C++ Virtual Functions

programiz.com/cpp-programming/virtual-functions

Join our newsletter for the latest updates.

In this tutorial, we will learn about C++ virtual function and its use with the help of examples.

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:
    void print() {
        // code
    }
};

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

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.

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

    // calls function of Base class
    base1->print();

    return 0;
}
```

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

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

Virtual functions are an integral part of polymorphism in C++. To learn more, check our tutorial on C++ Polymorphism.

Example 1: C++ virtual Function

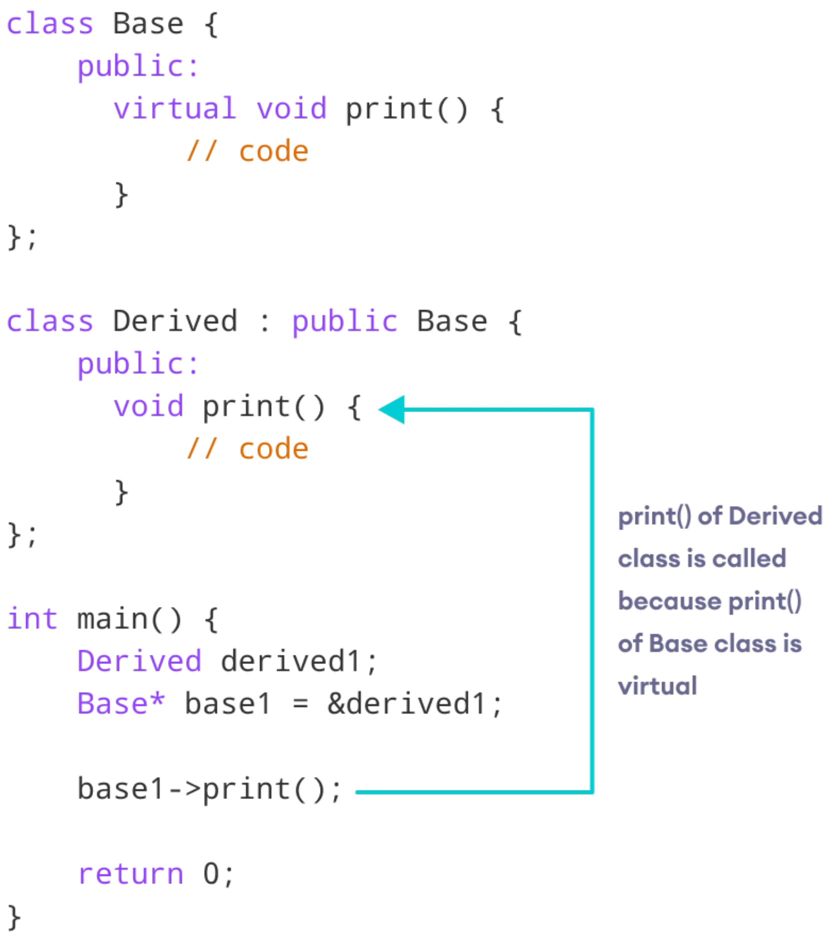
```
#include <iostream>
using namespace std;

class Base {
   public:
    virtual 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* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Output

```
Derived Function
```

Here, we have declared the `print()` function of `Base` as `virtual`.

So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object *derived1*.



```
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

Working of virtual functions in C++

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 definition
void 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()`.