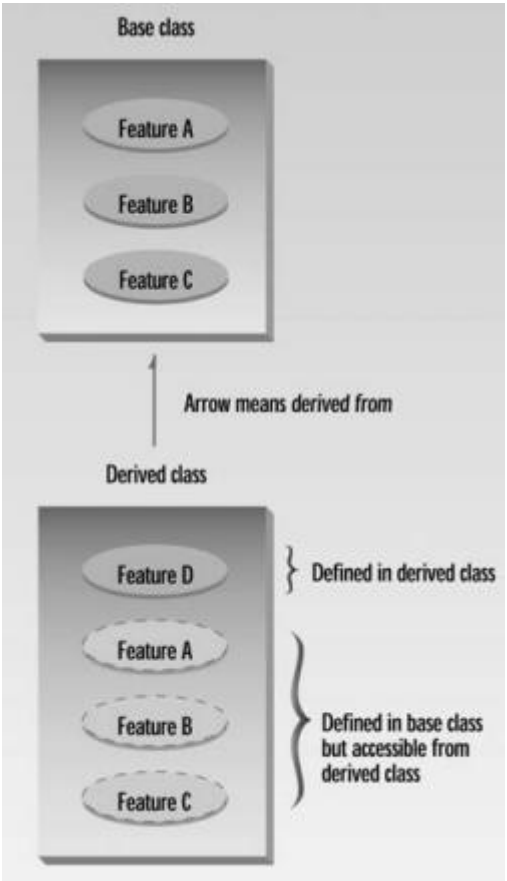
Inheritance

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. Inheritance is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure.



The arrow in Figure goes in the opposite direction of what you might expect. If it pointed down we would label it inheritance. However, the more common approach is to point the arrow up, from the derived class to the base class, and to think of it as a "derived from" arrow. Inheritance is an essential part of OOP. Its big payoff is that it permits code reusability. Once a base class is written and debugged, it need not be touched again,

but, using inheritance, can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability. Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program. An important result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations. We'll examine these features of inheritance in more detail after we've seen some specific instances of inheritance at work.

Derived Class and Base Class

Let's suppose that we have worked long and hard to make the Counter class operate just the way we want, and we're pleased with the results, except for one thing. We really need a way to decrement the count. Perhaps we're counting people entering a bank, and we want to increment the count when they come in and decrement it when they go out, so that the count represents the number of people in the bank at any moment. We could insert a decrement routine directly into the source code of the Counter class. However, there are several reasons that we might not want to do this. **First,** the Counter class works very well and has undergone many hours of testing and debugging. (Of course that's an exaggeration in this case, but it would be true in a larger and more complex class.) If we start fooling around with the source code for Counter, the testing process will need to be carried out again, and of course we may foul something up and spend hours debugging code that worked fine before we modified it.

Second In some situations there might be another reason for not modifying the Counter class: We might not have access to its source code,

especially if it was distributed as part of a class library. To avoid these problems we can use inheritance to create a new class based on Counter, without modifying Counter itself. Here's the listing for COUNTEN, which includes a new class, CountDn, that adds a decrement operator to the Counter class:

```
// inheritance with Counter class
#include <iostream>
class Counter //base class
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count(0) //no-arg constructor
{ }
Counter(int c) : count(c) //1-arg constructor
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
{ return Counter(++count); }
};
////////////////////////////////////////////////////////////
class CountDn : public Counter //derived class
{
public:
Counter operator -- () //decr count (prefix)
{ return Counter(--count); }
};
```

```
//////////////////////////////////////////////////////////
int main()
{
CountDn c1; //c1 of class CountDn
cout << "\nc1=" << c1.get_count(); //display c1
++c1; ++c1; ++c1; //increment c1, 3 times
cout << "\nc1=" << c1.get_count(); //display it
--c1; --c1; //decrement c1, twice
cout << "\nc1=" << c1.get_count(); //display it
cout << endl;
return 0;
}
```

Specifying the Derived Class

Following the Counter class in the listing is the specification for a new class, CountDn. This class incorporates a new function, operator--(), which decrements the count. However—and here's the key point—the new CountDn class inherits all the features of the Counter class. CountDn doesn't need a constructor or the get_count() or operator++() functions, because these already exist in Counter.
The first line of CountDn specifies that it is derived from Counter:
class CountDn : public Counter

Here we use a single colon (not the double colon used for the scope resolution operator), followed by the keyword public and the name of the base class Counter. This sets up the relationship between the classes. This line says that CountDn is derived from the base class Counter. (We'll explore the effect of the keyword public later.)

Accessing Base Class Members

An important topic in inheritance is knowing when a member function in the base class can be used by objects of the derived class. This is called **accessibility**. Let's see how the compiler handles the accessibility issue in the COUNTEN example.

Substituting Base Class Constructors

In the main() part of COUNTEN we create an object of class CountDn: CountDn c1;

This causes c1 to be created as an object of class CountDn and initialized to 0. But wait—how is this possible? There is no constructor in the CountDn class specifier, so what entity carries out the initialization? It turns out that—at least under certain circumstances—if you don't specify a constructor, the derived class will use an appropriate constructor from the base class. In COUNTEN there's no constructor in CountDn, so the compiler uses the no-argument constructor from Count.

This flexibility on the part of the compiler—using one function because another isn't available— appears regularly in inheritance situations. Generally, the substitution is what you want, but sometimes it can be unnerving.

Substituting Base Class Member Functions

The object c1 of the CountDn class also uses the operator++() and get_count() functions from the Counter class. The first is used to increment c1:

```
++c1;
```

The second is used to display the count in c1:

```
cout << "\nc1=" << c1.get_count();
```

Again the compiler, not finding these functions in the class of which c1 is a member, uses member functions from the base class.

Output of COUNTEN

In main() we increment c1 three times, print out the resulting value, decrement c1 twice, and finally print out its value again. Here's the output:

```
c1=0 ←□□□ after initialization
c1=3 ←□□□ after ++c1, ++c1, ++c1
c1=1 ←□□□ after --c1, --c1
```

The ++ operator, the constructors, the get_count() function in the Counter class, and the -- operator in the CountDn class all work with objects of type CountDn.

The protected Access Specifier

We have increased the functionality of a class without modifying it. Well, almost without modifying it. Let's look at the single change we made to the Counter class.

The data in the classes we've looked at so far, including count in the Counter class in the earlier COUNTPP3 program, have used the private access specifier.

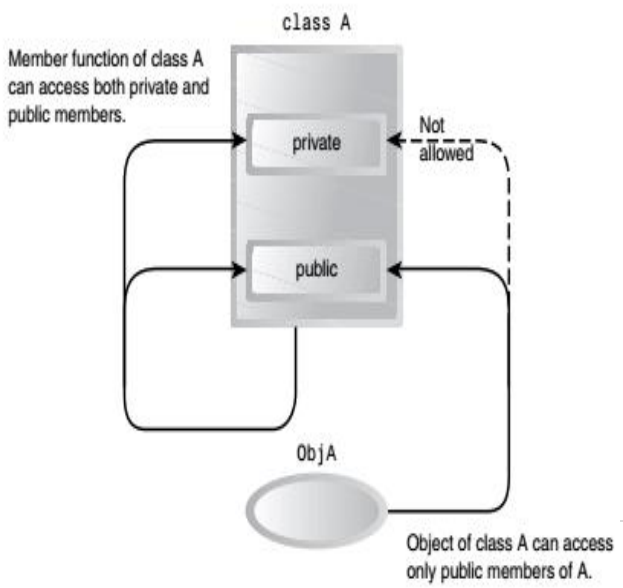
In the Counter class in COUNTEN, count is given a new specifier: protected. What does this do?

Let's first review what we know about the access specifiers private and public. A member function of a class can always access class members, whether they are public or private. But an object declared externally can only invoke (using the dot operator, for example) public members of the class. It's not allowed to use private members. For instance, suppose

an object objA is an instance of class A, and function funcA() is a member function of A. Then in main() (or any other function that is not a member of A) the statement objA.funcA();

will not be legal unless funcA() is public. The object objA cannot invoke private members of class A. Private members are, well, private. This is shown in Figure 9.3.

This is all we need to know if we don't use inheritance. With inheritance, however, there is awhole raft of additional possibilities. The question that concerns us at the moment is, can member functions of the derived class access members of the base class? In other words, can operator--() in CountDn access count in Counter? The answer is that member functions can access members of the base class if the members are public, or if they are protected. They can't access private members. We don't want to make count public, since that would allow it to be accessed by any function anywhere in the program and eliminate the advantages of data hiding. A protected member, on the other hand, can be accessed by member functions in its own class or—and here's the key—in any class derived from its own class. It can't be accessed from functions outside these classes, such as main(). This is just what we want. The situation is shown in Figure 9.4



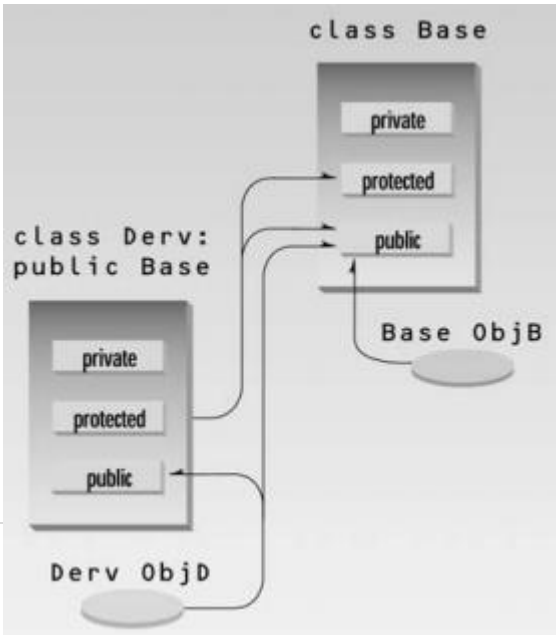


TABLE 9.1 Inheritance and Accessibility

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

The moral is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any member data that the derived classes might need to access should be made protected rather than private. This ensures that the class is "inheritance ready."