

C++ Inheritance

 programiz.com/cpp-programming/inheritance

Join our newsletter for the latest updates.

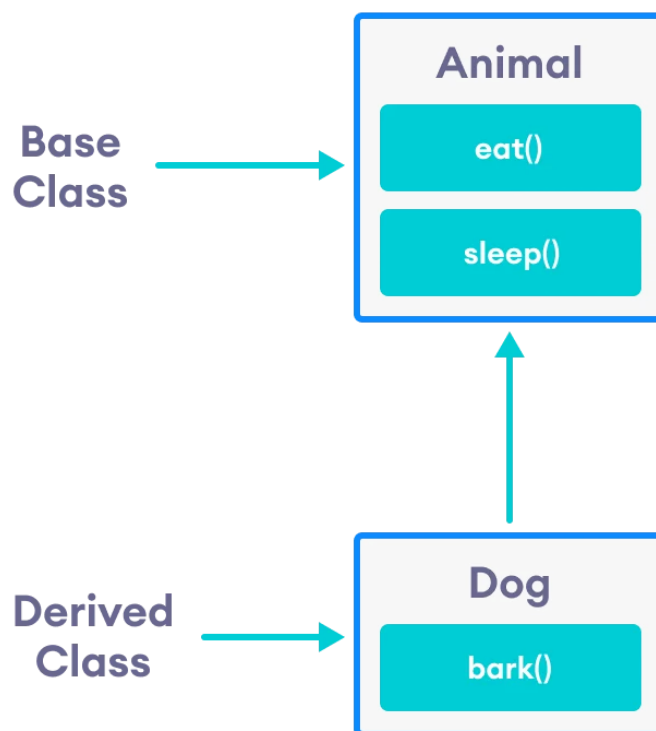
In this tutorial, we will learn about inheritance in C++ with the help of examples.

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).

The derived class inherits the features from the base class and can have additional features of its own. For example,

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

Here, the **Dog** class is derived from the **Animal** class. Since **Dog** is derived from **Animal**, members of **Animal** are accessible to **Dog**.



Inheritance in C++

Notice the use of the keyword **public** while inheriting Dog from Animal.

```
class Dog : public Animal {...};
```

We can also use the keywords `private` and `protected` instead of `public`. We will learn about the differences between using `private`, `public` and `protected` later in this tutorial.

is-a relationship

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.
-

Example 1: Simple Example of C++ Inheritance

```
// C++ program to demonstrate inheritance

#include <iostream>
using namespace std;

// base class
class Animal {

public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {

public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Here, *dog1* (the object of derived class **Dog**) can access members of the base class **Animal** . It's because **Dog** is inherited from **Animal** .

```
// Calling members of the Animal class
dog1.eat();
dog1.sleep();
```

C++ protected Members

The access modifier `protected` is especially relevant when it comes to C++ inheritance.

Like `private` members, `protected` members are inaccessible outside of the class. However, they can be accessed by **derived classes** and **friend classes/functions**.

We need `protected` members if we want to hide the data of a class, but still want that data to be inherited by its derived classes.

To learn more about protected, refer to our [C++ Access Modifiers](#) tutorial.

Example 2 : C++ protected Members

```

// C++ program to demonstrate protected members

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

// base class
class Animal {

private:
    string color;

protected:
    string type;

public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }

    void setColor(string clr) {
        color = clr;
    }

    string getColor() {
        return color;
    }
};

// derived class
class Dog : public Animal {

public:
    void setType(string tp) {
        type = tp;
    }

    void displayInfo(string c) {
        cout << "I am a " << type << endl;
        cout << "My color is " << c << endl;
    }

    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
    dog1.setColor("black");
}

```

```

// Calling member of the derived class
dog1.bark();
dog1.setType("mammal");

// Using getColor() of dog1 as argument
// getColor() returns string data
dog1.displayInfo(dog1.getColor());

return 0;
}

```

Output

```

I can eat!
I can sleep!
I can bark! Woof woof!!
I am a mammal
My color is black

```

Here, the variable *type* is **protected** and is thus accessible from the derived class **Dog** . We can see this as we have initialized **type** in the **Dog** class using the function **setType()** .

On the other hand, the **private** variable *color* cannot be initialized in **Dog** .

```

class Dog : public Animal {

    public:
        void setColor(string clr) {
            // Error: member "Animal::color" is inaccessible
            color = clr;
        }
};

```

Also, since the **protected** keyword hides data, we cannot access *type* directly from an object of **Dog** or **Animal** class.

```

// Error: member "Animal::type" is inaccessible
dog1.type = "mammal";

```

Access Modes in C++ Inheritance

In our previous tutorials, we have learned about C++ access specifiers such as public, private, and protected.

So far, we have used the **public** keyword in order to inherit a class from a previously-existing base class. However, we can also use the **private** and **protected** keywords to inherit classes. For example,

```
class Animal {  
    // code  
};  
  
class Dog : private Animal {  
    // code  
};  
  
class Cat : protected Animal {  
    // code  
};
```

The various ways we can derive classes are known as **access modes**. These access modes have the following effect:

1. **public:** If a derived class is declared in `public` mode, then the members of the base class are inherited by the derived class just as they are.
2. **private:** In this case, all the members of the base class become `private` members in the derived class.
3. **protected:** The `public` members of the base class become `protected` members in the derived class.

The `private` members of the base class are always `private` in the derived class.

To learn more, visit our [C++ public, private, protected inheritance](#) tutorial.

Member Function Overriding in Inheritance

Suppose, base class and derived class have member functions with the same name and arguments.

If we create an object of the derived class and try to access that member function, the member function in the derived class is invoked instead of the one in the base class.

The member function of derived class overrides the member function of base class.

Learn more about [Function overriding in C++](#).

Recommended Reading: *C++ Multiple Inheritance*