mode	Protected mode	Private Mode

class classOne{
public:
};
 	
class classTwo : public classOne{
}
};
	class classOne{
	public:
	};
	
	class classTwo : Protected classOne{
	}
	};
		class classOne{
		public:
		};
		
		class classTwo : private classOne{
		}
		};

Types Of Inheritance

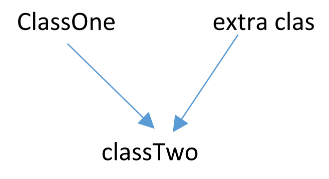
1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. Hybrid Inheritance
5. Hierarchical Inheritance

1. Single

```
 class classOne{
public:
};

class classTwo : public classOne{
   }
};
```

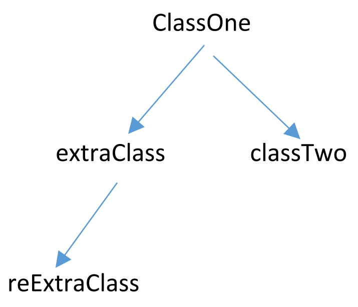
2. Multiple Inheritance

```
class classOne{
public:
};

class extraClass{
public:
```



};
 	
class classTwo : public classOne , extraClass{
   }
};	

3. Multi Level Inheritance

class classOne{
public:
 	
};
 	
class extraClass : public classOne{
public:
};
 	
Class reExtra : public extraClass {
Public:
};
 	
class classTwo : public classOne {
   }
};
	

Friend class

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes.

We can declare a friend class in C++ by using the **friend** keyword.

class1	class2	class3	………..	classN
Public				
Protected				
Private				

Real life case

- Custom content
- Page - owner/ admin - moderation
- Reliable second mail , id

[NB: details of friend class, friend function – Book]

Access Modifiers

1. Public
2. Private
3. Protected

Ways of mentioning or using an access modifier

Class A
{
 	
public :
Int i , j;
 	
void bc ()
{
S=u+t;
}
 	
public:
Int x , y
}
	public Class B
	{
	Int i , j;
	void ab ()
	{
	S=u+t;
	}
	Int x , y
	}
		Class C
		{
		Public Int i ,
		Private int j;
		
		Public int ac ()
		{
		B on;
		on.abc();
		S=u+t;
		}
		
		Int x , y
		
		}
			Int main ()
			
			{
			C obj;
			Obj.ac;
			
			}

Let us now look at each one of these access modifiers in detail:

1. Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

	public Class B
	{
	Int i , j;
	void ab ()
	{
	S=u+t;
	}
	Int x , y
	}
		Class C
		{
		Public Int i ,
		Private int j;
		
		Public int ac ()
		{
		B on;
		on.ab();
		
		}
		
		Int x , y
		
		}
			Int main ()
			
			{
			C obj;
			Obj.ac;
			
			}

2. Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

Class A
{
 	
private:
Int i , j;
 	
void bc ()
{
S=u+t;
}
Void nm()
{
Cout << I
// private
member are
		Class C
		{
		Public Int i ,
		Private int j;
		
		Public int ac ()
		{
		A on;
		on.bc();
		// A class member is private , so not accessible
		
		}
			Int main ()
			
			{
			C obj;
			Obj.ac;
			}

accessible inside the class } private: Int x , y }		Int x , y }	

Encapsulation

(Getter Setter Method)

Encapsulate

Private:

Int pet;

Int speed;

Int getPet()

{}

Int setSet()

{}

Polimorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

1. Compile Time Polimorphism
   - Method Overloding
   - Operator Overloading
2. Runtime Polimorphism
   - Method overriding
   - Virtual function

Based on the functionality, you can categorize polymorphism into two types:

- ### Compile-Time Polymorphism:

When the relationship between the definition of different functions and their function calls, is determined during the compile-time, it is known as compile-time polymorphism. This type of polymorphism is also known as static or early binding polymorphism. All the methods of compile-time polymorphism get called or invoked during the compile time.

You can implement compile-time polymorphism using function overloading and operator overloading. Method/function overloading is an implementation of compile-time polymorphism where the same name can be assigned to more than one method or function, having different arguments or signatures and different return types. Compile-time polymorphism has a much faster execution rate since all the methods that need to be executed are called during compile time. However, it is less preferred for handling complex problems since all the methods and details come to light only during the compile time. Hence, debugging becomes tougher.

The implementation of compile-time polymorphism is achieved in two ways:

Function overloading

Operator overloading

Runtime Polymorphism

In runtime polymorphism, the compiler resolves the object at run time and then it decides which function call should be associated with that object. It is also known as dynamic or late binding polymorphism. This type of polymorphism is executed through virtual functions and function overriding. All the methods of runtime polymorphism get invoked during the run time.

Method overriding is an application of run time polymorphism where two or more functions with the same name, arguments, and return type accompany different classes of the same structure. This method has a comparatively slower execution rate than compile-time polymorphism since all the methods that need to be executed are called during run time. Runtime polymorphism is known to be better for dealing with complex problems since all the methods and details turn up during the runtime itself.

The implementation of run time polymorphism can be achieved in two ways:

Function overriding

Virtual functions

Function Overriding : (reference - Programiz)

In C++ inheritance, we can have the same function in the base class as well as its derived classes.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

So, different functions are executed depending on the object calling the function.

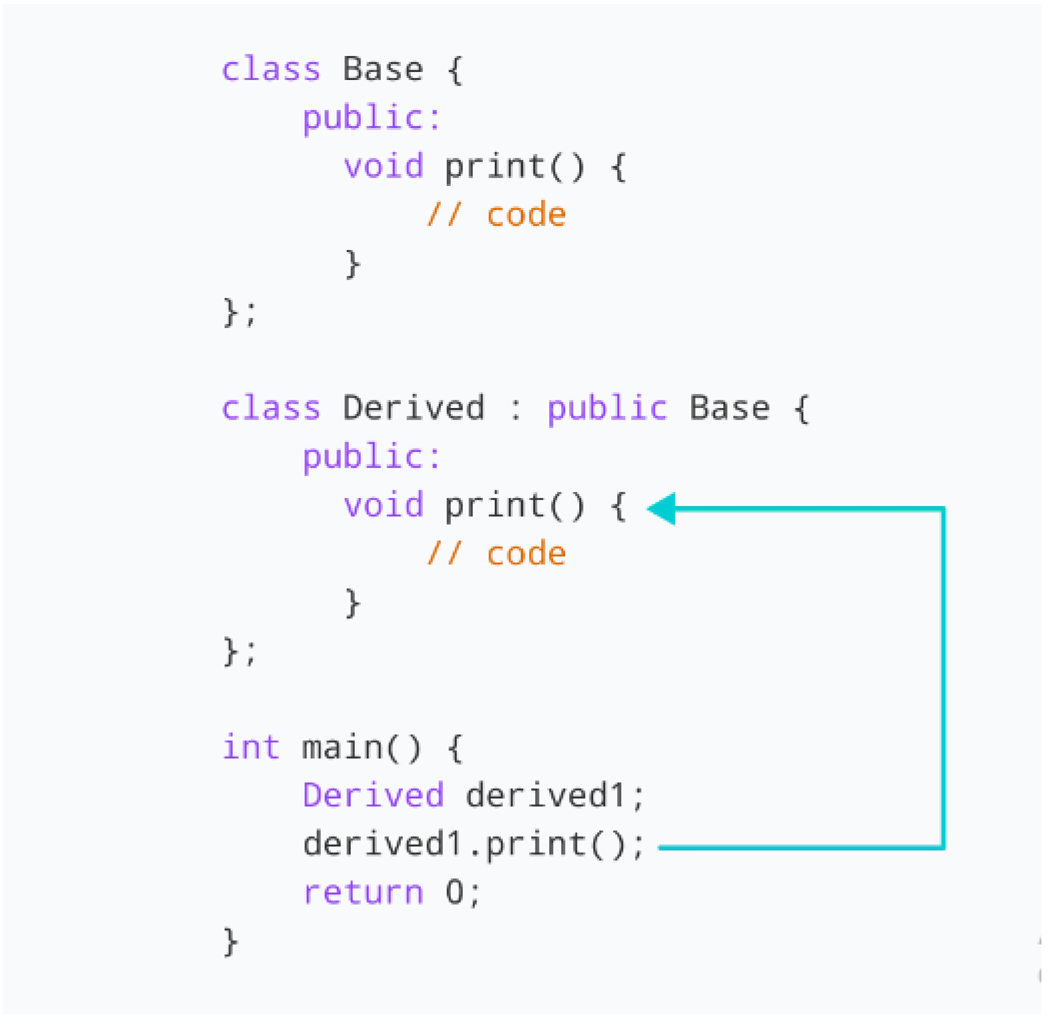
This is known as **function overriding** in C++. For example,

Example :

1. Animal code (in Program file)

2, Code-2

3, Code -3



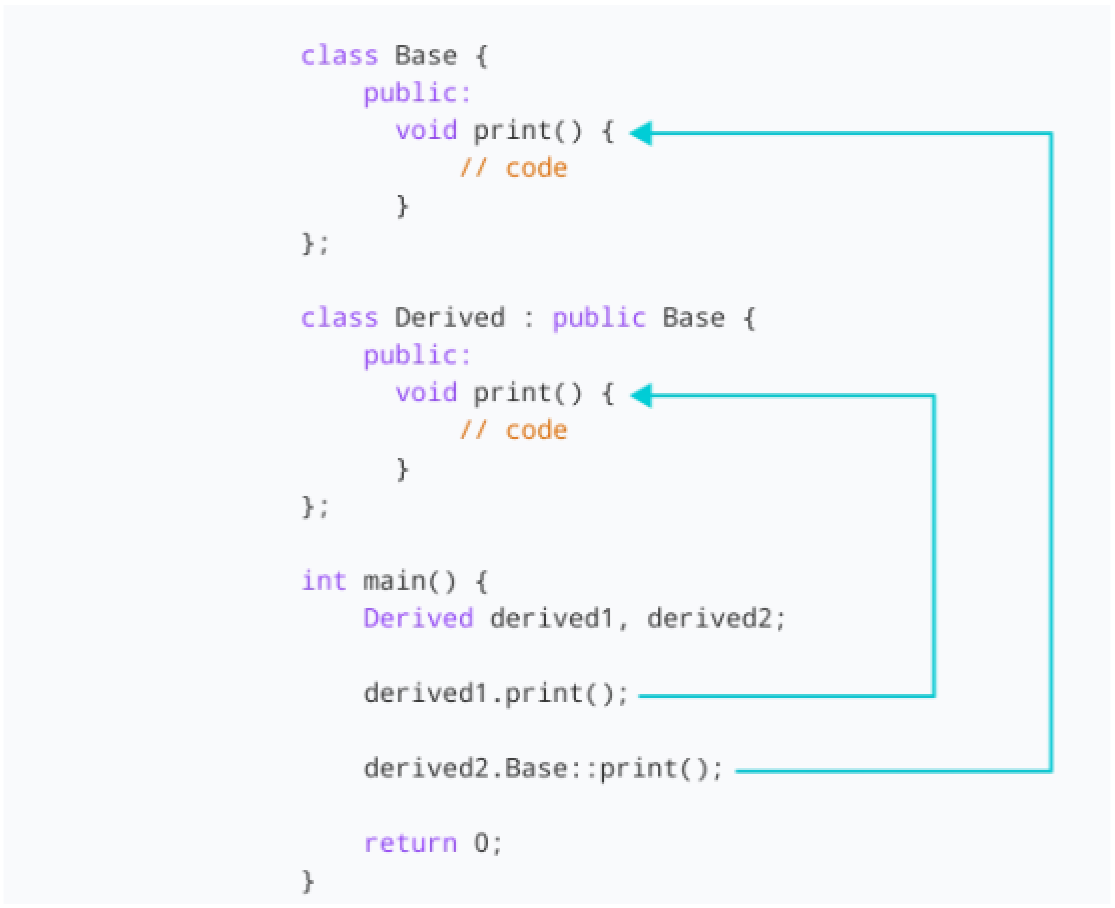
```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Here, the same function print() is defined in both Base and Derived classes. So, when we call print() from the Derived object derived1, the print() from Derived is executed by overriding the function in Base.

As we can see, the function was overridden because we called the function from an object of the Derived class. Had we called the print() function from an object of the Base class, the function would not have been overridden.

3, Code -3

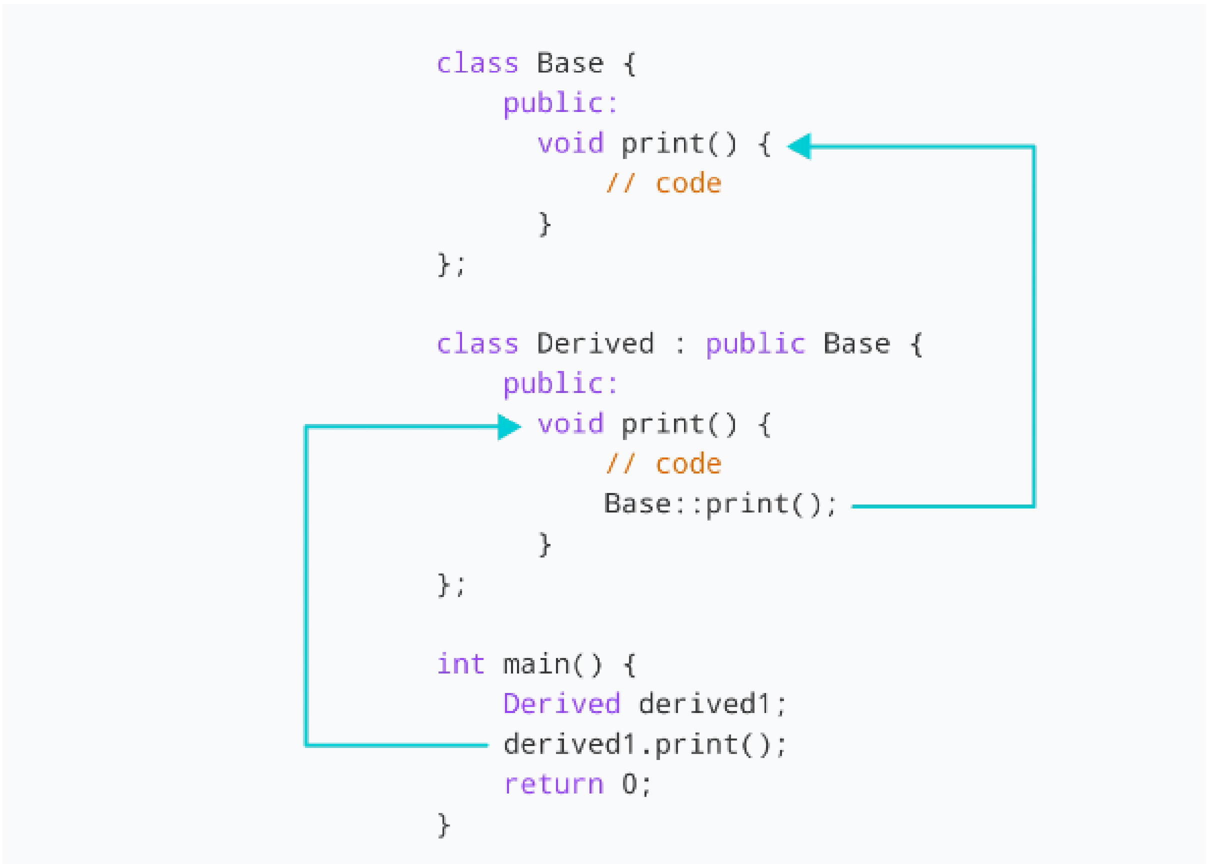
C++ Access Overridden Function to the Base Class



```cpp
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```

4, Code- 4

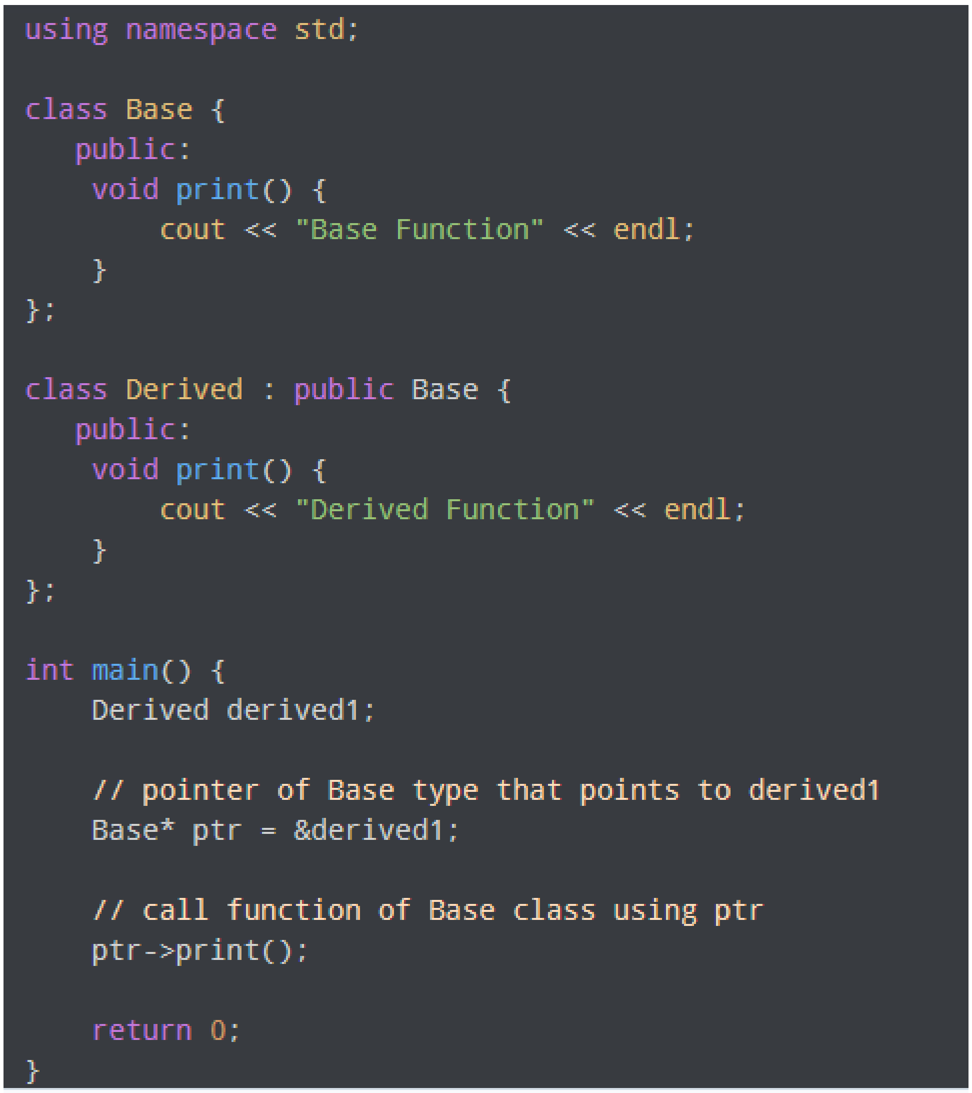
Call overriden function from derived class .



```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
          Base::print();
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

5, Code- 5

Calling overriden function using pointer



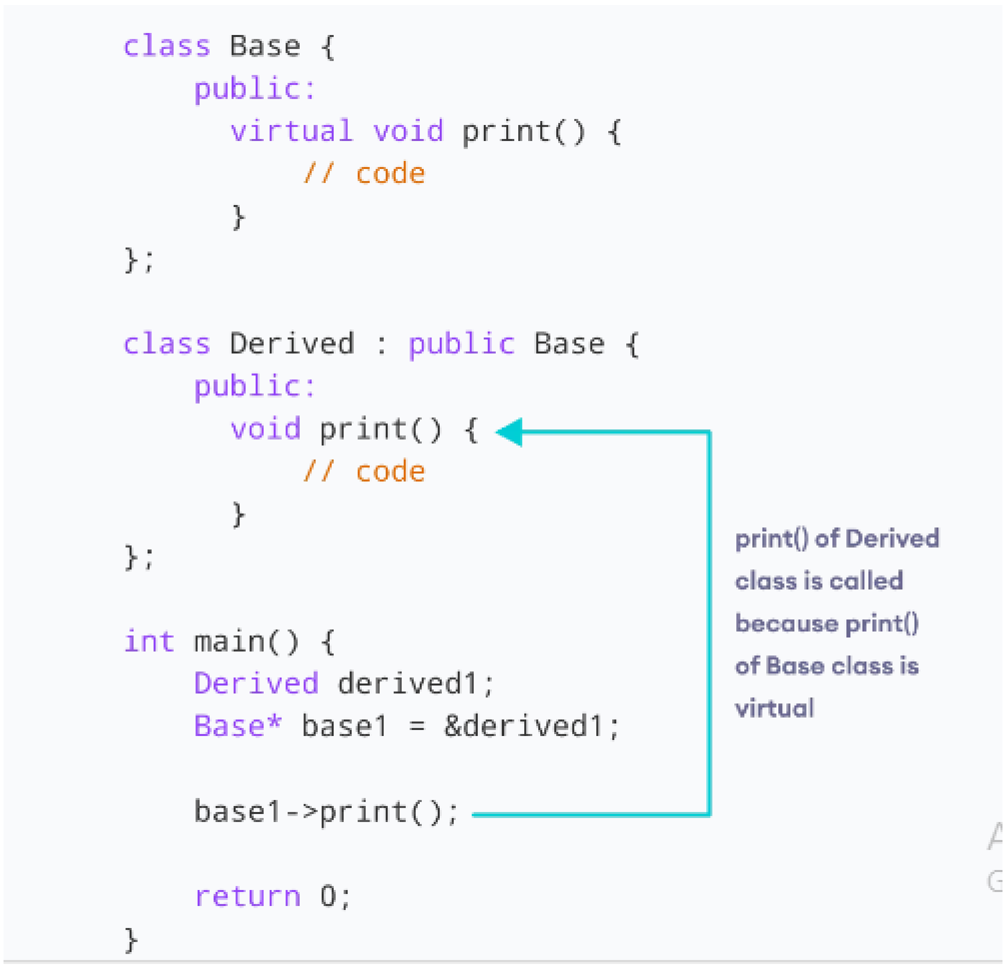
```cpp
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* ptr = &derived1;

    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

Virtual Function

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:



```
class Base {
    public:
      virtual void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```

print() of Derived class is called because print() of Base class is virtual

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

C++ override Identifier

C++ 11 has given us a new identifier `override` that is very useful to avoid bugs while using virtual functions.

This identifier specifies the member functions of the derived classes that override the member function of the base class.

For example,

```
class Base {

   public:

    virtual void print() {

        // code

    }

};

class Derived : public Base {

   public:

    void print() override {

        // code

    }

};
```

If we use a function prototype in `Derived` class and define that function outside of the class, then we use the following code:

```
class Derived : public Base {

   public:

    // function prototype

    void print() override;

};
```

```
// function definitionvoid Derived::print() {

    // code

}
```

Use of C++ override

When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.

Using the `override` identifier prompts the compiler to display error messages when these mistakes are made.

Otherwise, the program will simply compile but the virtual function will not be overridden.

Some of these possible mistakes are:

Functions with incorrect names: For example, if the virtual function in the base class is named `print()`, but we accidentally name the overriding function in the derived class as `pint()`.

Functions with different return types: If the virtual function is, say, of `void` type but the function in the derived class is of `int` type.

Functions with different parameters: If the parameters of the virtual function and the functions in the derived classes don't match.

No virtual function is declared in the base class.

Use of C++ Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat`.

Suppose each class has a data member named `type`. Suppose these variables are initialized through their respective constructors.

```
class Animal {

   private:

    string type;

    ... .. ...

    public:

      Animal(): type("Animal") {}
    ... .. ...
};
class Dog : public Animal {
   private:
    string type;
    ... .. ...
    public:
      Animal(): type("Dog") {}
    ... .. ...
};
class Cat : public Animal {
   private:
    string type;
      ... .. ...
    public:
      Animal(): type("Cat") {}
    ... .. ...
};
```

Now, let us suppose that our program requires us to create two `public` functions for each class:

1. `getType()` to return the value of `type`
2. `print()` to print the value of `type`

We could create both these functions in each class separately and override them, which will be long and tedious.

Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```
class Animal {
    ... .. ...
   public:
    ... .. ...

    virtual string getType {...}

};



... .. ...
```

```
... .. ...

void print(Animal* ani) {

    cout << "Animal: " << ani->getType() << endl;

}
```

This will make the code **shorter**, **cleaner**, and **less repetitive**.

Example 2: C++ virtual Function Demonstration

```
// C++ program to demonstrate the use of virtual function


#include <iostream>#include <string>using namespace std;
class Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Animal() : type("Animal") {}

    // declare virtual function
    virtual string getType() {
        return type;
    }
};
class Dog : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Dog() : type("Dog") {}

    string getType() override {
        return type;
    }
};
```

```
class Cat : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Cat() : type("Cat") {}
        string getType() override {
        return type;
 }
};
void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}
int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);


    return 0;
}
```

Output

```
Animal: Animal

Animal: Dog

Animal: Cat
```

Here, we have used the virtual function `getType()` and an `Animal` pointer `ani` in order to avoid repeating the `print()` function in every class.

```
void print(Animal* ani) {

    cout << "Animal: " << ani->getType() << endl;

}
```

In `main()`, we have created 3 `Animal` pointers to dynamically create objects of `Animal`, `Dog` and `Cat` classes.

```
// dynamically create objects using Animal pointers

Animal* animal1 = new Animal();

Animal* dog1 = new Dog();

Animal* cat1 = new Cat();
```

We then call the `print()` function using these pointers:

1. When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()`.
2. When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()`.
3. When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()`.