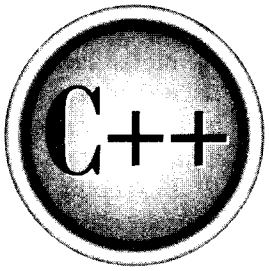
The Complete Reference



Chapter 16

Inheritance

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

In keeping with standard C++ terminology, a class that is inherited is referred to as a *base class*. The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.

C++'s support of inheritance is both rich and flexible. Inheritance was introduced in Chapter 11. It is examined in detail here.

Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name {
  // body of class
};
```

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public**, **private**, or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. Let's examine the ramifications of using **public** or **private** access. (The **protected** specifier is examined in the next section.)

When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class. For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**:

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

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
```

```
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // access member of base
  ob.show(); // access member of base

  ob.showk(); // uses member of derived class

  return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class. For example, the following program will not even compile because both **set( )** and **show( )** are now private elements of **derived**:

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n";}
};

// Public elements of base are private in derived.
class derived : private base {
  int k;
```

```
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // error, can't access set()
  ob.show(); // error, can't access show()

  return 0;
}
```

Remember *When a base class' access specifier is* ***private****, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.*

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. Here is an example:

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

class base {
```

```
protected:
  int i, j; // private to base, but accessible by derived
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  // derived may access base's i and j
  void setk() { k=i*j; }

  void showk() { cout << k << "\n"; }
};

int main()
{
  derived cb;

  ob.set(2, 3); // OK, known to derived
  ob.show(); // OK, known to derived

  ob.setk();
  ob.showk();

  return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected**, **derived**'s function **setk( )** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j**.

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

class base {
```

```
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// i and j inherited as protected.
class derived1 : public base {
  int k;
public:
  void setk() { k = i*j; } // legal
  void showk() { cout << k << "\n"; }
};

// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
  int m;
public:
  void setm() { m = i-j; } // legal
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(2, 3);
  ob1.show();
  ob1.setk();
  ob1.showk();

  ob2.set(3, 4);
  ob2.show();
  ob2.setk();
  ob2.setm();
  ob2.showk();
  ob2.showm();

  return 0;
}
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// Now, all elements of base are private in derived1.
class derived1 : private base {
  int k;
public:
  // this is legal because i and j are private to derived1
  void setk() { k = i*j; } // OK
  void showk() { cout << k << "\n"; }
};

// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
  int m;
public:
  // illegal because i and j are private to derived1
  void setm() { m = i-j; } // Error
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // error, can't use set()
```

```
  ob1.show(); // error, can't use show()

  ob2.set(3, 4); // error, can't use set()
  ob2.show(); // error, can't use show()

  return 0;
}
```

Note *Even though* ***base*** *is inherited as* ***private*** *by* ***derived1****,* ***derived1*** *still has access to* ***base****'s* ***public*** *and* ***protected*** *elements. However, it cannot pass along this privilege.*

Protected Base-Class Inheritance

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class. For example,

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

class base {
protected:
  int i, j; // private to base, but accessible by derived
public:
  void setij(int a, int b) { i=a; j=b; }
  void showij() { cout << i << " " << j << "\n"; }
};

// Inherit base as protected.
class derived : protected base{
  int k;
public:
  // derived may access base's i and j and setij().
  void setk() { setij(10, 12); k = i*j; }

  // may access showij() here
  void showall() { cout << k << " "; showij(); }
};

int main()
```

```
{
  derived ob;

//  ob.setij(2, 3); // illegal, setij() is
//                     protected member of derived

  ob.setk(); // OK, public member of derived
  ob.showall(); // OK, public member of derived

// ob.showij(); // illegal, showij() is protected
//                 member of derived

  return 0;
}
```

As you can see by reading the comments, even though **setij( )** and **showij( )** are public members of **base**, they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main( )**.

Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.

#include <iostream>
using namespace std;

class base1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};

class base2 {
protected:
  int y;
public:
```

```
  void showy() {cout << y << "\n";}
};

// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
  void set(int i, int j) { x=i; y=j; }
};

int main()
{
  derived ob;

  ob.set(10, 20); // provided by derived
  ob.showx(); // from base1
  ob.showy(); // from base2

  return 0;
}
```

As the example illustrates, to inherit more than one base class, use a comma-separated list. Further, be sure to use an access-specifier for each base inherited.

Constructors, Destructors, and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructors and destructors called? Second, how can parameters be passed to base-class constructors? This section examines these two important topics.

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence. To begin, examine this short program:

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

```
class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // do nothing but construct and destruct ob

  return 0;
}
```

As the comment in **main( )** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, first **base**'s constructor is executed followed by **derived**'s. Next (because **ob** is immediately destroyed in this program), **derived**'s destructor is called, followed by **base**'s.

The results of the foregoing experiment can be generalized. When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor. Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any

initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed.

In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

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

class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived1 : public base {
public:
  derived1() { cout << "Constructing derived1\n"; }
  ~derived1() { cout << "Destructing derived1\n"; }
};

class derived2: public derived1 {
public:
  derived2() { cout << "Constructing derived2\n"; }
  ~derived2() { cout << "Destructing derived2\n"; }
};

int main()
{
  derived2 ob;

  // construct and destruct ob

  return 0;
}
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

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

class base1 {
public:
  base1() { cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
public:
  base2() { cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // construct and destruct ob

  return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left. This means that had **base2** been specified before **base1** in **derived**'s list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructors that require arguments. In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax (see Chapter 12). However, how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                //...
                                baseN(arg-list)
{
  // body of derived constructor
}
```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

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

class base {
protected:
  int i;
public:
  base(int x) { i=x; cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
  int j;
public:
  // derived uses x; y is passed along to base.
  derived(int x, int y): base(y)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 4 3

  return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived( )** uses only **x**; **y** is passed along to **base( )**. In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

Here is an example that uses multiple base classes:

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

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base1\n"; }
};

class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4, 5);

  ob.show(); // displays 4 3 5

  return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class

requires it. In this situation, the arguments passed to the derived class are simply passed along to the base. For example, in this program, the derived class' constructor takes no arguments, but **base1( )** and **base2( )** do:

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

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  /* Derived constructor uses no parameter,
     but still must be declared as taking them to
     pass them along to base classes.
  */

  derived(int x, int y): base1(x), base2(y)
    { cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 3 4
```

```
  return 0;
}
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
  int j;
public:
  // derived uses both x and y and then passes them to base.
  derived(int x, int y): base(x, y)
    { j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

Granting Access

When a base class is inherited as **private**, all public and protected members of that class become private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification. For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**. In Standard C++, you have two ways to accomplish this. First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces and is discussed in Chapter 23. The second way to restore an inherited member's access specification is to employ an *access declaration* within the derived class. Access declarations are currently supported by Standard C++, but they are deprecated. This means that they should not be used for new code. Since there are still many, many existing programs that use access declarations, they will be examined here.

An access declaration takes this general form:

base-class::member;

The access declaration is put under the appropriate access heading in the derived class' declaration. Notice that no type declaration is required (or, indeed, allowed) in an access declaration.

To see how an access declaration works, let's begin with this short fragment:

```
class base {
public:
  int j; // public in base
};

// Inherit base as private.
class derived: private base {
public:

  // here is access declaration
  base::j; // make j public again
  .
  .
  .
};
```

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

```
base::j;
```

as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.

You can use an access declaration to restore the access rights of public and protected members. However, you cannot use an access declaration to raise or lower a member's access status. For example, a member declared as private in a base class cannot be made public by a derived class. (If C++ allowed this to occur, it would destroy its encapsulation mechanism!)

The following program illustrates the access declaration; notice how it uses access declarations to restore **j**, **seti( )**, and **geti( )** to **public** status.

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

class base {
  int i; // private to base
```

```
public:
  int j, k;
  void seti(int x) { i = x; }
  int geti() { return i; }
};

// Inherit base as private.
class derived: private base {
public:
  /* The next three statements override
     base's inheritance as private and restore j,
     seti(), and geti() to public access. */
  base::j; // make j public again - but not k
  base::seti; // make seti() public
  base::geti; // make geti() public

// base::i; // illegal, you cannot elevate access

  int a; // public
};

int main()
{
  derived ob;

//ob.i = 10; // illegal because i is private in derived

  ob.j = 20; // legal because j is made public in derived
//ob.k = 30; // illegal because k is private in derived

  ob.a = 40; // legal because a is public in derived
  ob.seti(10);

  cout << ob.geti() << " " << ob.j << " " << ob.a;

  return 0;
}
```

Access declarations are supported in C++ to accommodate those situations in which most of an inherited class is intended to be made private, but a few members are to retain their public or protected status.

Remember *While Standard C++ still supports access declarations, they are deprecated. This means that they are allowed for now, but they might not be supported in the future. Instead, the standard suggests achieving the same effect by applying the* ***using*** *keyword.*

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
```

```
  derived3 ob;

  ob.i = 10; // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;

  // i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;

  // also ambiguous, which i?
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

```
ob.i = 10;
```

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
```

```
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.derived1::i = 10; // scope resolved, use derived1's i
  ob.j = 20;
  ob.k = 30;

  // scope resolved
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // also resolved here
  cout << ob.derived1::i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As you can see, because the :: was applied, the program has manually selected **derived1**'s version of **base**. However, this solution raises a deeper issue: What if only one copy of **base** is actually required? Is there some way to prevent two copies from

being included in **derived3**? The answer, as you probably have guessed, is yes. This solution is achieved using **virtual** base classes.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base**:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // now unambiguous
```

```
  ob.j = 20;
  ob.k = 30;

  // unambiguous
  ob.sum = ob.i + ob.j + ob.k;

  // unambiguous
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous.

One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;

myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.