Inheritance

CHAPTER

9

IN THIS CHAPTER

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 9.1.

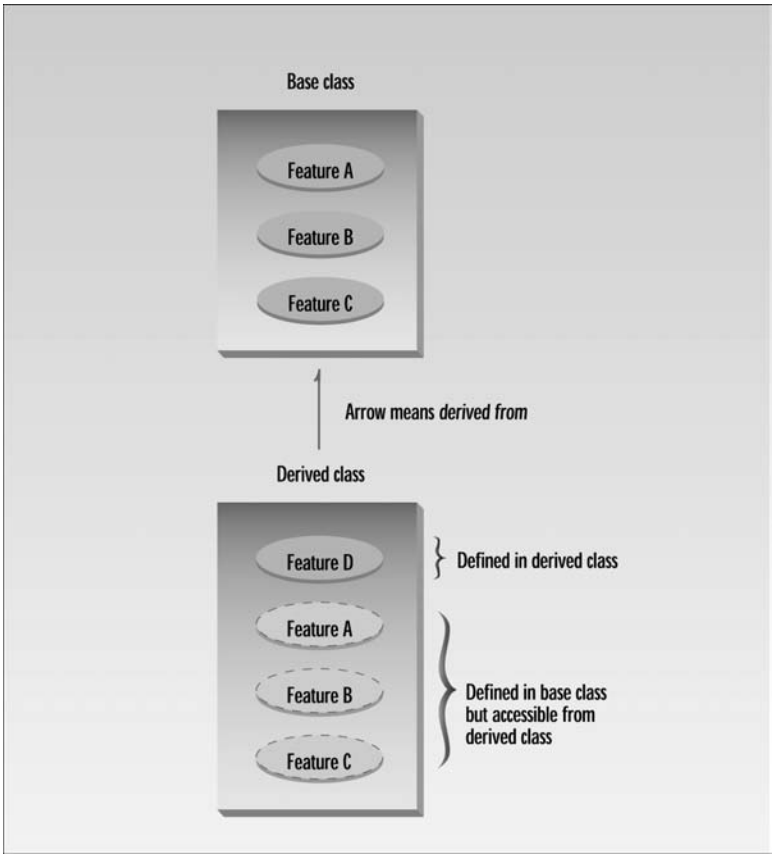

FIGURE 9.1
Inheritance.

The arrow in Figure 9.1 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

Remember the COUNTPP3 example from Chapter 8, "Operator Overloading"? This program used a class `Counter` as a general-purpose counter variable. A count could be initialized to 0 or to a specified number with constructors, incremented with the `++` operator, and read with the `get_count()` operator.

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.

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. (We'll discuss this issue further in Chapter 13, "Multifile Programs.")

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:

```
// counten.cpp
// inheritance with Counter class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
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;
   }
```

The listing starts off with the `Counter` class, which (with one small exception, which we'll look at later) has not changed since its appearance in COUNTPP3. Notice that, for simplicity, we haven't modeled this program on the POSTFIX program, which incorporated the second overloaded ++ operator to provide postfix notation.

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.)

Generalization in UML Class Diagrams

In the UML, inheritance is called *generalization*, because the parent class is a more general form of the child class. Or to put it another way, the child is more specific version of the parent. (We introduced the UML in Chapter 1, "The Big Picture," and encountered class diagrams in Chapter 8, "Operator Overloading.") The generalization in the COUNTEN program is shown in Figure 9.2.

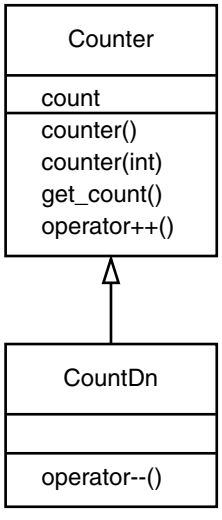

FIGURE 9.2
UML class diagram for COUNTEN.

In UML class diagrams, generalization is indicated by a triangular arrowhead on the line connecting the parent and child classes. Remember that the arrow means *inherited from* or *derived from* or *is a more specific version of*. The direction of the arrow emphasizes that the derived class *refers to* functions and data in the base class, while the base class has no access to the derived class.

Notice that we've added attributes (member data) and operations (member functions) to the classes in the diagram. The top area holds the class title, the middle area holds attributes, and the bottom area is for operations.

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 a whole 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.

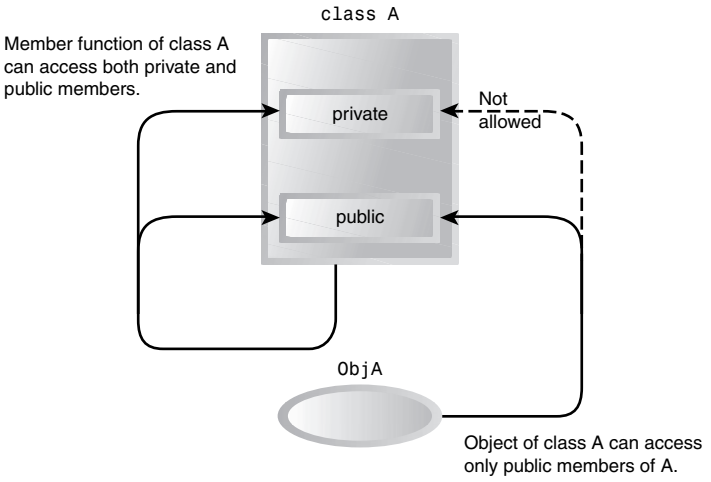

FIGURE 9.3
Access specifiers without inheritance.

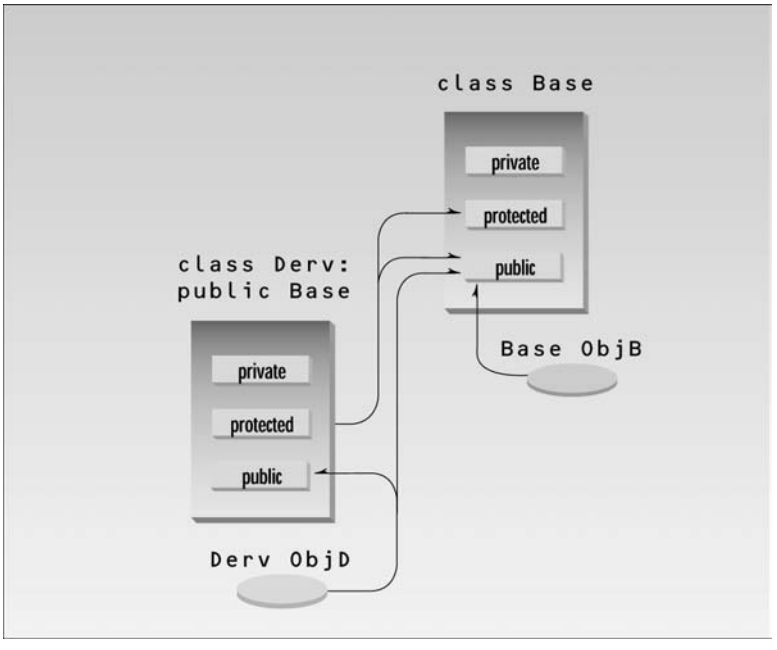

FIGURE 9.4
Access specifiers with inheritance.

Table 9.1 summarizes the situation in a different way.

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."

Dangers of `protected`

You should know that there's a disadvantage to making class members `protected`. Say you've written a class library, which you're distributing to the public. Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them. This makes protected members considerably less secure than private members. To avoid corrupted data, it's often safer to force derived classes to access data in the base class using only public functions in the base class, just as ordinary `main()` programs must do. Using the `protected` specifier leads to simpler programming, so we rely on it—perhaps a bit too much—in the examples in this book. You'll need to weigh the advantages of `protected` against its disadvantages in your own programs.

Base Class Unchanged

Remember that, even if other classes have been derived from it, the base class remains unchanged. In the `main()` part of COUNTEN, we could define objects of type `Counter`:

```
Counter c2;      <------- object of base class
```

Such objects would behave just as they would if `CountDn` didn't exist.

Note also that inheritance doesn't work in reverse. The base class and its objects don't know anything about any classes derived from the base class. In this example that means that objects of class `Counter`, such as `c2`, can't use the `operator--()` function in `CountDn`. If you want a counter that you can decrement, it must be of class `CountDn`, not `Counter`.

Other Terms

In some languages the base class is called the *superclass* and the derived class is called the *subclass*. Some writers also refer to the base class as the *parent* and the derived class as the *child*.

Derived Class Constructors

There's a potential glitch in the COUNTEN program. What happens if we want to initialize a `CountDn` object to a value? Can the one-argument constructor in `Counter` be used? The answer is no. As we saw in COUNTEN, the compiler will substitute a no-argument constructor from the base class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class. This is shown in the COUNTEN2 program.

```
// counten2.cpp
// constructors in derived class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   protected:                          //NOTE: not private
      unsigned int count;              //count
   public:
      Counter() : count()              //constructor, no args
         {  }
      Counter(int c) : count(c)        //constructor, one arg
         {  }
      unsigned int get_count() const   //return count
         { return count; }
      Counter operator ++ ()           //incr count (prefix)
         { return Counter(++count); }
   };
////////////////////////////////////////////////////////////////
class CountDn : public Counter
   {
   public:
      CountDn() : Counter()            //constructor, no args
         { }
      CountDn(int c) : Counter(c)      //constructor, 1 arg
         { }
      CountDn operator -- ()           //decr count (prefix)
         { return CountDn(--count); }
   };
////////////////////////////////////////////////////////////////
int main()
   {
```

```
   CountDn c1;                          //class CountDn
   CountDn c2(100);

   cout << "\nc1=" << c1.get_count();    //display
   cout << "\nc2=" << c2.get_count();    //display

   ++c1; ++c1; ++c1;                    //increment c1
   cout << "\nc1=" << c1.get_count();    //display it

   --c2; --c2;                          //decrement c2
   cout << "\nc2=" << c2.get_count();    //display it

   CountDn c3 = --c2;                   //create c3 from c2
   cout << "\nc3=" << c3.get_count();    //display c3
   cout << endl;
   return 0;
   }
```

This program uses two new constructors in the `CountDn` class. Here is the no-argument constructor:

```
CountDn() : Counter()
   { }
```

This constructor has an unfamiliar feature: the function name following the colon. This construction causes the `CountDn()` constructor to call the `Counter()` constructor in the base class. In `main()`, when we say

```
CountDn c1;
```

the compiler will create an object of type `CountDn` and then call the `CountDn` constructor to initialize it. This constructor will in turn call the `Counter` constructor, which carries out the work. The `CountDn()` constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty.

Calling a constructor from the initialization list may seem odd, but it makes sense. You want to initialize any variables, whether they're in the derived class or the base class, before any statements in either the derived or base-class constructors are executed. By calling the base-class constructor before the derived-class constructor starts to execute, we accomplish this.

The statement

```
CountDn c2(100);
```

in `main()` uses the one-argument constructor in `CountDn`. This constructor also calls the corresponding one-argument constructor in the base class:

```
CountDn(int c) : Counter(c)     <------- argument c is passed to Counter
   { }
```

This construction causes the argument `c` to be passed from `CountDn()` to `Counter()`, where it is used to initialize the object.

In `main()`, after initializing the `c1` and `c2` objects, we increment one and decrement the other and then print the results. The one-argument constructor is also used in an assignment statement.

```
CountDn c3 = --c2;
```

Overriding Member Functions

You can use member functions in a derived class that override—that is, have the same name as—those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes.

Here's an example based on the STAKARAY program from Chapter 7, "Arrays and Strings." That program modeled a stack, a simple data storage device. It allowed you to push integers onto the stack and pop them off. However, STAKARAY had a potential flaw. If you tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the `st[]` array. Or if you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array.

To cure these defects we've created a new class, `Stack2`, derived from `Stack`. Objects of `Stack2` behave in exactly the same way as those of `Stack`, except that you will be warned if you attempt to push too many items on the stack or if you try to pop an item from an empty stack. Here's the listing for STAKEN:

```
// staken.cpp
// overloading functions in base and derived classes
#include <iostream>
using namespace std;
#include <process.h>            //for exit()
////////////////////////////////////////////////////////////////
class Stack
   {
   protected:                   //NOTE: can't be private
      enum { MAX = 3 };         //size of stack array
      int st[MAX];              //stack: array of integers
      int top;                  //index to top of stack
   public:
      Stack()                   //constructor
         { top = -1; }
      void push(int var)        //put number on stack
         { st[++top] = var; }
      int pop()                 //take number off stack
         { return st[top--]; }
```

```
   };
//////////////////////////////////////////////////////////////
class Stack2 : public Stack
   {
   public:
      void push(int var)          //put number on stack
         {
         if(top >= MAX-1)         //error if stack full
            { cout << "\nError: stack is full"; exit(1); }
         Stack::push(var);        //call push() in Stack class
         }
      int pop()                   //take number off stack
         {
         if(top < 0)              //error if stack empty
            { cout << "\nError: stack is empty\n"; exit(1); }
         return Stack::pop();     //call pop() in Stack class
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   Stack2 s1;

   s1.push(11);                   //push some values onto stack
   s1.push(22);
   s1.push(33);

   cout << endl << s1.pop();      //pop some values from stack
   cout << endl << s1.pop();
   cout << endl << s1.pop();
   cout << endl << s1.pop();      //oops, popped one too many...
   cout << endl;
   return 0;
   }
```

In this program the `Stack` class is just the same as it was in the STAKARAY program, except that the data members have been made `protected`.

Which Function Is Used?

The `Stack2` class contains two functions, `push()` and `pop()`. These functions have the same names, and the same argument and return types, as the functions in `Stack`. When we call these functions from `main()`, in statements like

```
s1.push(11);
```

how does the compiler know which of the two `push()` functions to use? Here's the rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed. (This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.) We say that the derived class function *overrides* the base class function. So in the preceding statement, since `s1` is an object of class `Stack2`, the `push()` function in `Stack2` will be executed, not the one in `Stack`.

The `push()` function in `Stack2` checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the `push()` function in `Stack`. Similarly, the `pop()` function in `Stack2` checks to see whether the stack is empty. If it is, it prints an error message and exits; otherwise, it calls the `pop()` function in `Stack`.

In `main()` we push three items onto the stack, but we pop four. The last pop elicits an error message

```
33
22
11
Error: stack is empty
```

and terminates the program.

Scope Resolution with Overridden Functions

How do `push()` and `pop()` in `Stack2` access `push()` and `pop()` in `Stack`? They use the scope resolution operator, `::`, in the statements

```
Stack::push(var);
```

and

```
return Stack::pop();
```

These statements specify that the `push()` and `pop()` functions in `Stack` are to be called. Without the scope resolution operator, the compiler would think the `push()` and `pop()` functions in `Stack2` were calling themselves, which—in this case—would lead to program failure. Using the scope resolution operator allows you to specify exactly what class the function is a member of.

Inheritance in the English `Distance` Class

Here's a somewhat more complex example of inheritance. So far in this book the various programs that used the English `Distance` class assumed that the distances to be represented would always be positive. This is usually the case in architectural drawings. However, if we were

measuring, say, the water level of the Pacific Ocean as the tides varied, we might want to be able to represent negative feet-and-inches quantities. (Tide levels below mean-lower-low-water are called *minus tides*; they prompt clam diggers to take advantage of the larger area of exposed beach.)

Let's derive a new class from `Distance`. This class will add a single data item to our feet-and-inches measurements: a sign, which can be positive or negative. When we add the sign, we'll also need to modify the member functions so they can work with signed distances. Here's the listing for ENGLEN:

```
// englen.cpp
// inheritance using English Distances
#include <iostream>
using namespace std;
enum posneg { pos, neg };          //for sign in DistSign
////////////////////////////////////////////////////////////////
class Distance                     //English Distance class
   {
   protected:                      //NOTE: can't be private
      int feet;
      float inches;
   public:                         //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                      //2-arg constructor)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()               //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const        //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class DistSign : public Distance  //adds sign to Distance
   {
   private:
      posneg sign;                 //sign is pos or neg
   public:
                                   //no-arg constructor
      DistSign() : Distance()      //call base constructor
         { sign = pos; }           //set the sign to +
```

```
                                  //2- or 3-arg constructor
      DistSign(int ft, float in, posneg sg=pos) :
              Distance(ft, in)    //call base constructor
         { sign = sg; }           //set the sign

      void getdist()              //get length from user
         {
         Distance::getdist();     //call base getdist()
         char ch;                 //get sign from user
         cout << "Enter sign (+ or -): ";  cin >> ch;
         sign = (ch=='+') ? pos : neg;
         }
      void showdist() const       //display distance
         {
         cout << ( (sign==pos) ? "(+)" : "(-)" );  //show sign
         Distance::showdist();                     //ft and in
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   DistSign alpha;                  //no-arg constructor
   alpha.getdist();                 //get alpha from user

   DistSign beta(11, 6.25);         //2-arg constructor

   DistSign gamma(100, 5.5, neg);   //3-arg constructor

                                    //display all distances
   cout << "\nalpha = ";  alpha.showdist();
   cout << "\nbeta = ";  beta.showdist();
   cout << "\ngamma = ";  gamma.showdist();
   cout << endl;
   return 0;
   }
```

Here the `DistSign` class adds the functionality to deal with signed numbers. The `Distance` class in this program is just the same as in previous programs, except that the data is protected. Actually in this case it could be private, because none of the derived-class functions accesses it. However, it's safer to make it protected so that a derived-class function could access it if necessary.

Operation of ENGLEN

The `main()` program declares three different signed distances. It gets a value for `alpha` from the user and initializes `beta` to (+)11'–6.25" and `gamma` to (–)100'–5.5". In the output we use parentheses around the sign to avoid confusion with the hyphen separating feet and inches. Here's some sample output:

```
Enter feet: 6
Enter inches: 2.5
Enter sign (+ or -): -

alpha = (-)6'-2.5"
beta = (+)11'-6.25"
gamma = (-)100'-5.5"
```

The `DistSign` class is derived from `Distance`. It adds a single variable, `sign`, which is of type `posneg`. The `sign` variable will hold the sign of the distance. The `posneg` type is defined in an `enum` statement to have two possible values: `pos` and `neg`.

Constructors in `DistSign`

`DistSign` has two constructors, mirroring those in `Distance`. The first takes no arguments, the second takes either two or three arguments. The third, optional, argument in the second constructor is a sign, either `pos` or `neg`. Its default value is `pos`. These constructors allow us to define variables (objects) of type `DistSign` in several ways.

Both constructors in `DistSign` call the corresponding constructors in `Distance` to set the feet-and-inches values. They then set the `sign` variable. The no-argument constructor always sets it to `pos`. The second constructor sets it to `pos` if no third-argument value has been provided, or to a value (`pos` or `neg`) if the argument is specified.

The arguments `ft` and `in`, passed from `main()` to the second constructor in `DistSign`, are simply forwarded to the constructor in `Distance`.

Member Functions in `DistSign`

Adding a sign to `Distance` has consequences for both of its member functions. The `getdist()` function in the `DistSign` class must ask the user for the sign as well as for feet-and-inches values, and the `showdist()` function must display the sign along with the feet and inches. These functions call the corresponding functions in `Distance`, in the lines

```
Distance::getdist();
```

and

```
Distance::showdist();
```

These calls get and display the feet and inches values. The body of `getdist()` and `showdist()` in `DistSign` then go on to deal with the sign.

Abetting Inheritance

C++ is designed to make it efficient to create a derived class. Where we want to use parts of the base class, it's easy to do so, whether these parts are data, constructors, or member functions. Then we add the functionality we need to create the new improved class. Notice that in ENGLEN we didn't need to duplicate any code; instead we made use of the appropriate functions in the base class.

Class Hierarchies

In the examples so far in this chapter, inheritance has been used to add functionality to an existing class. Now let's look at an example where inheritance is used for a different purpose: as part of the original design of a program.

Our example models a database of employees of a widget company. We've simplified the situation so that only three kinds of employees are represented. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses.

The database stores a name and an employee identification number for all employees, no matter what their category. However, for managers, it also stores their titles and golf club dues. For scientists, it stores the number of scholarly articles they have published. Laborers need no additional data beyond their names and numbers.

Our example program starts with a base class `employee`. This class handles the employee's last name and employee number. From this class three other classes are derived: `manager`, `scientist`, and `laborer`. The `manager` and `scientist` classes contain additional information about these categories of employee, and member functions to handle this information, as shown in Figure 9.5.

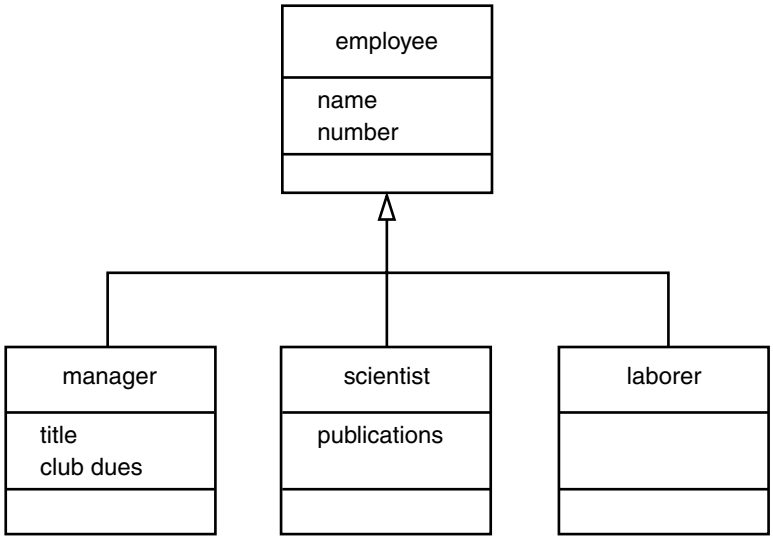

FIGURE 9.5
UML class diagram for EMPLOY.

Here's the listing for EMPLOY:

```
// employ.cpp
// models employee database using inheritance
#include <iostream>
using namespace std;
const int LEN = 80;                 //maximum length of names
////////////////////////////////////////////////////////////////
class employee                      //employee class
   {
   private:
      char name[LEN];               //employee name
      unsigned long number;         //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
```

```
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager : public employee    //management class
   {
   private:
      char title[LEN];             //"vice-president" etc.
      double dues;                 //golf club dues
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         }
   };
////////////////////////////////////////////////////////////////
class scientist : public employee  //scientist class
   {
   private:
      int pubs;                    //number of publications
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Number of publications: " << pubs;
         }
   };
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
   {
```

```
   };
//////////////////////////////////////////////////////////////
int main()
   {
   manager m1, m2;
   scientist s1;
   laborer l1;

   cout << endl;           //get data for several employees
   cout << "\nEnter data for manager 1";
   m1.getdata();

   cout << "\nEnter data for manager 2";
   m2.getdata();

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();
                           //display data for several employees
   cout << "\nData on manager 1";
   m1.putdata();

   cout << "\nData on manager 2";
   m2.putdata();

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

The `main()` part of the program declares four objects of different classes: two managers, a scientist, and a laborer. (Of course many more employees of each type could be defined, but the output would become rather large.) It then calls the `getdata()` member functions to obtain information about each employee, and the `putdata()` function to display this information. Here's a sample interaction with EMPLOY. First the user supplies the data.

```
Enter data for manager 1
   Enter last name: Wainsworth
   Enter number: 10
   Enter title: President
```

```
   Enter golf club dues: 1000000
Enter data on manager 2
   Enter last name: Bradley
   Enter number: 124
   Enter title: Vice-President
   Enter golf club dues: 500000
Enter data for scientist 1
   Enter last name: Hauptman-Frenglish
   Enter number: 234234
   Enter number of pubs: 999
Enter data for laborer 1
   Enter last name: Jones
   Enter number: 6546544
```

The program then plays it back.

```
Data on manager 1
   Name: Wainsworth
   Number: 10
   Title: President
   Golf club dues: 1000000
Data on manager 2
   Name: Bradley
   Number: 124
   Title: Vice-President
   Golf club dues: 500000
Data on scientist 1
    Name: Hauptman-Frenglish
   Number: 234234
   Number of publications: 999
Data on laborer 1
   Name: Jones
   Number: 6546544
```

A more sophisticated program would use an array or some other container to arrange the data so that a large number of employee objects could be accommodated.

"Abstract" Base Class

Notice that we don't define any objects of the base class `employee`. We use this as a general class whose sole purpose is to act as a base from which other classes are derived.

The `laborer` class operates identically to the `employee` class, since it contains no additional data or functions. It may seem that the `laborer` class is unnecessary, but by making it a separate class we emphasize that all classes are descended from the same source, `employee`. Also, if in the future we decided to modify the `laborer` class, we would not need to change the declaration for `employee`.

Classes used only for deriving other classes, as `employee` is in EMPLOY, are sometimes loosely called *abstract classes*, meaning that no actual instances (objects) of this class are created. However, the term *abstract* has a more precise definition that we'll look at in Chapter 11, "Virtual Functions."

Constructors and Member Functions

There are no constructors in either the base or derived classes, so the compiler creates objects of the various classes automatically when it encounters definitions like

```
manager m1, m2;
```

using the default constructor for `manager` calling the default constructor for `employee`.

The `getdata()` and `putdata()` functions in `employee` accept a name and number from the user and display a name and number. Functions also called `getdata()` and `putdata()` in the `manager` and `scientist` classes use the functions in `employee`, and also do their own work. In `manager`, the `getdata()` function asks the user for a title and the amount of golf club dues, and `putdata()` displays these values. In `scientist`, these functions handle the number of publications.

Inheritance and Graphics Shapes

In the CIRCLES program in Chapter 6, "Objects and Classes," we saw a program in which a class represented graphics circles that could be displayed on the screen. Of course, there are other kinds of shapes besides circles, such as squares and triangles. The very phrase "kinds of shapes" implies an inheritance relationship between something called a "shape" and specific kinds of shapes like circles and squares. We can use this relationship to make a program that is more robust and easier to understand than a program that treats different shapes as being unrelated.

In particular we'll make a `shape` class that's a base class (parent) of three derived classes: a `circle` class, a `rect` (for rectangle) class, and a `tria` (for triangle) class. As with other programs that use the Console Graphics Lite functions, you may need to read Appendix E, "Console Graphics Lite," and either Appendix C, "Microsoft Visual C++," or Appendix D, "Borland C++Builder" for your specific compiler to learn how to build the graphics files into your program. Here's the listing for MULTSHAP:

```
// multshap.cpp
// balls, rects, and polygons
#include "msoftcon.h"          //for graphics functions
////////////////////////////////////////////////////////////
class shape                    //base class
```

```
   {
   protected:
      int xCo, yCo;           //coordinates of shape
      color fillcolor;        //color
      fstyle fillstyle;       //fill pattern
   public:                    //no-arg constructor
      shape() : xCo(0), yCo(0), fillcolor(cWHITE),
                                          fillstyle(SOLID_FILL)
         {  }                 //4-arg constructor
      shape(int x, int y, color fc, fstyle fs) :
                  xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
         {  }
      void draw() const       //set color and fill style
         {
         set_color(fillcolor);
         set_fill_style(fillstyle);
         }
   };
////////////////////////////////////////////////////////////////
class circle : public shape
   {
   private:
      int radius;             //(xCo, yCo) is center
   public:
      circle() : shape()      //no-arg constr
         {  }
                              //5-arg constructor
      circle(int x, int y, int r, color fc, fstyle fs)
                  : shape(x, y, fc, fs), radius(r)
         {  }
      void draw() const       //draw the circle
         {
         shape::draw();
         draw_circle(xCo, yCo, radius);
         }
   };
////////////////////////////////////////////////////////////////
class rect : public shape
   {
   private:
      int width, height;      //(xCo, yCo) is upper-left corner
   public:
      rect() : shape(), height(0), width(0)     //no-arg ctor
         {  }                                   //6-arg ctor
      rect(int x, int y, int h, int w, color fc, fstyle fs) :
                        shape(x, y, fc, fs), height(h), width(w)
```

```
         {  }
      void draw() const       //draw the rectangle
         {
         shape::draw();
         draw_rectangle(xCo, yCo, xCo+width, yCo+height);
         set_color(cWHITE);   //draw diagonal
         draw_line(xCo, yCo, xCo+width, yCo+height);
         }
   };
////////////////////////////////////////////////////////////////
class tria : public shape
   {
   private:
      int height;             //(xCo, yCo) is tip of pyramid
   public:
      tria() : shape(), height(0) //no-arg constructor
         { }                  //5-arg constructor
      tria(int x, int y, int h, color fc, fstyle fs) :
                                shape(x, y, fc, fs), height(h)
         {  }
      void draw() const       //draw the triangle
         {
         shape::draw();
         draw_pyramid(xCo, yCo, height);
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   init_graphics();           //initialize graphics system

   circle cir(40, 12, 5, cBLUE, X_FILL);      //create circle
   rect rec(12, 7, 10, 15, cRED, SOLID_FILL); //create rectangle
   tria tri(60, 7,  11, cGREEN, MEDIUM_FILL); //create triangle

   cir.draw();                //draw all shapes
   rec.draw();
   tri.draw();
   set_cursor_pos(1, 25);     //lower-left corner
   return 0;
   }
```

When executed, this program produces three different shapes: a blue circle, a red rectangle, and a green triangle. Figure 9.6 shows the output of MULTSHAP.

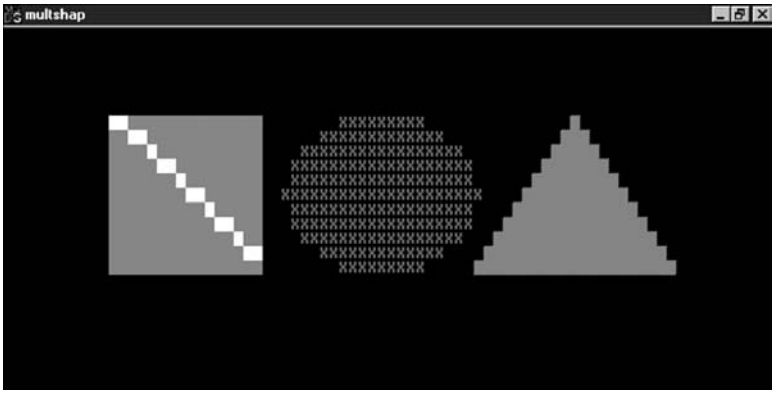

FIGURE 9.6
Output of the MULTSHAP program.

The characteristics that are common to all shapes, such as their location, color, and fill pattern, are placed in the `shape` class. Individual shapes have more specific attributes. A circle has a radius, for example, while a rectangle has a height and width. A `draw()` routine in `shape` handles the tasks specific to all shapes: setting their color and fill pattern. Overloaded `draw()` functions in the `circle`, `rect`, and `tria` classes take care of drawing their specific shapes once the color and pattern are determined.

As in the last example, the base class `shape` is an example of an abstract class, in that there is no meaning to instantiating an object of this class. What shape does a `shape` object display? The question doesn't make sense. Only a specific shape can display itself. The `shape` class exists only as a repository of attributes and actions that are common to all shapes.

Public and Private Inheritance

C++ provides a wealth of ways to fine-tune access to class members. One such access-control mechanism is the way derived classes are declared. Our examples so far have used publicly derived classes, with declarations like

```
class manager : public employee
```

which appeared in the EMPLOY example.

What is the effect of the `public` keyword in this statement, and what are the alternatives? Listen up: The keyword `public` specifies that objects of the derived class are able to access public member functions of the base class. The alternative is the keyword `private`. When this keyword is used, objects of the derived class cannot access public member functions of the base class. Since objects can never access `private` or `protected` members of a class, the result is that no member of the base class is accessible to objects of the derived class.

Access Combinations

There are so many possibilities for access that it's instructive to look at an example program that shows what works and what doesn't. Here's the listing for PUBPRIV:

```
// pubpriv.cpp
// tests publicly- and privately-derived classes
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A                  //base class
   {
   private:
      int privdataA;     //(functions have the same access
   protected:            //rules as the data shown here)
      int protdataA;
   public:
      int pubdataA;
   };
////////////////////////////////////////////////////////////////
class B : public A       //publicly-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         a = protdataA;  //OK
         a = pubdataA;   //OK
         }
   };
////////////////////////////////////////////////////////////////
class C : private A      //privately-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         a = protdataA;  //OK
         a = pubdataA;   //OK
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   int a;
```

```
B objB;
a = objB.privdataA;   //error: not accessible
a = objB.protdataA;   //error: not accessible
a = objB.pubdataA;    //OK (A public to B)

C objC;
a = objC.privdataA;   //error: not accessible
a = objC.protdataA;   //error: not accessible
a = objC.pubdataA;    //error: not accessible (A private to C)
return 0;
}
```

The program specifies a base class, `A`, with private, protected, and public data items. Two classes, `B` and `C`, are derived from `A`. `B` is publicly derived and `C` is privately derived.

As we've seen before, functions in the derived classes can access protected and public data in the base class. Objects of the derived classes cannot access private or protected members of the base class.

What's new is the difference between publicly derived and privately derived classes. Objects of the publicly derived class `B` can access public members of the base class `A`, while objects of the privately derived class `C` cannot; they can only access the public members of their own derived class. This is shown in Figure 9.7.

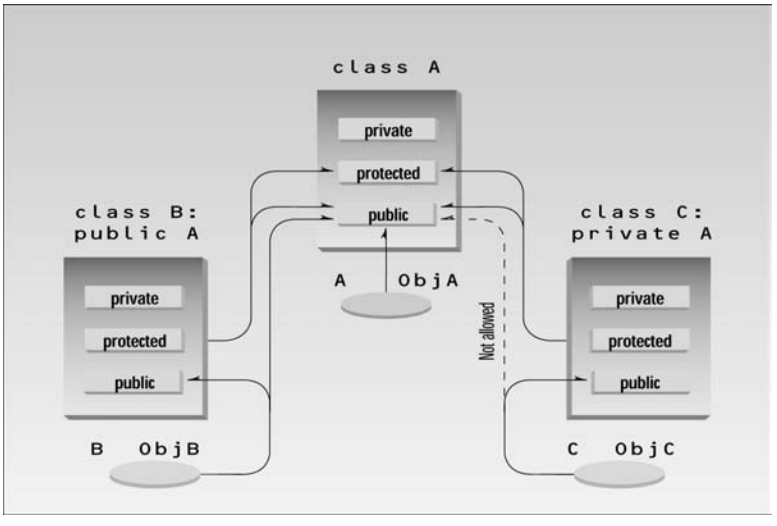

FIGURE 9.7
Public and private derivation.

If you don't supply any access specifier when creating a class, `private` is assumed.

Access Specifiers: When to Use What

How do you decide when to use private as opposed to public inheritance? In most cases a derived class exists to offer an improved—or a more specialized—version of the base class. We've seen examples of such derived classes (for instance, the `CountDn` class that adds the decrement operator to the `Counter` class and the `manager` class that is a more specialized version of the `employee` class). In such cases it makes sense for objects of the derived class to access the public functions of the base class if they want to perform a basic operation, and to access functions in the derived class to perform the more specialized operations that the derived class provides. In such cases public derivation is appropriate.

In some situations, however, the derived class is created as a way of completely modifying the operation of the base class, hiding or disguising its original interface. For example, imagine that you have already created a really nice `Array` class that acts like an array but provides protection against out-of-bounds array indexes. Then suppose you want to use this `Array` class as the basis for a `Stack` class, instead of using a basic array. You might derive `Stack` from `Array`, but you wouldn't want the users of `Stack` objects to treat them as if they were arrays, using the `[]` operator to access data items, for example. Objects of `Stack` should always be treated as if they were stacks, using `push()` and `pop()`. That is, you want to disguise the `Array` class as a `Stack` class. In this situation, private derivation would allow you to conceal all the `Array` class functions from objects of the derived `Stack` class.

Levels of Inheritance

Classes can be derived from classes that are themselves derived. Here's a miniprogram that shows the idea:

```
class A
   { };
class B : public A
   { };
class C : public B
   { };
```

Here `B` is derived from `A`, and `C` is derived from `B`. The process can be extended to an arbitrary number of levels—`D` could be derived from `C`, and so on.

As a more concrete example, suppose that we decided to add a special kind of laborer called a *foreman* to the EMPLOY program. We'll create a new program, EMPLOY2, that incorporates objects of class `foreman`.

Since a foreman is a kind of laborer, the `foreman` class is derived from the `laborer` class, as shown in Figure 9.8.

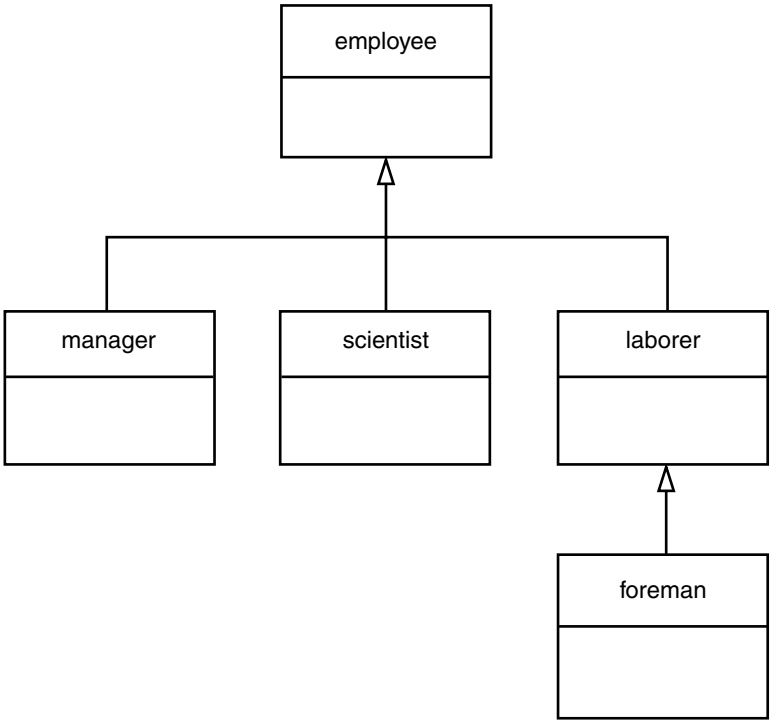

FIGURE 9.8
UML class diagram for EMPLOY2.

Foremen oversee the widget-stamping operation, supervising groups of laborers. They are responsible for the widget production quota for their group. A foreman's ability is measured by the percentage of production quotas successfully met. The `quotas` data item in the `foreman` class represents this percentage. Here's the listing for EMPLOY2:

```
// employ2.cpp
// multiple levels of inheritance
#include <iostream>
using namespace std;
const int LEN = 80;                 //maximum length of names
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      char name[LEN];               //employee name
      unsigned long number;         //employee number
   public:
      void getdata()
```

```
        {
        cout << "\n   Enter last name: "; cin >> name;
        cout << "   Enter number: ";      cin >> number;
        }
     void putdata() const
        {
        cout << "\n   Name: " << name;
        cout << "\n   Number: " << number;
        }
  };
////////////////////////////////////////////////////////////////
class manager : public employee    //manager class
  {
  private:
     char title[LEN];              //"vice-president" etc.
     double dues;                  //golf club dues
  public:
     void getdata()
        {
        employee::getdata();
        cout << "   Enter title: ";          cin >> title;
        cout << "   Enter golf club dues: "; cin >> dues;
        }
     void putdata() const
        {
        employee::putdata();
        cout << "\n   Title: " << title;
        cout << "\n   Golf club dues: " << dues;
        }
  };
////////////////////////////////////////////////////////////////
class scientist : public employee  //scientist class
  {
  private:
     int pubs;                     //number of publications
  public:
     void getdata()
        {
        employee::getdata();
        cout << "   Enter number of pubs: "; cin >> pubs;
        }
     void putdata() const
        {
        employee::putdata();
        cout << "\n   Number of publications: " << pubs;
        }
```

```
   };
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
   {
   };
////////////////////////////////////////////////////////////////
class foreman : public laborer     //foreman class
   {
   private:
      float quotas;  //percent of quotas met successfully
   public:
      void getdata()
         {
         laborer::getdata();
         cout << "   Enter quotas: "; cin >> quotas;
         }
      void putdata() const
         {
         laborer::putdata();
         cout << "\n   Quotas: " << quotas;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   laborer l1;
   foreman f1;

   cout << endl;
   cout << "\nEnter data for laborer 1";
   l1.getdata();
   cout << "\nEnter data for foreman 1";
   f1.getdata();

   cout << endl;
   cout << "\nData on laborer 1";
   l1.putdata();
   cout << "\nData on foreman 1";
   f1.putdata();
   cout << endl;
   return 0;
   }
```

Notice that a class hierarchy is not the same as an organization chart. An organization chart shows lines of command. A class hierarchy results from generalizing common characteristics. The more general the class, the higher it is on the chart. Thus a laborer is more general than a foreman, who is a specialized kind of laborer, so `laborer` is shown above `foreman` in the class hierarchy, although a foreman is probably paid more than a laborer.

Multiple Inheritance

A class can be derived from more than one base class. This is called *multiple inheritance*. Figure 9.9 shows how this looks when a class `C` is derived from base classes `A` and `B`.

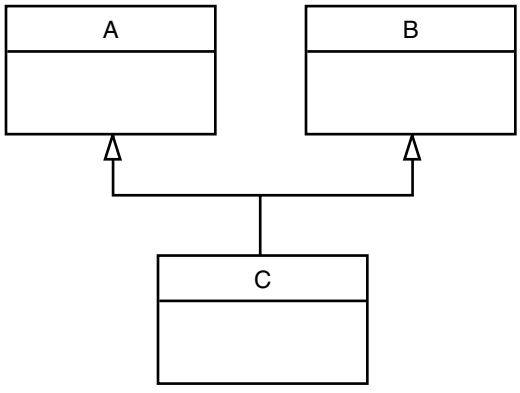

FIGURE 9.9
UML class diagram for multiple inheritance.

The syntax for multiple inheritance is similar to that for single inheritance. In the situation shown in Figure 9.9, the relationship is expressed like this:

```
class A                          // base class A
   {
   };
class B                          // base class B
   {
   };
class C : public A, public B     // C is derived from A and B
   {
   };
```

The base classes from which `C` is derived are listed following the colon in `C`'s specification; they are separated by commas.

Member Functions in Multiple Inheritance

As an example of multiple inheritance, suppose that we need to record the educational experience of some of the employees in the EMPLOY program. Let's also suppose that, perhaps in a different project, we've already developed a class called `student` that models students with different educational backgrounds. We decide that instead of modifying the `employee` class to incorporate educational data, we will add this data by multiple inheritance from the `student` class.

The `student` class stores the name of the school or university last attended and the highest degree received. Both these data items are stored as strings. Two member functions, `getedu()` and `putedu()`, ask the user for this information and display it.

Educational information is not relevant to every class of employee. Let's suppose, somewhat undemocratically, that we don't need to record the educational experience of laborers; it's only relevant for managers and scientists. We therefore modify `manager` and `scientist` so that they inherit from both the `employee` and `student` classes, as shown in Figure 9.10.

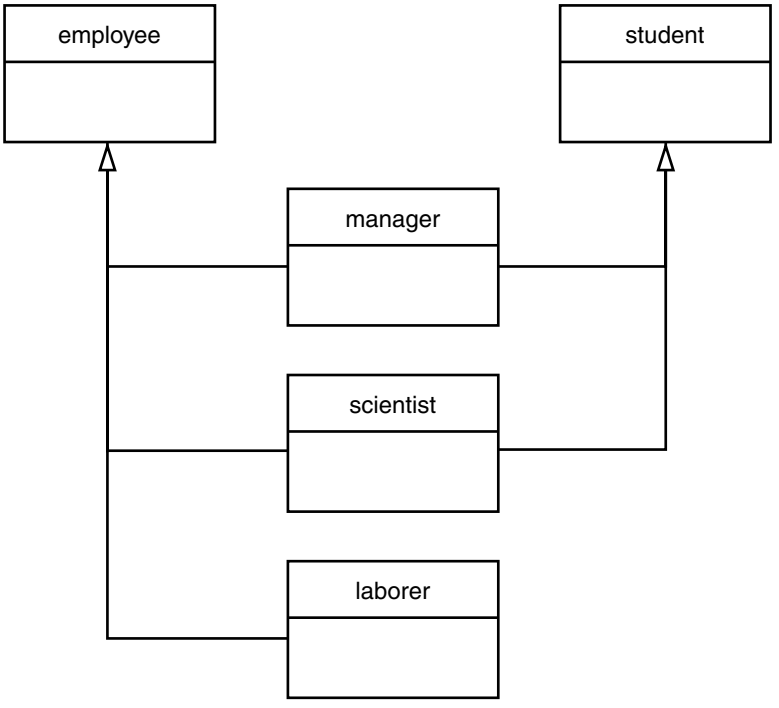

FIGURE 9.10
UML class diagram for EMPMULT.

Here's a miniprogram that shows these relationships (but leaves out everything else):

```
class student
   { };
class employee
   { };
class manager : private employee, private student
   { };
class scientist : private employee, private student
   { };
class laborer : public employee
   { };
```

And here, featuring considerably more detail, is the listing for EMPMULT:

```
//empmult.cpp
//multiple inheritance with employees and degrees
#include <iostream>
using namespace std;
const int LEN = 80;           //maximum length of names
////////////////////////////////////////////////////////////
class student                 //educational background
   {
   private:
      char school[LEN];       //name of school or university
      char degree[LEN];       //highest degree earned
   public:
      void getedu()
         {
         cout << "   Enter name of school or university: ";
         cin >> school;
         cout << "   Enter highest degree earned \n";
         cout << "   (Highschool, Bachelor's, Master's, PhD): ";
         cin >> degree;
         }
      void putedu() const
         {
         cout << "\n   School or university: " << school;
         cout << "\n   Highest degree earned: " << degree;
         }
   };
////////////////////////////////////////////////////////////
class employee
   {
   private:
      char name[LEN];         //employee name
      unsigned long number;   //employee number
```

```
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager : private employee, private student  //management
   {
   private:
      char title[LEN];        //"vice-president" etc.
      double dues;            //golf club dues
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         student::getedu();
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         student::putedu();
         }
   };
////////////////////////////////////////////////////////////////
class scientist : private employee, private student  //scientist
   {
   private:
      int pubs;     //number of publications
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         student::getedu();
         }
```

```
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Number of publications: " << pubs;
         student::putedu();
         }
   };
////////////////////////////////////////////////////////////////
class laborer : public employee             //laborer
   {
   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1;
   scientist s1, s2;
   laborer l1;

   cout << endl;
   cout << "\nEnter data for manager 1";    //get data for
   m1.getdata();                            //several employees

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for scientist 2";
   s2.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();

   cout << "\nData on manager 1";           //display data for
   m1.putdata();                            //several employees

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on scientist 2";
   s2.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

The `getdata()` and `putdata()` functions in the `manager` and `scientist` classes incorporate calls to functions in the `student` class, such as

```
student::getedu();
```

and

```
student::putedu();
```

These routines are accessible in `manager` and `scientist` because these classes are descended from `student`.

Here's some sample interaction with EMPMULT:

```
Enter data for manager 1
   Enter last name: Bradley
   Enter number: 12
   Enter title: Vice-President
   Enter golf club dues: 100000
   Enter name of school or university: Yale
   Enter highest degree earned
   (Highschool, Bachelor's, Master's, PhD): Bachelor's

Enter data for scientist 1
   Enter last name: Twilling
   Enter number: 764
   Enter number of pubs: 99
   Enter name of school or university: MIT
   Enter highest degree earned
   (Highschool, Bachelor's, Master's, PhD): PhD

Enter data for scientist 2
   Enter last name: Yang
   Enter number: 845
   Enter number of pubs: 101
   Enter name of school or university: Stanford
   Enter highest degree earned
   (Highschool, Bachelor's, Master's, PhD): Master's

Enter data for laborer 1
   Enter last name: Jones
   Enter number: 48323
```

As we saw in the EMPLOY and EMPLOY2 examples, the program then displays this information in roughly the same form.

`private` Derivation in EMPMULT

The `manager` and `scientist` classes in EMPMULT are privately derived from the `employee` and `student` classes. There is no need to use public derivation because objects of `manager` and `scientist` never call routines in the `employee` and `student` base classes. However, the `laborer` class must be publicly derived from `employer`, since it has no member functions of its own and relies on those in `employee`.

Constructors in Multiple Inheritance

EMPMULT has no constructors. Let's look at an example that does use constructors, and see how they're handled in multiple inheritance.

Imagine that we're writing a program for building contractors, and that this program models lumber-supply items. It uses a class that represents a quantity of lumber of a certain type: 100 8-foot-long construction grade 2×4s, for example.

The class should store various kinds of data about each such lumber item. We need to know the length (3'–6", for example) and we need to store the number of such pieces of lumber and their unit cost.

We also need to store a description of the lumber we're talking about. This has two parts. The first is the nominal dimensions of the cross-section of the lumber. This is given in inches. For instance, lumber 2 inches by 4 inches (for you metric folks, about 5 cm by 10 cm) is called a *two-by-four*. This is usually written *2×4*. We also need to know the grade of lumber—rough-cut, construction grade, surfaced-four-sides, and so on. We find it convenient to create a `Type` class to hold this data. This class incorporates member data for the nominal dimensions and the grade of the lumber, both expressed as strings, such as *2×6* and *construction*. Member functions get this information from the user and display it.

We'll use the `Distance` class from previous examples to store the length. Finally we create a `Lumber` class that inherits both the `Type` and `Distance` classes. Here's the listing for ENGLMULT:

```
// englmult.cpp
// multiple inheritance with English Distances
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Type                          //type of lumber
   {
   private:
      string dimensions;
      string grade;
```

```
   public:                          //no-arg constructor
      Type() : dimensions("N/A"), grade("N/A")
         {  }
                                    //2-arg constructor
      Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
      void gettype()                //get type from user
         {
         cout << "   Enter nominal dimensions (2x4 etc.): ";
         cin >> dimensions;
         cout << "   Enter grade (rough, const, etc.): ";
         cin >> grade;
         }
      void showtype() const         //display type
         {
         cout << "\n   Dimensions: " << dimensions;
         cout << "\n   Grade: " << grade;
         }
   };
////////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                          //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                       //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()                //get length from user
         {
         cout << "   Enter feet: ";  cin >> feet;
         cout << "   Enter inches: ";  cin >> inches;
         }
      void showdist() const         //display distance
         { cout  << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
   {
   private:
      int quantity;                       //number of pieces
      double price;                       //price of each piece
   public:                                //constructor (no args)
      Lumber() : Type(), Distance(), quantity(0), price(0.0)
```

```
      {  }
                                        //constructor (6 args)
   Lumber( string di, string gr,        //args for Type
           int ft, float in,            //args for Distance
           int qu, float prc ) :        //args for our data
           Type(di, gr),                //call Type ctor
           Distance(ft, in),            //call Distance ctor
           quantity(qu), price(prc)     //initialize our data
       {  }
   void getlumber()
      {
      Type::gettype();
      Distance::getdist();
      cout << "   Enter quantity: "; cin >> quantity;
      cout << "   Enter price per piece: "; cin >> price;
      }
   void showlumber() const
      {
      Type::showtype();
      cout << "\n   Length: ";
      Distance::showdist();
      cout << "\n   Price for " << quantity
          << " pieces: $" << price * quantity;
      }
};
////////////////////////////////////////////////////////////////
int main()
   {
   Lumber siding;                   //constructor (no args)

   cout << "\nSiding data:\n";
   siding.getlumber();              //get siding from user

                                    //constructor (6 args)
   Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

                                    //display lumber data
   cout << "\nSiding";  siding.showlumber();
   cout << "\nStuds";     studs.showlumber();
   cout << endl;
   return 0;
   }
```

The major new feature in this program is the use of constructors in the derived class `Lumber`. These constructors call the appropriate constructors in `Type` and `Distance`.

No-Argument Constructor

The no-argument constructor in `Type` looks like this:

```
Type()
   { strcpy(dimensions, "N/A"); strcpy(grade, "N/A"); }
```

This constructor fills in "N/A" (not available) for the `dimensions` and `grade` variables so the user will be made aware if an attempt is made to display data for an uninitialized lumber object.

You're already familiar with the no-argument constructor in the `Distance` class:

```
Distance() : feet(0), inches(0.0)
   {  }
```

The no-argument constructor in `Lumber` calls both of these constructors.

```
Lumber() : Type(), Distance(), quantity(0), price(0.0)
         {  }
```

The names of the base-class constructors follow the colon and are separated by commas. When the `Lumber()` constructor is invoked, these base-class constructors—`Type()` and `Distance()`—will be executed. The `quantity` and `price` attributes are also initialized.

Multi-Argument Constructors

Here is the two-argument constructor for `Type`:

```
Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
```

This constructor copies string arguments to the dimensions and grade member data items.

Here's the constructor for `Distance`, which is again familiar from previous programs:

```
Distance(int ft, float in) : feet(ft), inches(in)
   {  }
```

The constructor for `Lumber` calls both of these constructors, so it must supply values for their arguments. In addition it has two arguments of its own: the quantity of lumber and the unit price. Thus this constructor has six arguments. It makes two calls to the two constructors, each of which takes two arguments, and then initializes its own two data items. Here's what it looks like:

```
Lumber( string di, string gr,      //args for Type
        int ft, float in,          //args for Distance
        int qu, float prc ) :      //args for our data
        Type(di, gr),              //call Type ctor
        Distance(ft, in),          //call Distance ctor
        quantity(qu), price(prc)   //initialize our data
   {  }
```

Ambiguity in Multiple Inheritance

Odd sorts of problems may surface in certain situations involving multiple inheritance. Here's a common one. Two base classes have functions with the same name, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function? The name of the function alone is insufficient, since the compiler can't figure out which of the two functions is meant.

Here's an example, AMBIGU, that demonstrates the situation:

```
// ambigu.cpp
// demonstrates ambiguity in multiple inheritance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A
   {
   public:
      void show()  { cout << "Class A\n"; }
   };
class B
   {
   public:
      void show()  { cout << "Class B\n"; }
   };
class C : public A, public B
   {
   };
////////////////////////////////////////////////////////////////
int main()
   {
   C objC;            //object of class C
// objC.show();       //ambiguous--will not compile
   objC.A::show();    //OK
   objC.B::show();    //OK
   return 0;
   }
```

The problem is resolved using the scope-resolution operator to specify the class in which the function lies. Thus

```
objC.A::show();
```

refers to the version of `show()` that's in the `A` class, while

```
objC.B::show();
```

refers to the function in the B class. Stroustrup (see Appendix H, "Bibliography") calls this *disambiguation*.

Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree. The DIAMOND program shows how this looks.

```
//diamond.cpp
//investigates diamond-shaped multiple inheritance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A
   {
   public:
      void func();
   };
class B : public A
   {  };
class C : public A
   {  };
class D : public B, public C
   {  };
////////////////////////////////////////////////////////////////
int main()
   {
   D objD;
   objD.func();  //ambiguous: won't compile
   return 0;
   }
```

Classes B and C are both derived from class A, and class D is derived by multiple inheritance from both B and C. Trouble starts if you try to access a member function in class A from an object of class D. In this example objD tries to access func(). However, both B and C contain a copy of func(), inherited from A. The compiler can't decide which copy to use, and signals an error.

There are various advanced ways of coping with this problem, but the fact that such ambiguities can arise causes many experts to recommend avoiding multiple inheritance altogether. You should certainly not use it in serious programs unless you have considerable experience.

Aggregation: Classes Within Classes

We'll discuss aggregation here because, while it is not directly related to inheritance, both aggregation and inheritance are class relationships that are more specialized than associations. It is instructive to compare and contrast them.

If a class B is derived by inheritance from a class A, we can say that "B is a *kind of* A." This is because B has all the characteristics of A, and in addition some of its own. It's like saying that a starling is a kind of bird: A starling has the characteristics shared by all birds (wings, feathers, and so on) but has some distinctive characteristics of its own (such as dark iridescent plumage). For this reason inheritance is often called a "kind of" relationship.

Aggregation is called a "has a" relationship. We say a library has a book or an invoice has an item line. Aggregation is also called a "part-whole" relationship: the book is part of the library.

In object-oriented programming, aggregation may occur when one object is an attribute of another. Here's a case where an object of class A is an attribute of class B:

```
class A
   {
   };
class B
   {
   A objA;  // define objA as an object of class A
   };
```

In the UML, aggregation is considered a special kind of association. Sometimes it's hard to tell when an association is also an aggregation. It's always safe to call a relationship an association, but if class A contains objects of class B, and is organizationally superior to class B, it's a good candidate for aggregation. A company might have an aggregation of employees, or a stamp collection might have an aggregation of stamps.

Aggregation is shown in the same way as association in UML class diagrams, except that the "whole" end of the association line has an open diamond-shaped arrowhead. Figure 9.11 shows how this looks.

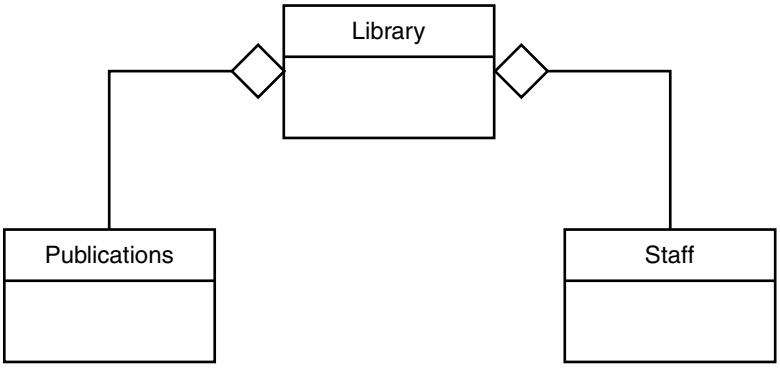

FIGURE 9.11
UML class diagram showing aggregation.

Aggregation in the EMPCONT Program

Let's rearrange the EMPMULT program to use aggregation instead of inheritance. In EMPMULT the `manager` and `scientist` classes are derived from the `employee` and `student` classes using the inheritance relationship. In our new program, EMPCONT, the `manager` and `scientist` classes contain instances of the `employee` and `student` classes as attributes. This aggregation relationship is shown in Figure 9.12.

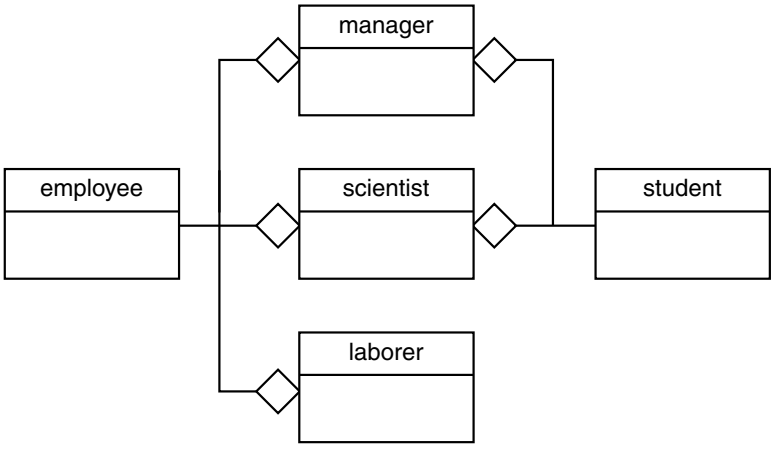

FIGURE 9.12
UML class diagram for EMPCONT.

The following miniprogram shows these relationships in a different way:

```
class student
   {};
class employee
   {};
class manager
   {
   student stu;    // stu is an object of class student
   employee emp;   // emp is an object of class employee
   };
class scientist
   {
   student stu;    // stu is an object of class student
   employee emp;   // emp is an object of class employee
   };
class laborer
   {
   employee emp;   // emp is an object of class employee
   };
```

Here's the full-scale listing for EMPCONT:

```
// empcont.cpp
// containership with employees and degrees
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class student                 //educational background
   {
   private:
      string school;          //name of school or university
      string degree;          //highest degree earned
   public:
      void getedu()
         {
         cout << "   Enter name of school or university: ";
         cin >> school;
         cout << "   Enter highest degree earned \n";
         cout << "   (Highschool, Bachelor's, Master's, PhD): ";
         cin >> degree;
         }
      void putedu() const
         {
         cout << "\n   School or university: " << school;
         cout << "\n   Highest degree earned: " << degree;
         }
   };
////////////////////////////////////////////////////////////
class employee
   {
   private:
      string name;            //employee name
      unsigned long number;   //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////
class manager                 //management
```

```
   {
   private:
      string title;           //"vice-president" etc.
      double dues;            //golf club dues
      employee emp;           //object of class employee
      student stu;            //object of class student
   public:
      void getdata()
         {
         emp.getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class scientist               //scientist
   {
   private:
      int pubs;               //number of publications
      employee emp;           //object of class employee
      student stu;            //object of class student
   public:
      void getdata()
         {
         emp.getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Number of publications: " << pubs;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class laborer                 //laborer
   {
```

```
   private:
      employee emp;           //object of class employee
   public:
      void getdata()
         { emp.getdata(); }
      void putdata() const
         { emp.putdata(); }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1;
   scientist s1, s2;
   laborer l1;

   cout << endl;
   cout << "\nEnter data for manager 1";    //get data for
   m1.getdata();                            //several employees

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for scientist 2";
   s2.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();

   cout << "\nData on manager 1";           //display data for
   m1.putdata();                            //several employees

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on scientist 2";
   s2.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

The `student` and `employee` classes are the same in EMPCONT as they were in EMPMULT, but they are related in a different way to the `manager` and `scientist` classes.

Composition: A Stronger Aggregation

Composition is a stronger form of aggregation. It has all the characteristics of aggregation, plus two more:

- The part may belong to only one whole.
- The lifetime of the part is the same as the lifetime of the whole.

A car is composed of doors (among other things). The doors can't belong to some other car, and they are born and die along with the car. A room is composed of a floor, ceiling, and walls. While aggregation is a "has a" relationship, composition is a "consists of" relationship.

In UML diagrams, composition is shown in the same way as aggregation, except that the diamond-shaped arrowhead is solid instead of open. This is shown in Figure 9.13.

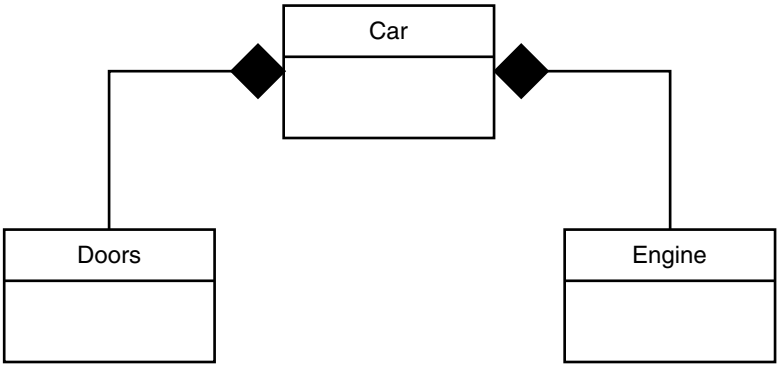

FIGURE 9.13
UML class diagram showing composition.

Even a single object can be related to a class by composition. In a car there is only one engine.

Inheritance and Program Development

The program-development process, as practiced for decades by programmers everywhere, is being fundamentally altered by object-oriented programming. This is due not only to the use of classes in OOP but to inheritance as well. Let's see how this comes about.

Programmer A creates a class. Perhaps it's something like the `Distance` class, with a complete set of member functions for arithmetic operations on a user-defined data type.

Programmer B likes the `Distance` class but thinks it could be improved by using signed distances. The solution is to create a new class, like `DistSign` in the ENGLEN example, that is derived from `Distance` but incorporates the extensions necessary to implement signed distances.

Programmers C and D then write applications that use the `DistSign` class.

Programmer B may not have access to the source code for the `Distance` member functions, and programmers C and D may not have access to the source code for `DistSign`. Yet, because of the software reusability feature of C++, B can modify and extend the work of A, and C and D can make use of the work of B (and A).

Notice that the distinction between software tool developers and application writers is becoming blurred. Programmer A creates a general-purpose programming tool, the `Distance` class. Programmer B creates a specialized version of this class, the `DistSign` class. Programmers C and D create applications. A is a tool developer, and C and D are applications developers. B is somewhere in between. In any case OOP is making the programming scene more flexible and at the same time more complex.

In Chapter 13 we'll see how a class can be divided into a client-accessible part and a part that is distributed only in object form, so it can be used by other programmers without the distribution of source code.

Summary

A class, called the *derived class*, can inherit the features of another class, called the *base class*. The derived class can add other features of its own, so it becomes a specialized version of the base class. Inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using hierarchical relationships.

Accessibility of base class members from derived classes and from objects of derived classes is an important issue. Data or functions in the base class that are prefaced by the keyword *protected* can be accessed from derived classes but not by any other objects, including objects of derived classes. Classes may be publicly or privately derived from base classes. Objects of a publicly derived class can access public members of the base class, while objects of a privately derived class cannot.

A class can be derived from more than one base class. This is called *multiple inheritance*. A class can also be contained within another class.

In the UML, inheritance is called generalization. This relationship is represented in class diagrams by an open triangle pointing to the base (parent) class.

Aggregation is a “has a” or “part-whole” relationship: one class contains objects of another class. Aggregation is represented in UML class diagrams by an open diamond pointing to the “whole” part of the part-whole pair. Composition is a strong form of aggregation. Its arrowhead is solid rather than open.

Inheritance permits the reusability of software: Derived classes can extend the capabilities of base classes with no need to modify—or even access the source code of—the base class. This leads to new flexibility in the software development process, and to a wider range of roles for software developers.

Questions

Answers to these questions can be found in Appendix G.

1. Inheritance is a way to
   a. make general classes into more specific classes.
   b. pass arguments to objects of classes.
   c. add features to existing classes without rewriting them.
   d. improve data hiding and encapsulation.
2. A “child” class is said to be _____ from a base class.
3. Advantages of inheritance include
   a. providing class growth through natural selection.
   b. facilitating class libraries.
   c. avoiding the rewriting of code.
   d. providing a useful conceptual framework.
4. Write the first line of the specifier for a class `Bosworth` that is publicly derived from a class `Alphonso`.
5. True or false: Adding a derived class to a base class requires fundamental changes to the base class.
6. To be accessed from a member function of the derived class, data or functions in the base class must be `public` or _____.
7. If a base class contains a member function `basefunc()`, and a derived class does not contain a function with this name, can an object of the derived class access `basefunc()`?
8. Assume that the classes mentioned in Question 4 and the class `Alphonso` contain a member function called `alfunc()`. Write a statement that allows object `BosworthObj` of class `Bosworth` to access `alfunc()`.

9. True or false: If no constructors are specified for a derived class, objects of the derived class will use the constructors in the base class.
10. If a base class and a derived class each include a member function with the same name, which member function will be called by an object of the derived class, assuming the scope-resolution operator is not used?
11. Write a declarator for a no-argument constructor of the derived class `Bosworth` of Question 4 that calls a no-argument constructor in the base class `Alphonso`.
12. The scope-resolution operator usually

    a. limits the visibility of variables to a certain function.

    b. tells what base class a class is derived from.

    c. specifies a particular class.

    d. resolves ambiguities.
13. True or false: It is sometimes useful to specify a class from which no objects will ever be created.
14. Assume that there is a class `Derv` that is derived from a base class `Base`. Write the declarator for a derived-class constructor that takes one argument and passes this argument along to the constructor in the base class.
15. Assume a class `Derv` that is privately derived from class `Base`. An object of class `Derv` located in `main()` can access

    a. public members of `Derv`.

    b. protected members of `Derv`.

    c. private members of `Derv`.

    d. public members of `Base`.

    e. protected members of `Base`.

    f. private members of `Base`.
16. True or false: A class `D` can be derived from a class `C`, which is derived from a class `B`, which is derived from a class `A`.
17. A class hierarchy

    a. shows the same relationships as an organization chart.

    b. describes "has a" relationships.

    c. describes "is a kind of" relationships.

    d. shows the same relationships as a family tree.
18. Write the first line of a specifier for a class `Tire` that is derived from class `Wheel` and from class `Rubber`.

19. Assume a class `Derv` derived from a base class `Base`. Both classes contain a member function `func()` that takes no arguments. Write a statement to go in a member function of `Derv` that calls `func()` in the base class.
20. True or false: It is illegal to make objects of one class members of another class.
21. In the UML, inheritance is called _____.
22. Aggregation is

    a. a stronger form of instantiation.

    b. a stronger form of generalization.

    c. a stronger form of composition.

    d. a "has a" relationship.
23. True or false: the arrow representing generalization points to the more specific class.
24. Composition is a _____ form of _____.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class `publication` that stores the title (a string) and price (type `float`) of a publication. From this class derive two classes: `book`, which adds a page count (type `int`), and `tape`, which adds a playing time in minutes (type `float`). Each of these three classes should have a `getdata()` function to get its data from the user at the keyboard, and a `putdata()` function to display its data.

    Write a `main()` program to test the `book` and `tape` classes by creating instances of them, asking the user to fill in data with `getdata()`, and then displaying the data with `putdata()`.

*2. Recall the STRCONV example from Chapter 8. The `String` class in this example has a flaw: It does not protect itself if its objects are initialized to have too many characters. (The `SZ` constant has the value 80.) For example, the definition

```
String s = "This string will surely exceed the width of the "
        "screen, which is what the SZ constant represents.";
```

    will cause the `str` array in `s` to overflow, with unpredictable consequences, such as crashing the system.

    With `String` as a base class, derive a class `Pstring` (for "protected string") that prevents buffer overflow when too long a string constant is used in a definition. A new constructor in the derived class should copy only `SZ-1` characters into `str` if the string constant is longer, but copy the entire constant if it's shorter. Write a `main()` program to test different lengths of strings.

*3. Start with the `publication`, `book`, and `tape` classes of Exercise 1. Add a base class `sales` that holds an array of three `floats` so that it can record the dollar sales of a particular publication for the last three months. Include a `getdata()` function to get three sales amounts from the user, and a `putdata()` function to display the sales figures. Alter the `book` and `tape` classes so they are derived from both `publication` and `sales`. An object of class `book` or `tape` should input and output sales data along with its other data. Write a `main()` function to create a `book` object and a `tape` object and exercise their input/output capabilities.

4. Assume that the publisher in Exercises 1 and 3 decides to add a third way to distribute books: on computer disk, for those who like to do their reading on their laptop. Add a `disk` class that, like `book` and `tape`, is derived from `publication`. The `disk` class should incorporate the same member functions as the other classes. The data item unique to this class is the disk type: either CD or DVD. You can use an `enum` type to store this item. The user could select the appropriate type by typing `c` or `d`.

5. Derive a class called `employee2` from the `employee` class in the EMPLOY program in this chapter. This new class should add a type `double` data item called `compensation`, and also an `enum` type called `period` to indicate whether the employee is paid hourly, weekly, or monthly. For simplicity you can change the `manager`, `scientist`, and `laborer` classes so they are derived from `employee2` instead of `employee`. However, note that in many circumstances it might be more in the spirit of OOP to create a separate base class called `compensation` and three new classes `manager2`, `scientist2`, and `laborer2`, and use multiple inheritance to derive these three classes from the original `manager`, `scientist`, and `laborer` classes and from `compensation`. This way none of the original classes needs to be modified.

6. Start with the ARROVER3 program in Chapter 8. Keep the `safearay` class the same as in that program, and, using inheritance, derive the capability for the user to specify both the upper and lower bounds of the array in a constructor. This is similar to Exercise 9 in Chapter 8, except that inheritance is used to derive a new class (you can call it `safehilo`) instead of modifying the original class.

7. Start with the COUNTEN2 program in this chapter. It can increment or decrement a counter, but only using prefix notation. Using inheritance, add the ability to use postfix notation for both incrementing and decrementing. (See Chapter 8 for a description of postfix notation.)

8. Operators in some computer languages, such as Visual Basic, allow you to select parts of an existing string and assign them to other strings. (The Standard C++ `string` class offers a different approach.) Using inheritance, add this capability to the `Pstring` class of Exercise 2. In the derived class, `Pstring2`, incorporate three new functions: `left()`, `mid()`, and `right()`.

```
s2.left(s1, n)    // s2 is assigned the leftmost n characters
                  //    from s1
s2.mid(s1, s, n)  // s2 is assigned the middle n characters
                  //    from s1, starting at character number s
                  //    (leftmost character is 0)
s2.right(s1, n)   // s2 is assigned the rightmost n characters
                  //    from s1
```

You can use `for` loops to copy the appropriate parts of `s1`, character by character, to a temporary `Pstring2` object, which is then returned. For extra credit, have these functions return by reference, so they can be used on the left side of the equal sign to change parts of an existing string.

9. Start with the `publication`, `book`, and `tape` classes of Exercise 1. Suppose you want to add the date of publication for both books and tapes. From the `publication` class, derive a new class called `publication2` that includes this member data. Then change `book` and `tape` so they are derived from `publication2` instead of `publication`. Make all the necessary changes in member functions so the user can input and output dates along with the other data. For the dates, you can use the `date` class from Exercise 5 in Chapter 6, which stores a date as three `int`s, for month, day, and year.
10. There is only one kind of manager in the EMPMULT program in this chapter. Any serious company has executives as well as managers. From the `manager` class derive a class called `executive`. (We'll assume an executive is a high-end kind of manager.) The additional data in the `executive` class will be the size of the employee's yearly bonus and the number of shares of company stock held in his or her stock-option plan. Add the appropriate member functions so these data items can be input and displayed along with the other `manager` data.
11. Various situations require that pairs of numbers be treated as a unit. For example, each screen coordinate has an x (horizontal) component and a y (vertical) component. Represent such a pair of numbers as a structure called `pair` that comprises two `int` member variables.

    Now, assume you want to be able to store `pair` variables on a stack. That is, you want to be able to place a pair (which contains two integers) onto a stack using a single call to a `push()` function with a structure of type `pair` as an argument, and retrieve a pair using a single call to a `pop()` function, which will return a structure of type `pair`. Start with the `Stack2` class in the STAKEN program in this chapter, and from it derive a new class called `pairStack`. This new class need contain only two members: the overloaded `push()` and `pop()` functions. The `pairStack::push()` function will need to make two calls to `Stack2::push()` to store the two integers in its pair, and the `pairStack::pop()` function will need to make two calls to `Stack2::pop()` (although not necessarily in the same order).

12. Amazing as it may seem, the old British pounds-shillings-pence money notation (£9.19.11—see Exercise 10 in Chapter 4, "Structures") isn't the whole story. A penny was further divided into halfpennies and farthings, with a farthing being worth 1/4 of a penny. There was a halfpenny coin, a farthing coin, and a halffarthing coin. Fortunately all this can be expressed numerically in eighths of a penny:

    1/8 penny is a halffarthing

    1/4 penny is a farthing

    3/8 penny is a farthing and a half

    1/2 penny is a halfpenny (pronounced ha'penny)

    5/8 penny is a halfpenny plus a halffarthing

    3/4 penny is a halfpenny plus a farthing

    7/8 penny is a halfpenny plus a farthing and a half

    Let's assume we want to add to the `sterling` class the ability to handle such fractional pennies. The I/O format can be something like `£1.1.1-1/4` or `£9.19.11-7/8`, where the hyphen separates the fraction from the pennies.

    Derive a new class called `sterfrac` from `sterling`. It should be able to perform the four arithmetic operations on sterling quantities that include eighths of a penny. Its only member data is an `int` indicating the number of eighths; you can call it `eighths`. You'll need to overload many of the functions in `sterling` to handle the eighths. The user should be able to type any fraction in lowest terms, and the display should also show fractions in lowest terms. It's not necessary to use the full-scale `fraction` class (see Exercise 11 in Chapter 6), but you could try that for extra credit.