**C++ Inheritance**

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](https://www.programiz.com/cpp-programming/object-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

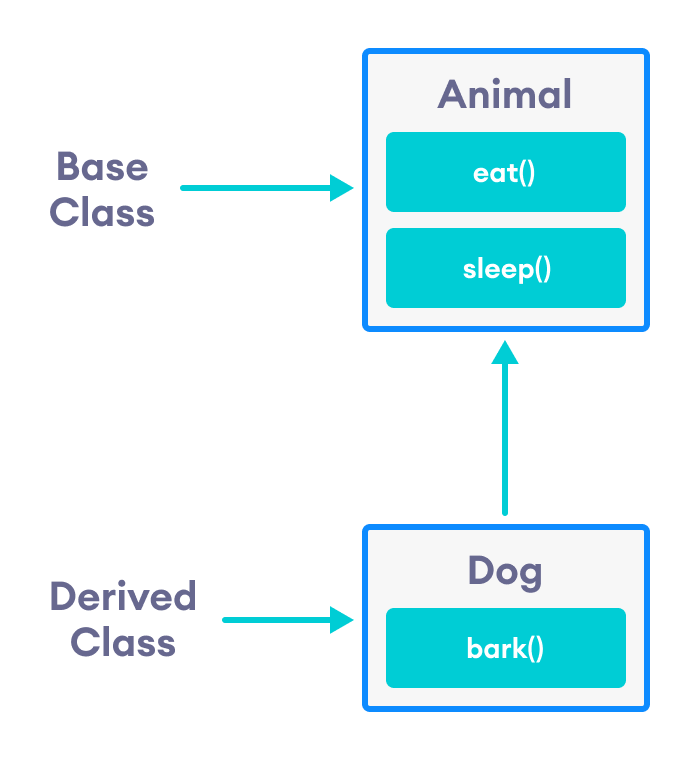
};

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](https://www.programiz.com/cpp-programming/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](https://www.programiz.com/cpp-programming/public-protected-private-inheritance).

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](https://www.programiz.com/cpp-programming/public-protected-private-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++](https://www.programiz.com/cpp-programming/function-overriding).

**Public, Protected and Private Inheritance in C++ Programming**

In this tutorial, we will learn to use public, protected and private inheritance in C++ with the help of examples.

In [C++ inheritance](https://www.programiz.com/cpp-programming/inheritance), we can derive a child class from the base class in different access modes. For example,

class Base {

.... ... ....

};

class Derived : public Base {

.... ... ....

};

Notice the keyword public in the code

class Derived : public Base

This means that we have created a derived class from the base class in **public mode**. Alternatively, we can also derive classes in **protected** or **private** modes.

These 3 keywords (public, protected, and private) are known as **access specifiers** in C++ inheritance.

**public, protected and private inheritance in C++**

**public**, **protected,** and **private** inheritance have the following features:

* **public inheritance** makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.
* **protected inheritance** makes the public and protected members of the base class protected in the derived class.
* **private inheritance** makes the public and protected members of the base class private in the derived class.

**Note:** private members of the base class are inaccessible to the derived class.

class Base {

public:

int x;

protected:

int y;

private:

int z;

};

class PublicDerived: public Base {

// x is public

// y is protected

// z is not accessible from PublicDerived

};

class ProtectedDerived: protected Base {

// x is protected

// y is protected

// z is not accessible from ProtectedDerived

};

class PrivateDerived: private Base {

// x is private

// y is private

// z is not accessible from PrivateDerived

}

**Example 1: C++ public Inheritance**

// C++ program to demonstrate the working of public inheritance

#include <iostream>

using namespace std;

class Base {

private:

int pvt = 1;

protected:

int prot = 2;

public:

int pub = 3;

// function to access private member

int getPVT() {

return pvt;

}

};

class PublicDerived : public Base {

public:

// function to access protected member from Base

int getProt() {

return prot;

}

};

int main() {

PublicDerived object1;

cout << "Private = " << object1.getPVT() << endl;

cout << "Protected = " << object1.getProt() << endl;

cout << "Public = " << object1.pub << endl;

return 0;

}

**Output**

Private = 1

Protected = 2

Public = 3

Here, we have derived PublicDerived from Base in **public mode**.

As a result, in PublicDerived:

* prot is inherited as **protected**.
* pub and getPVT() are inherited as **public**.
* pvt is inaccessible since it is **private** in Base.

Since **private** and **protected** members are not accessible, we need to create public functions getPVT() and getProt() to access them:

// Error: member "Base::pvt" is inaccessible

cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible

cout << "Protected = " << object1.prot;

**Accessibility in Public Inheritance**

|  |  |  |  |
| --- | --- | --- | --- |
| Accessibility | private members | protected members | public members |
| Base Class | Yes | Yes | Yes |
| Derived Class | No | Yes | Yes |

**Example 2: C++ protected Inheritance**

// C++ program to demonstrate the working of protected inheritance

#include <iostream>

using namespace std;

class Base {

private:

int pvt = 1;

protected:

int prot = 2;

public:

int pub = 3;

// function to access private member

int getPVT() {

return pvt;

}

};

class ProtectedDerived : protected Base {

public:

// function to access protected member from Base

int getProt() {

return prot;

}

// function to access public member from Base

int getPub() {

return pub;

}

};

int main() {

ProtectedDerived object1;

cout << "Private cannot be accessed." << endl;

cout << "Protected = " << object1.getProt() << endl;

cout << "Public = " << object1.getPub() << endl;

return 0;

}

Output

Private cannot be accessed.

Protected = 2

Public = 3

Here, we have derived ProtectedDerived from Base in **protected mode**.

As a result, in ProtectedDerived:

* prot, pub and getPVT() are inherited as **protected**.
* pvt is inaccessible since it is **private** in Base.

As we know, **protected** members cannot be accessed directly.

As a result, we cannot use getPVT() from ProtectedDerived. That is also why we need to create the getPub() function in ProtectedDerived in order to access the pub variable.

// Error: member "Base::getPVT()" is inaccessible

cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible

cout << "Public = " << object1.pub;

**Accessibility in Protected Inheritance**

|  |  |  |  |
| --- | --- | --- | --- |
| Accessibility | private members | protected members | public members |
| Base Class | Yes | Yes | Yes |
| Derived Class | No | Yes | Yes (inherited as protected variables) |

**Example 3: C++ private Inheritance**

// C++ program to demonstrate the working of private inheritance

#include <iostream>

using namespace std;

class Base {

private:

int pvt = 1;

protected:

int prot = 2;

public:

int pub = 3;

// function to access private member

int getPVT() {

return pvt;

}

};

class PrivateDerived : private Base {

public:

// function to access protected member from Base

int getProt() {

return prot;

}

// function to access private member

int getPub() {

return pub;

}

};

int main() {

PrivateDerived object1;

cout << "Private cannot be accessed." << endl;

cout << "Protected = " << object1.getProt() << endl;

cout << "Public = " << object1.getPub() << endl;

return 0;

}

**Output**

Private cannot be accessed.

Protected = 2

Public = 3

Here, we have derived PrivateDerived from Base in **private mode**.

As a result, in PrivateDerived:

* prot, pub and getPVT() are inherited as **private**.
* pvt is inaccessible since it is **private** in Base.

As we know, private members cannot be accessed directly.

As a result, we cannot use getPVT() from PrivateDerived. That is also why we need to create the getPub() function in PrivateDerived in order to access the pub variable.

// Error: member "Base::getPVT()" is inaccessible

cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible

cout << "Public = " << object1.pub;

**Accessibility in Private Inheritance**

|  |  |  |  |
| --- | --- | --- | --- |
| Accessibility | private members | protected members | public members |
| Base Class | Yes | Yes | Yes |
| Derived Class | No | Yes (inherited as private variables) | Yes (inherited as private variables) |

# C++ Function Overriding

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

As we know, [inheritance](https://www.programiz.com/cpp-programming/inheritance) is a feature of OOP that allows us to create derived classes from a base class. The derived classes inherit features of the base class.

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.

This is known as **function overriding** in C++. The function in derived class overrides the function in base class.

## Example 1: C++ Function Overriding

// C++ program to demonstrate function overriding

#include <iostream>

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;

derived1.print();

return 0;

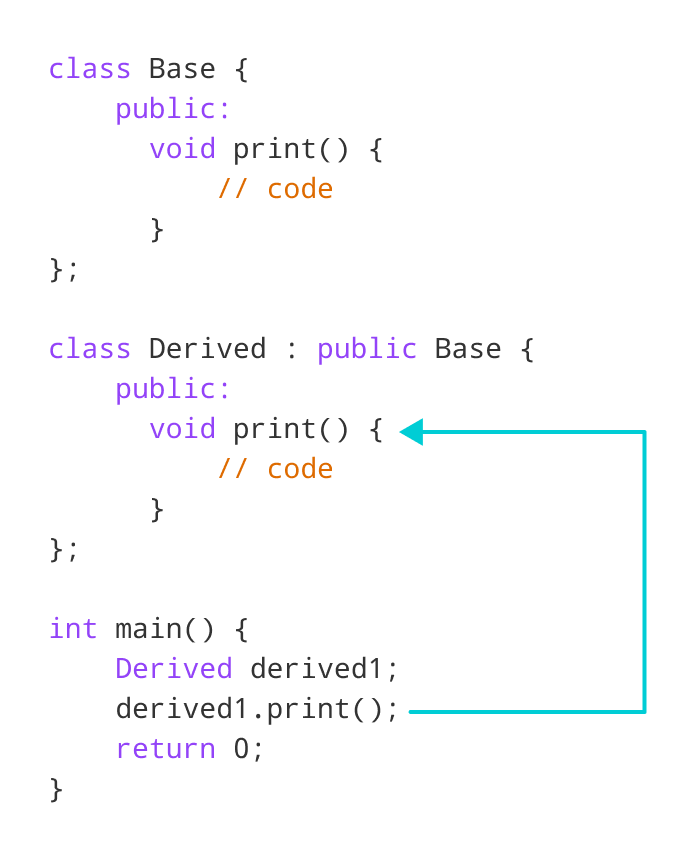
}

**Output**

Derived Function

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.

Working of function overriding in C++

## Access Overridden Function in C++

To access the overridden function of the base class, we use the scope resolution operator ::.

We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

### Example 2: C++ Access Overridden Function to the Base Class

// C++ program to access overridden function

// in main() using the scope resolution operator ::

#include <iostream>

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, derived2;

derived1.print();

// access print() function of the Base class

derived2.Base::print();

return 0;

}

**Output**

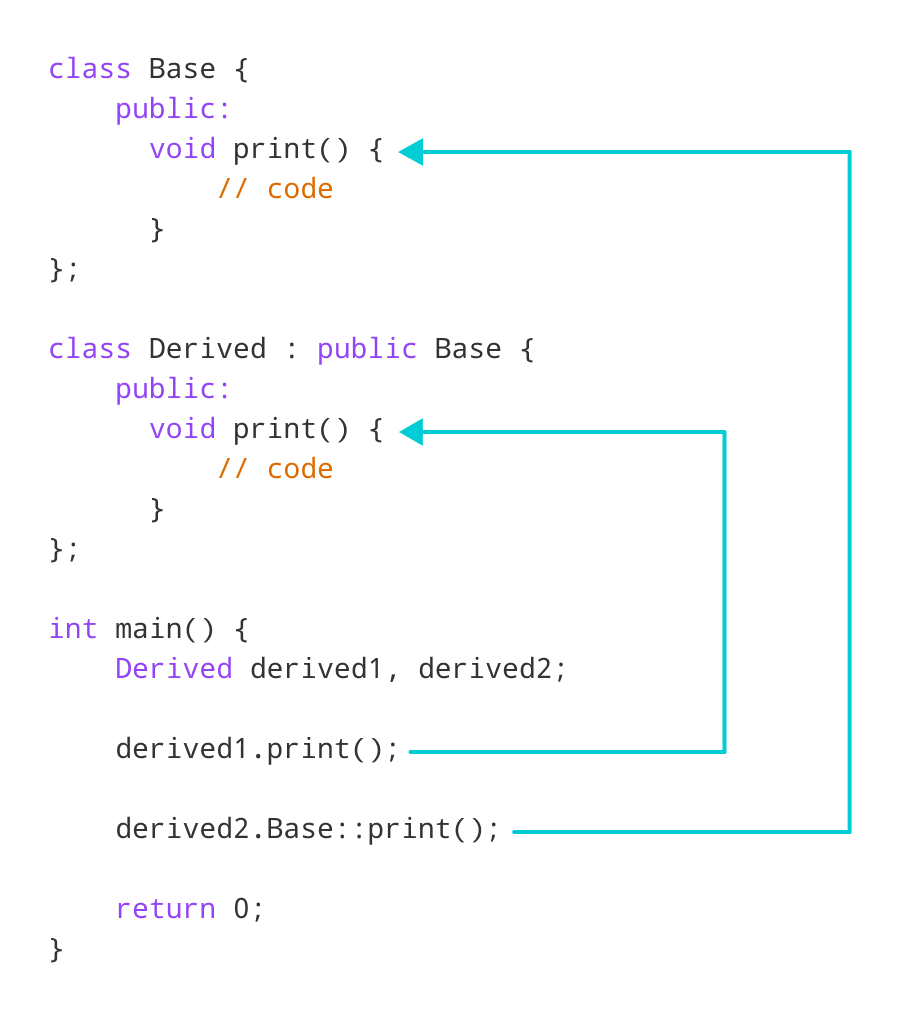
Derived Function

Base Function

Here, this statement

derived2.Base::print();

accesses the print() function of the Base class.

Access overridden function using object of derived class in C++

### Example 3: C++ Call Overridden Function From Derived Class

// C++ program to call the overridden function

// from a member function of the derived class

#include <iostream>

using namespace std;

class Base {

public:

void print() {

cout << "Base Function" << endl;

}

};

class Derived : public Base {

public:

void print() {

cout << "Derived Function" << endl;

// call overridden function

Base::print();

}

};

int main() {

Derived derived1;

derived1.print();

return 0;

}

**Output**

Derived Function

Base Function

In this program, we have called the overridden function inside the Derived class itself.

class Derived : public Base {

public:

void print() {

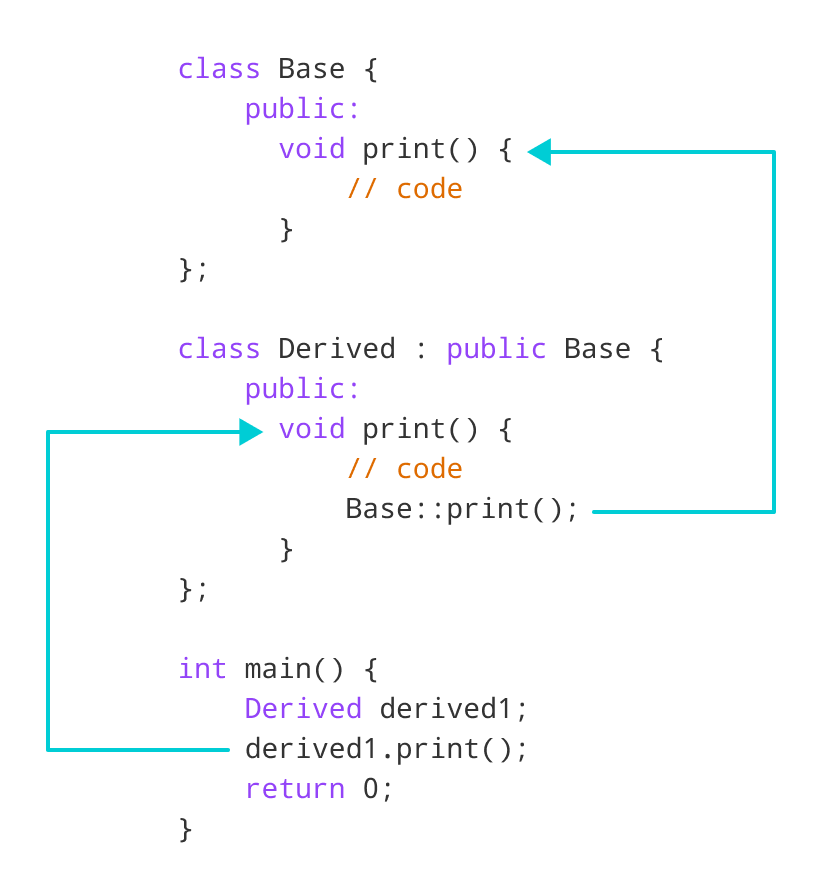
cout << "Derived Function" << endl;

Base::print();

}

};

Notice the code Base::print();, which calls the overridden function inside the Derived class.

Access overridden function inside derived class in C++

### Example 4: C++ Call Overridden Function Using Pointer

// C++ program to access overridden function using pointer

// of Base type that points to an object of Derived class

#include <iostream>

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;

}

**Output**

Base Function

In this program, we have created a pointer of Base type named ptr. This pointer points to the Derived object derived1.

// pointer of Base type that points to derived1

Base\* ptr = &derived1;

When we call the print() function using ptr, it calls the overridden function from Base.

// call function of Base class using ptr

ptr->print();

This is because even though ptr points to a Derived object, it is actually of Base type. So, it calls the member function of Base.

In order to override the Base function instead of accessing it, we need to use [virtual functions](https://www.programiz.com/cpp-programming/virtual-functions) in the Base class.

# C++ Multiple, Multilevel and Hierarchical Inheritance

#### In this article, you will learn about different models of inheritance in C++ programming: Multiple, Multilevel and Hierarchical inheritance with examples.

[Inheritance](https://www.programiz.com/cpp-programming/inheritance) is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

There are various models of inheritance in C++ programming.

## C++ Multilevel Inheritance

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

class A

{

... .. ...

};

class B: public A

{

... .. ...

};

class C: public B

{

... ... ...

};

Here, class B is derived from the base class A and the class C is derived from the derived class B.

## Example 1: C++ Multilevel Inheritance

#include <iostream>

using namespace std;

class A

{

public:

void display()

{

cout<<"Base class content.";

}

};

class B : public A

{

};

class C : public B

{

};

int main()

{

C obj;

obj.display();

return 0;

}

**Output**

Base class content.

In this program, class C is derived from class B (which is derived from base class A).

The obj object of class C is defined in the main() function.

When the display() function is called, display() in class A is executed. It's because there is no display() function in class C and class B.

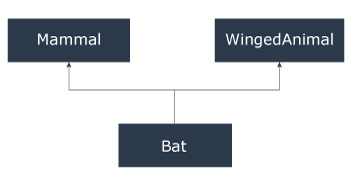
The compiler first looks for the display() function in class C. Since the function doesn't exist there, it looks for the function in class B (as C is derived from B).

The function also doesn't exist in class B, so the compiler looks for it in class A (as B is derived from A).

If display() function exists in C, the compiler overrides display() of class A (because of [member function overriding](https://www.programiz.com/cpp-programming/function-overriding)).

## C++ Multiple Inheritance

In C++ programming, a class can be derived from more than one parents. For example: A class Bat is derived from base classes Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.



## Example 2: Multiple Inheritance in C++ Programming

#include <iostream>

using namespace std;

class Mammal {

public:

Mammal()

{

cout << "Mammals can give direct birth." << endl;

}

};

class WingedAnimal {

public:

WingedAnimal()

{

cout << "Winged animal can flap." << endl;

}

};

class Bat: public Mammal, public WingedAnimal {

};

int main()

{

Bat b1;

return 0;

}

**Output**

Mammals can give direct birth.

Winged animal can flap.

### Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

class base1

{

public:

void someFunction( )

{ .... ... .... }

};

class base2

{

void someFunction( )

{ .... ... .... }

};

class derived : public base1, public base2

{

};

int main()

{

derived obj;

obj.someFunction() // Error!

}

This problem can be solved using scope resolution function to specify which function to class either base1or base2

int main()

{

obj.base1::someFunction( ); // Function of base1 class is called

obj.base2::someFunction(); // Function of base2 class is called.

}

## C++ Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as [hierarchical inheritance](http://www.programtopia.net/cplusplus/docs/hierarchical-inheritance-c-programming?utm_source=programiz&utm_campaign=display). In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example: Physics, Chemistry, Biology are derived from Science class.

### Syntax of Hierarchical Inheritance

class base\_class {

... .. ...

}

class first\_derived\_class: public base\_class {

... .. ...

}

class second\_derived\_class: public base\_class {

... .. ...

}

class third\_derived\_class: public base\_class {

... .. ...

}

# C++ friend Function and friend Classes

#### In this tutorial, we will learn to create friend functions and friend classes in C++ with the help of examples.

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

class MyClass {

private:

int member1;

}

int main() {

MyClass obj;

// Error! Cannot access private members from here.

obj.member1 = 5;

}

However, there is a feature in C++ called **friend functions** that break this rule and allow us to access member functions from outside the class.

Similarly, there is a **friend class** as well, which we will learn later in this tutorial.

## friend Function in C++

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the friend keyword inside the body of the class.

class className {

... .. ...

friend returnType functionName(arguments);

... .. ...

}

### Example 1: Working of friend Function

// C++ program to demonstrate the working of friend function

#include <iostream>

using namespace std;

class Distance {

private:

int meter;

// friend function

friend int addFive(Distance);

public:

Distance() : meter(0) {}

};

// friend function definition

int addFive(Distance d) {

//accessing private members from the friend function

d.meter += 5;

return d.meter;

}

int main() {

Distance D;

cout << "Distance: " << addFive(D);

return 0;

}

**Output**

Distance: 5

Here, addFive() is a friend function that can access both **private** and **public** data members.

Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use.

A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

### Example 2: Add Members of Two Different Classes

// Add members of two different classes using friend functions

#include <iostream>

using namespace std;

// forward declaration

class ClassB;

class ClassA {

public:

// constructor to initialize numA to 12

ClassA() : numA(12) {}

private:

int numA;

// friend function declaration

friend int add(ClassA, ClassB);

};

class ClassB {

public:

// constructor to initialize numB to 1

ClassB() : numB(1) {}

private:

int numB;

// friend function declaration

friend int add(ClassA, ClassB);

};

// access members of both classes

int add(ClassA objectA, ClassB objectB) {

return (objectA.numA + objectB.numB);

}

int main() {

ClassA objectA;

ClassB objectB;

cout << "Sum: " << add(objectA, objectB);

return 0;

}

**Output**

Sum: 13

In this program, ClassA and ClassB have declared add() as a friend function. Thus, this function can access **private** data of both classes.

One thing to notice here is the friend function inside ClassA is using the ClassB. However, we haven't defined ClassB at this point.

// inside classA

friend int add(ClassA, ClassB);

For this to work, we need a forward declaration of ClassB in our program.

// forward declaration

class ClassB;

## friend Class in C++

We can also use a friend Class in C++ using the friend keyword. For example,

class ClassB;

class ClassA {

// ClassB is a friend class of ClassA

friend class ClassB;

... .. ...

}

class ClassB {

... .. ...

}

When a class is declared a friend class, all the member functions of the friend class become friend functions.

Since classB is a friend class, we can access all members of classA from inside classB.

However, we cannot access members of ClassB from inside classA. It is because friend relation in C++ is only granted, not taken.

### Example 3: C++ friend Class

// C++ program to demonstrate the working of friend class

#include <iostream>

using namespace std;

// forward declaration

class ClassB;

class ClassA {

private:

int numA;

// friend class declaration

friend class ClassB;

public:

// constructor to initialize numA to 12

ClassA() : numA(12) {}

};

class ClassB {

private:

int numB;

public:

// constructor to initialize numB to 1

ClassB() : numB(1) {}

// member function to add numA

// from ClassA and numB from ClassB

int add() {

ClassA objectA;

return objectA.numA + numB;

}

};

int main() {

ClassB objectB;

cout << "Sum: " << objectB.add();

return 0;

}

**Output**

Sum: 13

Here, ClassB is a friend class of ClassA. So, ClassB has access to the members of classA.

In ClassB, we have created a function add() that returns the sum of numA and numB.

Since ClassB is a friend class, we can create objects of ClassA inside of ClassB.

**C++ Virtual Functions**

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](https://www.programiz.com/cpp-programming/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;

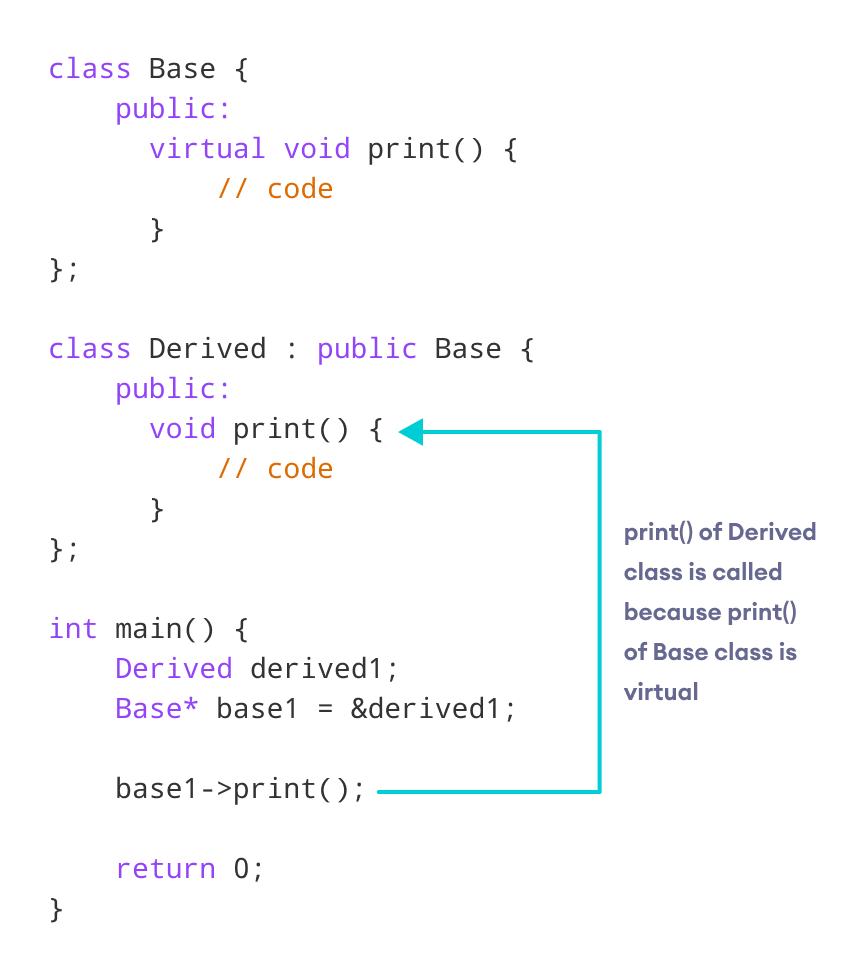
}

**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.

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().

**C++ Templates**

In this article, you'll learn about templates in C++. You'll learn to use the power of templates for generic programming.

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

* Function Templates
* Class Templates

**Function Templates**

A function template works in a similar to a normal [function](https://www.programiz.com/cpp-programming/function), with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

**How to declare a function template?**

A function template starts with the keyword **template** followed by template parameter/s inside  **< >** which is followed by function declaration.

**template** <**class** T>

T someFunction(T arg)

{

... .. ...

}

In the above code, T is a template argument that accepts different data types (int, float), and **class** is a keyword.

You can also use keyword **typename** instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

**Example 1: Function Template to find the largest number**

**Program to display largest among two numbers using function templates.**

// If two characters are passed to function template, character with larger ASCII value is displayed.

#include <iostream>

using namespace std;

// template function

template <class T>

T Large(T n1, T n2)

{

return (n1 > n2) ? n1 : n2;

}

int main()

{

int i1, i2;

float f1, f2;

char c1, c2;

cout << "Enter two integers:\n";

cin >> i1 >> i2;

cout << Large(i1, i2) <<" is larger." << endl;

cout << "\nEnter two floating-point numbers:\n";

cin >> f1 >> f2;

cout << Large(f1, f2) <<" is larger." << endl;

cout << "\nEnter two characters:\n";

cin >> c1 >> c2;

cout << Large(c1, c2) << " has larger ASCII value.";

return 0;

}

**Output**

Enter two integers:

5

10

10 is larger.

Enter two floating-point numbers:

12.4

10.2

12.4 is larger.

Enter two characters:

z

Z

z has larger ASCII value.

In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.

Large() function returns the largest among the two arguments using a simple [conditional operation](https://www.programiz.com/cpp-programming/if-else).

Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.

Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

**Example 2: Swap Data Using Function Templates**

**Program to swap data using function templates.**

#include <iostream>

using namespace std;

template <typename T>

void Swap(T &n1, T &n2)

{

T temp;

temp = n1;

n1 = n2;

n2 = temp;

}

int main()

{

int i1 = 1, i2 = 2;

float f1 = 1.1, f2 = 2.2;

char c1 = 'a', c2 = 'b';

cout << "Before passing data to function template.\n";

cout << "i1 = " << i1 << "\ni2 = " << i2;

cout << "\nf1 = " << f1 << "\nf2 = " << f2;

cout << "\nc1 = " << c1 << "\nc2 = " << c2;

Swap(i1, i2);

Swap(f1, f2);

Swap(c1, c2);

cout << "\n\nAfter passing data to function template.\n";

cout << "i1 = " << i1 << "\ni2 = " << i2;

cout << "\nf1 = " << f1 << "\nf2 = " << f2;

cout << "\nc1 = " << c1 << "\nc2 = " << c2;

return 0;

}

**Output**

Before passing data to function template.

i1 = 1

i2 = 2

f1 = 1.1

f2 = 2.2

c1 = a

c2 = b

After passing data to function template.

i1 = 2

i2 = 1

f1 = 2.2

f2 = 1.1

c1 = b

c2 = a

In this program, instead of calling a function by passing a value, a [call by reference](https://www.programiz.com/cpp-programming/pointers-function) is issued.

The Swap() function template takes two arguments and swaps them by reference.

**Class Templates**

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

**How to declare a class template?**

**template** <**class** T>

class className

{

... .. ...

public:

T var;

T someOperation(T arg);

... .. ...

};

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

**How to create a class template object?**

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType> classObject;

For example:

className<int> classObject;

className<float> classObject;

className<string> classObject;

**Example 3: Simple calculator using Class template**

Program to add, subtract, multiply and divide two numbers using class template

#include <iostream>

using namespace std;

template <class T>

class Calculator

{

private:

T num1, num2;

public:

Calculator(T n1, T n2)

{

num1 = n1;

num2 = n2;

}

void displayResult()

{

cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;

cout << "Addition is: " << add() << endl;

cout << "Subtraction is: " << subtract() << endl;

cout << "Product is: " << multiply() << endl;

cout << "Division is: " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main()

{

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl << "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

**Output**

Int results:

Numbers are: 2 and 1.

Addition is: 3

Subtraction is: 1

Product is: 2

Division is: 2

Float results:

Numbers are: 2.4 and 1.2.

Addition is: 3.6

Subtraction is: 1.2

Product is: 2.88

Division is: 2

In the above program, a class template Calculator is declared.

The class contains two private members of type T: num1 & num2, and a constructor to initalize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function displayResult() to display the final output to the screen.

In the main() function, two different Calculator objects intCalc and floatCalc are created for data types: int and float respectively. The values are initialized using the constructor.

Notice we use <int> and <float> while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for int and float, which are then used accordingly.

Then, displayResult() of both objects is called which performs the Calculator operations and displays the output.