Chapter 12. Object-Oriented Programming: Inheritance

Say not you know another entirely, till you have divided an inheritance with him.

Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.

Bertrand Russell

Good as it is to inherit a library, it is better to collect one.

Augustine Birrell

Save base authority from others' books.

William Shakespeare

OBJECTIVES

In this chapter you will learn:

1. To create classes by inheriting from existing classes.
2. How inheritance promotes software reuse.
3. The notions of base classes and derived classes and the relationships between them.
4. The \texttt{protected} member access specifier.
5. The use of constructors and destructors in inheritance hierarchies.
6. The differences between \texttt{public}, \texttt{protected} and \texttt{private} inheritance.
7. The use of inheritance to customize existing software.

Outline

12.1 Introduction

12.2 Base Classes and Derived Classes

12.3 \texttt{protected} Members
12.4 Relationship between Base Classes and Derived Classes

12.4.1 Creating and Using a CommissionEmployee Class

12.4.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance

12.4.3 Creating a CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy

12.4.4 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

12.4.5 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

12.5 Constructors and Destructors in Derived Classes

12.6 public, protected and private Inheritance

12.7 Software Engineering with Inheritance

12.8 Wrap-Up

Summary

Terminology

Self-Review Exercises

Answers to Self-Review Exercises

Exercises

12.1. Introduction

This chapter continues our discussion of object-oriented programming (OOP) by introducing another of its key features, inheritance. Inheritance is a form of software reuse in which the programmer creates a class that absorbs an existing class's data and behaviors and enhances them with new capabilities. Software reusability saves time during program development. It also encourages the reuse of proven, debugged, high-quality software, which increases the likelihood that a system will be implemented effectively.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class. (Other programming languages, such as Java, refer to the base class as the superclass and the derived class as the subclass.) A derived class represents a more specialized group of objects. Typically, a derived
class contains behaviors inherited from its base class plus additional behaviors. As we will see, a derived class can also customize behaviors inherited from the base class. A direct base class is the base class from which a derived class explicitly inherits. An indirect base class is inherited from two or more levels up in the class hierarchy. In the case of single inheritance, a class is derived from one base class. C++ also supports multiple inheritance, in which a derived class inherits from multiple (possibly unrelated) base classes. Single inheritance is straightforward. We show several examples that should enable the reader to become proficient quickly. Multiple inheritance can be complex and error prone. We cover multiple inheritance in Chapter 24, Other Topics.

C++ offers three kinds of inheritance: public, protected, and private. In this chapter, we concentrate on public inheritance and briefly explain the other two. In Chapter 21, Data Structures, we show how private inheritance can be used as an alternative to composition. The third form, protected inheritance, is rarely used. With public inheritance, every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes. For example, if we have vehicle as a base class and car as a derived class, then all cars are vehicles, but not all vehicles are cars. As we continue our study of object-oriented programming in Chapter 12 and Chapter 13, we take advantage of this relationship to perform some interesting manipulations.

Experience in building software systems indicates that significant amounts of code deal with closely related special cases. When programmers are preoccupied with special cases, the details can obscure the big picture. With object-oriented programming, programmers focus on the commonalities among objects in the system rather than on the special cases.

We distinguish between the is-a relationship and the has-a relationship. The is-a relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class. For example, a car is a vehicle, so any properties and behaviors of a vehicle are also properties of a car. By contrast, the has-a relationship represents composition. (Composition was discussed in Chapter 10.) In a has-a relationship, an object contains one or more objects of other classes as members. For example, a car includes many components: it has a steering wheel, has a brake pedal, has a transmission, and has many other components.

Derived-class member functions might require access to base-class data members and member functions. A derived class can access the non-private members of its base class. Base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class. A derived class can effect state changes in private base-class members, but only through non-private member functions provided in the base class and inherited into the derived class.

Software Engineering Observation 12.1

Member functions of a derived class cannot directly access private members of the base class.

Software Engineering Observation 12.2

If a derived class could access its base class's private members, classes that inherit from that derived class could access that data as well. This would
One problem with inheritance is that a derived class can inherit data members and member functions it does not need or should not have. It is the class designer's responsibility to ensure that the capabilities provided by a class are appropriate for future derived classes. Even when a base-class member function is appropriate for a derived class, the derived class often requires that member function to behave in a manner specific to the derived class. In such cases, the base-class member function can be redefined in the derived class with an appropriate implementation.

propagate access to what should be private data, and the benefits of information hiding would be lost.

12.2. Base Classes and Derived Classes

Often, an object of one class is an object of another class, as well. For example, in geometry, a rectangle is a quadrilateral (as are squares, parallelograms and trapezoids). Thus, in C++, class Rectangle can be said to inherit from class Quadrilateral. In this context, class Quadrilateral is a base class, and class Rectangle is a derived class. A rectangle is a specific type of quadrilateral, but it is incorrect to claim that a quadrilateral is a rectangle the quadrilateral could be a parallelogram or some other shape. Figure 12.1 lists several simple examples of base classes and derived classes.

Because every derived-class object is an object of its base class, and one base class can have many derived classes, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class Vehicle represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived class Car represents a smaller, more specific subset of all vehicles.

Inheritance relationships form treelike hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they are employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class supplying members to other classes, a derived class inheriting its members from other classes, or both.
Let us develop a simple inheritance hierarchy with five levels (represented by the UML class diagram in Fig. 12.2). A university community has thousands of members.

Figure 12.2. Inheritance hierarchy for university CommunityMembers.

These members consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. Some administrators, however, also teach classes. Note that we have used multiple inheritance to form class AdministratorTeacher. Also note that this inheritance hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors and seniors.

Each arrow in the hierarchy (Fig. 12.2) represents an is-a relationship. For example, as we follow the arrows in this class hierarchy, we can state "an Employee is a CommunityMember" and "a Teacher is a Faculty member." CommunityMember is the direct base class of Employee, Student and Alumnus. In addition, CommunityMember is an indirect base class of all the other classes in the diagram. Starting from the bottom of the diagram, the reader can follow the arrows and apply the is-a relationship to the topmost base class. For example, an AdministratorTeacher is an Administrator, is a Faculty member, is an Employee and is a CommunityMember.

Now consider the Shape inheritance hierarchy in Fig. 12.3. This hierarchy begins with base class Shape. Classes TwoDimensionalShape and ThreeDimensionalShape derive from base class Shape. Shapes are either TwoDimensionalShapes or ThreeDimensionalShapes. The third level of this hierarchy contains some more specific types of TwoDimensionalShapes and ThreeDimensionalShapes. As in Fig. 12.2, we can follow the arrows from the bottom of the diagram to the topmost base class in this class hierarchy to identify several is-a relationships. For instance, a triangle is a TwoDimensionalShape and is a Shape, while a Sphere is a ThreeDimensionalShape and is a Shape. Note that this hierarchy could contain many other classes, such as Rectangles, Ellipses and trapezoids, which are all TwoDimensionalShapes.
To specify that class TwoDimensionalShape (Fig. 12.3) is derived from (or inherits from) class Shape, class TwoDimensionalShape could be defined in C++ as follows:

```cpp
class TwoDimensionalShape : public Shape
```

This is an example of **public inheritance**, the most commonly used form. We also will discuss **private inheritance** and **protected inheritance** (Section 12.6). With all forms of inheritance, **private** members of a base class are not accessible directly from that class's derived classes, but these **private** base-class members are still inherited (i.e., they are still considered parts of the derived classes). With **public** inheritance, all other base-class members retain their original member access when they become members of the derived class (e.g., **public** members of the base class become **public** members of the derived class, and, as we will soon see, **protected** members of the base class become **protected** members of the derived class). Through these inherited base-class members, the derived class can manipulate **private** members of the base class (if these inherited members provide such functionality in the base class). Note that **friend** functions are not inherited.

Inheritance is not appropriate for every class relationship. In **Chapter 10**, we discussed the has-a relationship, in which classes have members that are objects of other classes. Such relationships create classes by composition of existing classes. For example, given the classes **Employee**, **BirthDate** and **TelephoneNumber**, it is improper to say that an Employee is a BirthDate or that an Employee is a TelephoneNumber. However, it is appropriate to say that an Employee has a BirthDate and that an Employee has a TelephoneNumber.

It is possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the members of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class (i.e., such objects have an is-a relationship with the base class). In **Chapter 13**, Object-Oriented Programming: Polymorphism, we consider many examples that take advantage of this relationship.

**12.3. protected Members**

**Chapter 3** introduced access specifiers **public** and **private**. A base class's **public** members are
accessible within the body of that base class and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that base class or one of its derived classes. A base class's private members are accessible only within the body of that base class and the friends of that base class. In this section, we introduce an additional access specifier: protected.

Using protected access offers an intermediate level of protection between public and private access. A base class's protected members can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes derived from that base class.

Derived-class member functions can refer to public and protected members of the base class simply by using the member names. When a derived-class member function redefines a base-class member function, the base-class member can be accessed from the derived class by preceding the base-class member name with the base-class name and the binary scope resolution operator (::). We discuss accessing redefined members of the base class in Section 12.4 and using protected data in Section 12.4.4.

12.4. Relationship between Base Classes and Derived Classes

In this section, we use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a base class and a derived class. Commission employees (who will be represented as objects of a base class) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a derived class) receive a base salary plus a percentage of their sales. We divide our discussion of the relationship between commission employees and base-salaried commission employees into a carefully paced series of five examples:

1. In the first example, we create class CommissionEmployee, which contains as private data members a first name, last name, social security number, commission rate (percentage) and gross (i.e., total) sales amount.

2. The second example defines class BasePlusCommissionEmployee, which contains as private data members a first name, last name, social security number, commission rate, gross sales amount and base salary. We create the latter class by writing every line of code the class requires we will soon see that it is much more efficient to create this class simply by inheriting from class CommissionEmployee.

3. The third example defines a new version of class BasePlusCommissionEmployee class that inherits directly from class CommissionEmployee (i.e., a BasePlusCommissionEmployee is a CommissionEmployee who also has a base salary) and attempts to access class CommissionEmployee's private members; this results in compilation errors, because the derived class does not have access to the base class's private data.

4. The fourth example shows that if CommissionEmployee's data is declared as protected, a new version of class BasePlusCommissionEmployee that inherits from class
CommissionEmployee can access that data directly. For this purpose, we define a new version of class CommissionEmployee with protected data. Both the inherited and noninherited BasePlusCommissionEmployee classes contain identical functionality, but we show how the version of BasePlusCommissionEmployee that inherits from class CommissionEmployee is easier to create and manage.

5. After we discuss the convenience of using protected data, we create the fifth example, which sets the CommissionEmployee data members back to private to enforce good software engineering. This example demonstrates that derived class BasePlusCommissionEmployee can use base class CommissionEmployee’s public member functions to manipulate CommissionEmployee’s private data.

12.4.1. Creating and Using a CommissionEmployee Class

Let us first examine CommissionEmployee’s class definition (Figs. 12.4, 12.5). The CommissionEmployee header file (Fig. 12.4) specifies class CommissionEmployee’s public services, which include a constructor (lines 1213) and member functions earnings (line 30) and print (line 31). Lines 1528 declare public get and set functions for manipulating the class’s data members (declared in lines 3337) firstName, lastName, socialSecurityNumber, grossSales and commissionRate. The CommissionEmployee header file specifies each of these data members as private, so objects of other classes cannot directly access this data. Declaring data members as private and providing non-private get and set functions to manipulate and validate the data members helps enforce good software engineering. Member functions setGrossSales (defined in lines 5760 of Fig. 12.5) and setCommissionRate (defined in lines 6972 of Fig. 12.5), for example, validate their arguments before assigning the values to data members grossSales and commissionRate, respectively.

Figure 12.4. CommissionEmployee class header file.

(This item is displayed on page 640 in the print version)

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5  
6  #include <string> // C++ standard string class
7  using std::string;
8  
9  class CommissionEmployee
10  {
11    public:
12        CommissionEmployee( const string &, const string &, const string &,
13            double = 0.0, double = 0.0 );
14    
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17    
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20    
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23    
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26    
27    void setCommissionRate( double ); // set commission rate (percentage)
```
Figure 12.5. Implementation file for CommissionEmployee class that represents an employee who is paid a percentage of gross sales.

(This item is displayed on pages 640 - 642 in the print version)
The CommissionEmployee constructor definition purposely does not use member-initializer syntax in the first several examples of this section, so that we can demonstrate how private and protected specifiers affect member access in derived classes. As shown in Fig. 12.5, lines 1315, we assign values to data members firstName, lastName and socialSecurityNumber in the constructor body. Later in this section, we will return to using member-initializer lists in the constructors.
Note that we do not validate the values of the constructor's arguments first, last and ssn before assigning them to the corresponding data members. We certainly could validate the first and last names perhaps by ensuring that they are of a reasonable length. Similarly, a social security number could be validated to ensure that it contains nine digits, with or without dashes (e.g., 123-45-6789 or 123456789).

Member function earnings (lines 8184) calculates a CommissionEmployee's earnings. Line 83 multiplies the commissionRate by the grossSales and returns the result. Member function print (lines 8793) displays the values of a CommissionEmployee object's data members.

Figure 12.6 tests class CommissionEmployee. Lines 1617 instantiate object employee of class CommissionEmployee and invoke CommissionEmployee's constructor to initialize the object with "Sue" as the first name, "Jones" as the last name, "222-22-2222" as the social security number, 10000 as the gross sales amount and .06 as the commission rate. Lines 2329 use employee's get functions to display the values of its data members. Lines 3132 invoke the object's member functions setGrossSales and setCommissionRate to change the values of data members grossSales and commissionRate, respectively. Line 36 then calls employee's print member function to output the updated CommissionEmployee information. Finally, line 39 displays the CommissionEmployee's earnings, calculated by the object's earnings member function using the updated values of data members grossSales and commissionRate.
12.4.2. Creating a BasePlusCommissionEmployee Class Without Using Inheritance

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new and independent) class `BasePlusCommissionEmployee` (Figs. 12.7 12.8), which contains a first name, last name, social security number, gross sales amount, commission rate and base salary.

Figure 12.7. BasePlusCommissionEmployee class header file.
(This item is displayed on pages 644 - 645 in the print version)

```
1  // Fig. 12.7: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class definition represents an employee
3  // that receives a base salary in addition to commission.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6  
7  #include <string> // C++ standard string class
8  using std::string;
```
Figure 12.8. BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission.

(This item is displayed on pages 646 - 648 in the print version)
setBaseSalary( salary );  // validate and store base salary  
}  // end BasePlusCommissionEmployee constructor

void BasePlusCommissionEmployee::setFirstName( const string &first )  
{ 
    firstName = first;  // should validate
}  // end function setFirstName

string BasePlusCommissionEmployee::getFirstName() const  
{ 
    return firstName;
}  // end function getFirstName

void BasePlusCommissionEmployee::setLastName( const string &last )  
{ 
    lastName = last;  // should validate
}  // end function setLastName

string BasePlusCommissionEmployee::getLastName() const  
{ 
    return lastName;
}  // end function getLastName

void BasePlusCommissionEmployee::setSocialSecurityNumber( const string &ssn )  
{ 
    socialSecurityNumber = ssn;  // should validate
}  // end function setSocialSecurityNumber

string BasePlusCommissionEmployee::getSocialSecurityNumber() const  
{ 
    return socialSecurityNumber;
}  // end function getSocialSecurityNumber

void BasePlusCommissionEmployee::setGrossSales( double sales )  
{ 
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
}  // end function setGrossSales

double BasePlusCommissionEmployee::getGrossSales() const  
{ 
    return grossSales;
}  // end function getGrossSales

void BasePlusCommissionEmployee::setCommissionRate( double rate )  
{ 
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
}  // end function setCommissionRate

double BasePlusCommissionEmployee::getCommissionRate() const  
{ 
    return commissionRate;
}  // end function getCommissionRate
Defining Class BasePlusCommissionEmployee

The BasePlusCommissionEmployee header file (Fig. 12.7) specifies class BasePlusCommissionEmployee's public services, which include the class's constructor (lines 13-14) and member functions earnings (line 34) and print (line 35). Lines 16-32 declare public get and set functions for the class's private data members (declared in lines 37-42) firstName, lastName, socialSecurityNumber, grossSales, commissionRate and baseSalary. These variables and member functions encapsulate all the necessary features of a base-salaried commission employee. Note the similarity between this class and class CommissionEmployee (Figs. 12.4, 12.5) in this example, we will not yet exploit that similarity.

```cpp
// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
    return baseSalary + ( commissionRate * grossSales );
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried commission employee: " << firstName << ' ' << lastName << "social security number: " << socialSecurityNumber
        << "gross sales: " << grossSales
        << "commission rate: " << commissionRate
        << "base salary: " << baseSalary;
} // end function print
```

Class BasePlusCommissionEmployee's earnings member function (defined in lines 96-99 of Fig. 12.8) computes the earnings of a base-salaried commission employee. Line 98 returns the result of adding the employee's base salary to the product of the commission rate and the employee's gross sales.

Testing Class BasePlusCommissionEmployee

Figure 12.9 tests class BasePlusCommissionEmployee. Lines 17-18 instantiate object employee of class BasePlusCommissionEmployee, passing "Bob", "Lewis", "333-33-3333", 5000, .04 and 300 to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 24-31 use BasePlusCommissionEmployee's get functions to retrieve the values of the object's data members for output. Line 33 invokes the object's print function.
setBaseSalary member function to change the base salary. Member function setBaseSalary (Fig. 12.8, lines 8487) ensures that data member baseSalary is not assigned a negative value, because an employee’s base salary cannot be negative. Line 37 of Fig. 12.9 invokes the object’s print member function to output the updated BasePlusCommissionEmployee’s information, and line 40 calls member function earnings to display the BasePlusCommissionEmployee’s earnings.

---

**Figure 12.9. BasePlusCommissionEmployee class test program.**

(This item is displayed on pages 648 - 649 in the print version)

```cpp
1 // Fig. 12.9: fig12_09.cpp
2 // Testing class BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instantiate BasePlusCommissionEmployee object
17     BasePlusCommissionEmployee
18         employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // set floating-point output formatting
21     cout << fixed << setprecision( 2 );
22
23     // get commission employee data
24     cout << "Employee information obtained by get functions: \n"
25     << "\nFirst name is " << employee.getFirstName()
26     << "\nLast name is " << employee.getLastName()
27     << "\nSocial security number is "
28     << employee.getSocialSecurityNumber()
29     << "\nGross sales is " << employee.getGrossSales()
30     << "\nCommission rate is " << employee.getCommissionRate()
31     << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary( 1000 ); // set base salary
34
35     cout << "\nUpdated employee information output by print function: \n"
36     << endl;
37     employee.print(); // display the new employee information
38
39     // display the employee's earnings
40     cout << "\nEmployee's earnings: $" << employee.earnings() << endl;
41     return 0;
42 } // end main
```
Exploring the Similarities Between Class `BasePlusCommissionEmployee` and Class `CommissionEmployee`

Note that much of the code for class `BasePlusCommissionEmployee` ([Figs. 12.7,12.8]) is similar, if not identical, to the code for class `CommissionEmployee` ([Figs. 12.4,12.5]). For example, in class `BasePlusCommissionEmployee`, private data members `firstName` and `lastName` and member functions `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. Classes `CommissionEmployee` and `BasePlusCommissionEmployee` also both contain private data members `socialSecurityNumber`, `commissionRate` and `grossSales`, as well as get and set functions to manipulate these members. In addition, the `BasePlusCommissionEmployee` constructor is almost identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are private data member `baseSalary` and member functions `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s `print` member function is nearly identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s `print` also outputs the value of data member `baseSalary`.

We literally copied code from class `CommissionEmployee` and pasted it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and member functions that manipulate the base salary. This "copy-and-paste" approach is often error prone and time consuming. Worse yet, it can spread many physical copies of the same code throughout a system, creating a code-maintenance nightmare. Is there a way to "absorb" the data members and member functions of a class in a way that makes them part of other classes without duplicating code? In the next several examples, we do exactly this, using inheritance.

Software Engineering Observation 12.3
Copying and pasting code from one class to another can spread errors across multiple source code files. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.

Software Engineering Observation 12.4

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, software developers need to make the changes only in the base class-derived classes then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

12.4.3. Creating a \texttt{CommissionEmployeeBasePlusCommissionEmployee} Inheritance Hierarchy

Now we create and test a new version of class \texttt{BasePlusCommissionEmployee} (Figs. 12.10,12.11) that derives from class \texttt{CommissionEmployee} (Figs. 12.4,12.5). In this example, a \texttt{BasePlusCommissionEmployee} object is a \texttt{CommissionEmployee} (because inheritance passes on the capabilities of class \texttt{CommissionEmployee}), but class \texttt{BasePlusCommissionEmployee} also has data member \texttt{baseSalary} (Fig. 12.10, line 24). The colon (:) in line 12 of the class definition indicates inheritance. Keyword \texttt{public} indicates the type of inheritance. As a derived class (formed with \texttt{public} inheritance), \texttt{BasePlusCommissionEmployee} inherits all the members of class \texttt{CommissionEmployee}, except for the constructors each class provides its own constructors that are specific to the class. [Note that destructors, too, are not inherited.] Thus, the \texttt{public} services of \texttt{BasePlusCommissionEmployee} include its constructor (lines 1516) and the \texttt{public} member functions inherited from class \texttt{CommissionEmployee} although we cannot see these inherited member functions in \texttt{BasePlusCommissionEmployee}'s source code, they are nevertheless a part of derived class \texttt{BasePlusCommissionEmployee}. The derived class's \texttt{public} services also include member functions \texttt{setBaseSalary}, \texttt{getBaseSalary}, \texttt{earnings} and \texttt{print} (lines 1822).

[Page 651]

\textbf{Figure 12.10.} \texttt{BasePlusCommissionEmployee} class definition indicating inheritance relationship with class \texttt{CommissionEmployee}.

(This item is displayed on page 650 in the print version)
Figure 12.11. `BasePlusCommissionEmployee` implementation file: private base-class data cannot be accessed from derived class.

(This item is displayed on pages 651 - 653 in the print version)
<< lastName << "\nSocial Security Number: " << socialSecurityNumber
<< "\nGross Sales: " << grossSales
<< "\nCommission Rate: " << commissionRate
<< "\nBase Salary: " << baseSalary;
} // end function print
Figure 12.11 shows BasePlusCommissionEmployee’s member-function implementations. The constructor (lines 1017) introduces base-class initializer syntax (line 14), which uses a member initializer to pass arguments to the base-class (CommissionEmployee) constructor. C++ requires a derived-class constructor to call its base-class constructor to initialize the base-class data members that are inherited into the derived class. Line 14 accomplishes this task by invoking the CommissionEmployee constructor by name, passing the constructor’s parameters first, last, ssn, sales and rate as arguments to initialize base-class data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate. If BasePlusCommissionEmployee’s constructor did not invoke class CommissionEmployee's constructor explicitly, C++ would attempt to invoke class CommissionEmployee’s default constructor but the class does not have such a constructor, so the compiler would issue an error. Recall from Chapter 3 that the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor. However, CommissionEmployee does explicitly include a constructor, so a default constructor is not provided and any attempts to implicitly call CommissionEmployee's default constructor would result in compilation errors.

Common Programming Error 12.1

A compilation error occurs if a derived-class constructor calls one of its base-class constructors with arguments that are inconsistent with the number and types of parameters specified in one of the base-class constructor definitions.

Performance Tip 12.1

In a derived-class constructor, initializing member objects and invoking base-class constructors explicitly in the member initializer list prevents duplicate initialization in which a default constructor is called, then data members are modified again in the derived-class constructor's body.

The compiler generates errors for line 35 of Fig. 12.11 because base class CommissionEmployee’s data members commissionRate and grossSales are private, derived class BasePlusCommissionEmployee's member functions are not allowed to access base class CommissionEmployee's private data. Note that we used red text in Fig. 12.11 to indicate erroneous code. The compiler issues additional errors at lines 4245 of BasePlusCommissionEmployee's print member function for the same reason. As you can see, C++ rigidly enforces restrictions on accessing private data members, so that even a derived class (which is intimately related to its base class) cannot access the base class's private data. [Note: To save space, we show only the error messages from Visual C++ .NET in this example. The error messages produced by your compiler may differ from those shown here. Also notice that we highlight key portions of the lengthy error messages in bold.]
We purposely included the erroneous code in Fig. 12.11 to demonstrate that a derived class's member functions cannot access its base class's private data. The errors in BasePlusCommissionEmployee could have been prevented by using the get member functions inherited from class CommissionEmployee. For example, line 35 could have invoked getCommissionRate and getGrossSales to access CommissionEmployee's private data members commissionRate and grossSales, respectively. Similarly, lines 42-45 could have used appropriate get member functions to retrieve the values of the base class's data members. In the next example, we show how using protected data also allows us to avoid the errors encountered in this example.

Including the Base Class Header File in the Derived Class Header File with #include

Notice that we #include the base class's header file in the derived class's header file (line 10 of Fig. 12.10). This is necessary for three reasons. First, for the derived class to use the base class's name in line 12, we must tell the compiler that the base class exists—the class definition in CommissionEmployee.h does exactly that.

The second reason is that the compiler uses a class definition to determine the size of an object of that class (as we discussed in Section 3.8). A client program that creates an object of a class must #include the class definition to enable the compiler to reserve the proper amount of memory for the object. When using inheritance, a derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes. Including the base class's definition in line 10 allows the compiler to determine the memory requirements for the base class's data members that become part of a derived-class object and thus contribute to the total size of the derived-class object.

The last reason for line 10 is to allow the compiler to determine whether the derived class uses the base class's inherited members properly. For example, in the program of Figs. 12.10-12.11, the compiler uses the base-class header file to determine that the data members being accessed by the derived class are private in the base class. Since these are inaccessible to the derived class, the compiler generates errors. The compiler also uses the base class's function prototypes to validate function calls made by the derived class to the inherited base-class functions. You will see an example of such a function call in Fig. 12.16.

Linking Process in an Inheritance Hierarchy

In Section 3.9, we discussed the linking process for creating an executable GradeBook application. In that example, you saw that the client's object code was linked with the object code for class GradeBook, as well as the object code for any C++ Standard Library classes used in either the client code or in class GradeBook.

The linking process is similar for a program that uses classes in an inheritance hierarchy. The process requires the object code for all classes used in the program and the object code for the direct and indirect base classes of any derived classes used by the program. Suppose a client wants to create an application that uses class BasePlusCommissionEmployee, which is a derived class of CommissionEmployee (we will see an example of this in Section 12.4.4). When compiling the client application, the client's object code must be linked with the object code for classes BasePlusCommissionEmployee and CommissionEmployee, because BasePlusCommissionEmployee inherits member functions from its base class CommissionEmployee. The code is also linked with the object code for any C++ Standard Library classes used in class CommissionEmployee, class BasePlusCommissionEmployee or the client code. This provides the program with access to the implementations of all of the functionality that the
program may use.

12.4.4. CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

To enable class BasePlusCommissionEmployee to directly access CommissionEmployee data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as protected in the base class. As we discussed in Section 12.3, a base class’s protected members can be accessed by members and friends of the base class and by members and friends of any classes derived from that base class.

Good Programming Practice 12.1

Declare public members first, protected members second and private members last.

Defining Base Class CommissionEmployee With protected Data

Class CommissionEmployee (Figs. 12.12, 12.13) now declares data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate as protected (Fig. 12.12, lines 3337) rather than private. The member-function implementations in Fig. 12.13 are identical to those in Fig. 12.5.

Figure 12.12. CommissionEmployee class definition that declares protected data to allow access by derived classes.

(This item is displayed on pages 655 - 656 in the print version)

```cpp
1  // Fig. 12.12: CommissionEmployee.h
2  // CommissionEmployee class definition with protected data.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10  {
11    public:
12      CommissionEmployee( const string &, const string &, const string &,
13                              double = 0.0, double = 0.0 );
14
15      void setFirstName( const string & ); // set first name
16      string getFirstName() const; // return first name
17
18      void setLastName( const string & ); // set last name
19      string getLastName() const; // return last name
20
21      void setSocialSecurityNumber( const string & ); // set SSN
22      string getSocialSecurityNumber() const; // return SSN
23
24      void setGrossSales( double ); // set gross sales amount
25      double getGrossSales() const; // return gross sales amount
```
Figure 12.13. CommissionEmployee class with protected data.
(This item is displayed on pages 656 - 658 in the print version)
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales > 0.0 ) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return commissionRate * grossSales;
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: " << firstName << ' ' << lastName
        << "social security number: " << socialSecurityNumber
        << "gross sales: " << grossSales
        << "commission rate: " << commissionRate;
} // end function print
We now modify class `BasePlusCommissionEmployee` (Figs. 12.14-12.15) so that it inherits from the version of class `CommissionEmployee` in Figs. 12.12-12.13. Because class `BasePlusCommissionEmployee` inherits from this version of class `CommissionEmployee`, objects of class `BasePlusCommissionEmployee` can access inherited data members that are declared `protected` in class `CommissionEmployee` (i.e., data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, and `commissionRate`). As a result, the compiler does not generate errors when compiling the `BasePlusCommissionEmployee` earnings and print member-function definitions in Fig. 12.15 (lines 3236 and 3947, respectively). This shows the special privileges that a derived class is granted to access `protected` baseclass data members. Objects of a derived class also can access `protected` members in any of that derived class’s indirect base classes.

Figure 12.14. `BasePlusCommissionEmployee` class header file.
(This item is displayed on page 658 in the print version)

```cpp
1  // Fig. 12.14: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8    using std::string;
9
10  #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12  class BasePlusCommissionEmployee : public CommissionEmployee
13  {
14  public:
15    BasePlusCommissionEmployee( const string &, const string &, 
16         const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23  private:
24    double baseSalary; // base salary
25  }; // end class BasePlusCommissionEmployee
26  
27  #endif
```

Figure 12.15. `BasePlusCommissionEmployee` implementation file for `BasePlusCommissionEmployee` class that inherits `protected` data from `CommissionEmployee`.

```cpp
1  // Fig. 12.15: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4    using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
```
Class `BasePlusCommissionEmployee` does not inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s constructor (Fig. 12.15, lines 1017) calls class `CommissionEmployee`'s constructor explicitly (line 14). Recall that `BasePlusCommissionEmployee`'s constructor must explicitly call the constructor of class `CommissionEmployee`, because `CommissionEmployee` does not contain a default constructor that could be invoked implicitly.

Testing the Modified `BasePlusCommissionEmployee` Class

Figure 12.16 uses a `BasePlusCommissionEmployee` object to perform the same tasks that Fig. 12.9 performed on an object of the first version of class `BasePlusCommissionEmployee` (Figs. 12.7, 12.8). Note that the outputs of the two programs are identical. We created the first class `BasePlusCommissionEmployee` without using inheritance and created this version of `BasePlusCommissionEmployee` using inheritance; however, both classes provide the same...
functionality. Note that the code for class BasePlusCommissionEmployee (i.e., the header and implementation files), which is 74 lines, is considerably shorter than the code for the noninherited version of the class, which is 154 lines, because the inherited version absorbs part of its functionality from CommissionEmployee, whereas the noninherited version does not absorb any functionality. Also, there is now only one copy of the CommissionEmployee functionality declared and defined in class CommissionEmployee. This makes the source code easier to maintain, modify and debug, because the source code related to a CommissionEmployee exists only in the files of Figs. 12.12, 12.13.

Figure 12.16. Protected base-class data can be accessed from derived class.
(This item is displayed on pages 660 - 661 in the print version)

```cpp
1  // Fig. 12.16: fig12_16.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7  
8  #include <iomanip>
9  using std::setprecision;
10  
11  // BasePlusCommissionEmployee class definition
12  #include "BasePlusCommissionEmployee.h"
13  
14  int main()
15  {
16     // instantiate BasePlusCommissionEmployee object
17     BasePlusCommissionEmployee
18        employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19     
20     // set floating-point output formatting
21     cout << fixed << setprecision( 2 );
22     
23     // get commission employee data
24     cout << "Employee information obtained by get functions: \n"
25        << "\nFirst name is " << employee.getFirstName() 
26        << "\nLast name is " << employee.getLastName() 
27        << "\nSocial security number is " 
28        << employee.getSocialSecurityNumber() 
29        << "\nGross sales is " << employee.getGrossSales() 
30        << "\nCommission rate is " << employee.getCommissionRate() 
31        << "\nBase salary is " << employee.getBaseSalary() << endl;
32     
33     employee.setBaseSalary( 1000 ); // set base salary
34     
35     cout << "\nUpdated employee information output by print function: \n"
36        << endl;
37     employee.print(); // display the new employee information
38     
39     // display the employee's earnings
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41     
42     return 0;
43  } // end main
```
Notes on Using protected Data

In this example, we declared base-class data members as protected, so that derived classes could modify the data directly. Inheriting protected data members slightly increases performance, because we can directly access the members without incurring the overhead of calls to set or get member functions. In most cases, however, it is better to use private data members to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

Using protected data members creates two major problems. First, the derived-class object does not have to use a member function to set the value of the base-class's protected data member. Therefore, a derived-class object easily can assign an invalid value to the protected data member, thus leaving the object in an inconsistent state. For example, with CommissionEmployee's data member grossSales declared as protected, a derived-class (e.g., BasePlusCommissionEmployee) object can assign a negative value to grossSales. The second problem with using protected data members is that derived-class member functions are more likely to be written so that they depend on the base-class implementation. In practice, derived classes should depend only on the base-class services (i.e., non private member functions) and not on the base-class implementation. With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. For example, if for some reason we were to change the names of data members firstName and lastName to first and last, then we would have to do so for all occurrences in which a derived class references these base-class data members directly. In such a case, the software is said to be fragile or brittle, because a small change in the base class can "break" derived-class implementation. The programmer should be able to change the base-class implementation while still providing the same services to derived classes. (Of course, if the base-class services change, we must reimplement our derived classes good object-oriented design attempts to prevent this.)

Software Engineering Observation 12.5
We now reexamine our hierarchy once more, this time using the best software engineering practices. Class `CommissionEmployee` (Figs. 12.17, 12.18) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as private (Fig. 12.17, lines 3337) and provides public member functions `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `print` for manipulating these values. If we decide to change the data member names, the `earnings` and `print` definitions will not require modificationonly the definitions of the get and set member functions that directly manipulate the data members will need to change. Note that these changes occur solely within the base class no changes to the derived class are needed. Localizing the effects of changes like this is a good software engineering practice. Derived class `BasePlusCommissionEmployee` (Figs. 12.19, 12.20) inherits `CommissionEmployee`'s non-private member functions and can access the private base-class members via those member functions.

```
1 // Fig. 12.17: CommissionEmployee.h
2 // CommissionEmployee class definition with good software engineering.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11    public:
12      CommissionEmployee( const string &, const string &, const string &,
```

It is appropriate to use the protected access specifier when a base class should provide a service (i.e., a member function) only to its derived classes (and friends), not to other clients.

Software Engineering Observation 12.6

Declaring base-class data members private (as opposed to declaring them protected) enables programmers to change the base-class implementation without having to change derived-class implementations.

Error-Prevention Tip 12.1

When possible, avoid including protected data members in a base class. Rather, include non-private member functions that access private data members, ensuring that the object maintains a consistent state.

12.4.5. CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

Figure 12.17. CommissionEmployee class defined using good software engineering practices.
Figure 12.18. CommissionEmployee class implementation file: CommissionEmployee class uses member functions to manipulate its private data.

(This item is displayed on pages 663 - 665 in the print version)
return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: "
        << getFirstName() << ' ' << getLastName()
        << "\nsocial security number: " << getSocialSecurityNumber()
Figure 12.19. BasePlusCommissionEmployee class header file.
(This item is displayed on page 666 in the print version)

```cpp
// Fig. 12.19: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee
{
  public:
    BasePlusCommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

    double earnings() const; // calculate earnings
    void print() const; // print BasePlusCommissionEmployee object
  private:
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Figure 12.20. BasePlusCommissionEmployee class that inherits from class CommissionEmployee but cannot directly access the class's private data.
(This item is displayed on pages 666 - 667 in the print version)

```cpp
// Fig. 12.20: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee( const string &first, const string &last, const string &ssn, double sales, double rate, double salary ) :
  CommissionEmployee( first, last, ssn, sales, rate )
{ // explicitly call base-class constructor

  setBaseSalary( salary ); // validate and store base salary
}
```

```cpp
90     << "\ngross sales: " << getGrossSales()
91     << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

```cpp
90        << "\ngross sales: " << getGrossSales()
91        << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```
In the `CommissionEmployee` constructor implementation (Fig. 12.18, lines 916), note that we use member initializers (line 12) to set the values of members `firstName`, `lastName` and `socialSecurityNumber`. We show how derived-class `BasePlusCommissionEmployee` (Figs. 12.19, 12.20) can invoke non-private base-class member functions (`setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber` and `getSocialSecurityNumber`) to manipulate these data members.

Performance Tip 12.2

Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, today's optimizing compilers are carefully designed to perform many optimizations implicitly (such as inlining set and get member-function calls). As a result, programmers should write code that adheres to proper software engineering principles, and leave optimization issues to the compiler. A good rule is, "Do not second-guess the compiler."

---

Class `BasePlusCommissionEmployee` (Figs. 12.19, 12.20) has several changes to its member-function implementations (Fig. 12.20) that distinguish it from the previous version of the class (Figs.

---

file://C:\Documents and Settings\Administrator\Local Settings\Temp\~hhEEC6.htm
Member functions `earnings` (Fig. 12.20, lines 3235) and `print` (lines 3846) each invoke member function `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. This insulates `earnings` and `print` from potential changes to the implementation of data member `baseSalary`. For example, if we decide to rename data member `baseSalary` or change its type, only member functions `setBaseSalary` and `getBaseSalary` will need to change.

Class `BasePlusCommissionEmployee`'s `earnings` function (Fig. 12.20, lines 3235) redefines class `CommissionEmployee`'s `earnings` member function (Fig. 12.18, lines 7982) to calculate the earnings of a base-salaried commission employee. Class `BasePlusCommissionEmployee`'s version of `earnings` obtains the portion of the employee's earnings based on commission alone by calling base-class `CommissionEmployee`'s `earnings` function with the expression `CommissionEmployee::earnings()` (Fig. 12.20, line 34). `BasePlusCommissionEmployee`'s `earnings` function then adds the base salary to this value to calculate the total earnings of the employee. Note the syntax used to invoke a redefined base-class member function from a derived class: place the base-class name and the binary scope resolution operator (`::`) before the base-class member-function name. This member-function invocation is a good software engineering practice. Recall from Software Engineering Observation 9.9 that, if an object's member function performs the actions needed by another object, we should call that member function rather than duplicating its code body. By having `BasePlusCommissionEmployee`'s `earnings` function invoke `CommissionEmployee`'s `earnings` function to calculate part of a `BasePlusCommissionEmployee` object's earnings, we avoid duplicating the code and reduce code-maintenance problems.

Common Programming Error 12.2

When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the `::` operator prefixed with the name of the base class when referencing the base class's member function causes infinite recursion, because the derived-class member function would then call itself.

Common Programming Error 12.3

Including a base-class member function with a different signature in the derived class hides the base-class version of the function. Attempts to call the base-class version through the public interface of a derived-class object result in compilation errors.

Similarly, `BasePlusCommissionEmployee`'s `print` function (Fig. 12.20, lines 3846) redefines class `CommissionEmployee`'s `print` member function (Fig. 12.18, lines 8592) to output information that is appropriate for a base-salaried commission employee. Class `BasePlusCommissionEmployee`'s version displays part of a `BasePlusCommissionEmployee` object's information (i.e., the string "commission employee" and the values of class `CommissionEmployee`'s private data members) by calling `CommissionEmployee`'s `print` member function with the qualified name `CommissionEmployee::print()` (Fig. 12.20, line 43). `BasePlusCommissionEmployee`'s `print`
function then outputs the remainder of a `BasePlusCommissionEmployee` object's information (i.e., the value of class `BasePlusCommissionEmployee`'s base salary).

**Figure 12.21** performs the same manipulations on a `BasePlusCommissionEmployee` object as did **Fig. 12.9** and **Fig. 12.16** on objects of classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively. Although each "base-salaried commission employee" class behaves identically, class `BasePlusCommissionEmployee` is the best engineered. By using inheritance and by calling member functions that hide the data and ensure consistency, we have efficiently and effectively constructed a well-engineered class.

**Figure 12.21. Base-class private data is accessible to a derived class via public or protected member function inherited by the derived class.**

(This item is displayed on pages 668 - 669 in the print version)

```cpp
1  // Fig. 12.21: fig12_21.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11  // BasePlusCommissionEmployee class definition
12  #include "BasePlusCommissionEmployee.h"
13
14  int main()
15  {
16     // instantiate BasePlusCommissionEmployee object
17     BasePlusCommissionEmployee
18        employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // set floating-point output formatting
21     cout << fixed << setprecision( 2 );
22
23     // get commission employee data
24     cout << "Employee information obtained by get functions: \n"
25        << "\nFirst name is " << employee.getFirstName()
26        << "\nLast name is " << employee.getLastName()
27        << "\nSocial security number is "
28        << employee.getSocialSecurityNumber()
29        << "\nGross sales is " << employee.getGrossSales()
30        << "\nCommission rate is " << employee.getCommissionRate()
31        << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary( 1000 ); // set base salary
34
35     cout << "\nUpdated employee information output by print function: \n"
36        << endl;
37     employee.print(); // display the new employee information
38
39     // display the employee's earnings
40     cout << "\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43  } // end main
```
In this section, you saw an evolutionary set of examples that was carefully designed to teach key capabilities for good software engineering with inheritance. You learned how to create a derived class using inheritance, how to use protected base-class members to enable a derived class to access inherited base-class data members and how to redefine base-class functions to provide versions that are more appropriate for derived-class objects. In addition, you learned how to apply software engineering techniques from Chapters 9 and 10 and this chapter to create classes that are easy to maintain, modify and debug.

12.5. Constructors and Destructors in Derived Classes

As we explained in the preceding section, instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor). Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing first. The original derived-class constructor's body finishes executing last. Each base-class constructor initializes the base-class data members that the derived-class object inherits. For example, consider the CommissionEmployee/BasePlusCommissionEmployee hierarchy from Figs. 12.17, 12.20. When a program creates an object of class BasePlusCommissionEmployee, the CommissionEmployee constructor is called. Since class CommissionEmployee is at the base of the hierarchy, its constructor executes, initializing the private data members of CommissionEmployee that are part of the BasePlusCommissionEmployee object. When CommissionEmployee's constructor completes...
Software Engineering Observation 12.7

When a program creates a derived-class object, the derived-class constructor immediately calls the base-class constructor, the base-class constructor's body executes, then the derived class's member initializers execute and finally the derived-class constructor's body executes. This process cascades up the hierarchy if the hierarchy contains more than two levels.

When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in reverse of the order in which the constructors executed. When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

Software Engineering Observation 12.8

Suppose that we create an object of a derived class where both the base class and the derived class contain objects of other classes. When an object of that derived class is created, first the constructors for the base class's member objects execute, then the base-class constructor executes, then the constructors for the derived class's member objects execute, then the derived class's constructor executes. Destructors for derived-class objects are called in the reverse of the order in which their corresponding constructors are called.

Base-class constructors, destructors and overloaded assignment operators (see Chapter 11, Operator Overloading; String and Array Objects) are not inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class constructors, destructors and overloaded assignment operators.

Our next example revisits the commission employee hierarchy by defining class CommissionEmployee (Figs. 12.22-12.23) and class BasePlusCommissionEmployee (Figs. 12.24-12.25) that contain constructors and destructors, each of which prints a message when it is invoked. As you will see in the output in Fig. 12.26, these messages demonstrate the order in which the constructors and destructors are called for objects in an inheritance hierarchy.

Figure 12.22. CommissionEmployee class header file.
class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
        double = 0.0, double = 0.0 );
    ~CommissionEmployee(); // destructor

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate( double ); // set commission rate
    double getCommissionRate() const; // return commission rate

    double earnings() const; // calculate earnings
    void print() const; // print CommissionEmployee object

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee

// Fig. 12.23: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
using std::cout;
using std::endl;

#include "CommissionEmployee.h" // CommissionEmployee class definition

// constructor
CommissionEmployee::CommissionEmployee(
        const string &first, const string &last, const string &ssn,
        double sales, double rate )
{   
    firstName( first ), lastName( last ), socialSecurityNumber( ssn )

    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate

    cout << "CommissionEmployee constructor: " << endl;
    print();
    cout << "\n\n";
} // end CommissionEmployee constructor
// destructor
CommissionEmployee::~CommissionEmployee()
{
    cout << "CommissionEmployee destructor: " << endl;
    print();
    cout << "\n\n";
} // end CommissionEmployee destructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate
```cpp
85   // return commission rate
86   double CommissionEmployee::getCommissionRate() const
87   {
88      return commissionRate;
89   } // end function getCommissionRate
90
91   // calculate earnings
92   double CommissionEmployee::earnings() const
93   {
94      return getCommissionRate() * getGrossSales();
95   } // end function earnings
96
97   // print CommissionEmployee object
98   void CommissionEmployee::print() const
99   {
100      cout << "commission employee: "
101         << getFirstName() << ' ' << getLastName()
102         << "social security number: " << getSocialSecurityNumber()
103         << "gross sales: " << getGrossSales()
104         << "commission rate: " << getCommissionRate();
105   } // end function print
```

**Figure 12.24.** BasePlusCommissionEmployee class header file.

(This item is displayed on page 674 in the print version)
In this example, we modified the CommissionEmployee constructor (lines 1021 of Fig. 12.23) and included a CommissionEmployee destructor (lines 2429), each of which outputs a line of text upon its invocation. We also modified the BasePlusCommissionEmployee constructor (lines 1122 of Fig. 12.25) and included a BasePlusCommissionEmployee destructor (lines 2530), each of which outputs a line of text upon its invocation.

Figure 12.25. BasePlusCommissionEmployee’s constructor outputs text.
(This item is displayed on pages 674 - 675 in the print version)
Figure 12.26 demonstrates the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Function main (lines 1534) begins by instantiating CommissionEmployee object employee1 (lines 2122) in a separate block inside main (lines 2023). The object goes in and out of scope immediately (the end of the block is reached as soon as the object is created), so both the CommissionEmployee constructor and destructor are called. Next, lines 2627 instantiate BasePlusCommissionEmployee object employee2. This invokes the CommissionEmployee constructor to display outputs with values passed from the BasePlusCommissionEmployee constructor, then the output specified in the BasePlusCommissionEmployee constructor is performed. Lines 3031 then instantiate BasePlusCommissionEmployee object employee3. Again, the CommissionEmployee and BasePlusCommissionEmployee constructors are both called. Note that, in each case, the body of the CommissionEmployee constructor is executed before the body of the BasePlusCommissionEmployee constructor executes. When the end of main is reached, the destructors are called for objects employee2 and employee3. But, because destructors are called in the reverse order of their corresponding constructors, the BasePlusCommissionEmployee destructor and CommissionEmployee destructor are called (in that order) for object employee3, then the BasePlusCommissionEmployee and CommissionEmployee destructors are called (in that order) for object employee2.

Figure 12.26. Constructor and destructor call order.

(This item is displayed on pages 676 - 678 in the print version)
{ // begin new scope
    CommissionEmployee employee1("Bob", "Lewis", "333-33-3333", 5000, .04 );
} // end scope

cout << endl;
BasePlusCommissionEmployee
    employee2( "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
cout << endl;

BasePlusCommissionEmployee
    employee3( "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
cout << endl;
return 0;
} // end main

CommissionEmployee constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee destructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee constructor:
basesalaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06

BasePlusCommissionEmployee constructor:
basesalaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePlusCommissionEmployee constructor:
basesalaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

BasePlusCommissionEmployee destructor:
basesalaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
12.6. public, protected and private Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. Use of protected and private inheritance is rare, and each should be used only with great care; we normally use public inheritance in this book. (Chapter 21 demonstrates private inheritance as an alternative to composition.) Figure 12.27 summarizes for each type of inheritance the accessibility of base-class members in a derived class. The first column contains the base-class access specifiers.

Figure 12.27. Summary of base-class member accessibility in a derived class.
(This item is displayed on page 679 in the print version)
When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class. When deriving from a private base class, public and protected members of the base class become private members (e.g., the functions become utility functions) of the derived class. Private and protected inheritance are not is-a relationships.

12.7. Software Engineering with Inheritance

In this section, we discuss the use of inheritance to customize existing software. When we use inheritance to create a new class from an existing one, the new class inherits the data members and member functions of the existing class, as described in Fig. 12.27. We can customize the new class to meet our needs by including additional members and by redefining base-class members. The derived-class programmer does this in C++ without accessing the base class's source code. The derived class must be able to link to the base class's object code. This powerful capability is attractive to independent software vendors (ISVs). ISVs can develop proprietary classes for sale or license and make these classes available to users in object-code format. Users then can derive new classes from these library classes rapidly and without accessing the ISVs' proprietary source code. All the ISVs need to supply with the object code are the header files.
Sometimes it is difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software development process. Object-oriented programming facilitates software reuse, thus shortening development times and enhancing software quality.

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. Just as shrink-wrapped software produced by independent software vendors became an explosive-growth industry with the arrival of the personal computer, so, too, interest in the creation and sale of class libraries is growing exponentially. Application designers build their applications with these libraries, and library designers are being rewarded by having their libraries included with the applications. The standard C++ libraries that are shipped with C++ compilers tend to be rather general purpose and limited in scope. However, there is massive worldwide commitment to the development of class libraries for a huge variety of applications arenas.

Software Engineering Observation 12.9

At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should "factor out" common attributes and behaviors and place these in a base class, then use inheritance to form derived classes, endowing them with capabilities beyond those inherited from the base class.

Software Engineering Observation 12.10

The creation of a derived class does not affect its base class's source code. Inheritance preserves the integrity of a base class.

Software Engineering Observation 12.11

Just as designers of non-object-oriented systems should avoid proliferation of functions, designers of object-oriented systems should avoid proliferation of classes. Proliferation of classes creates management problems and can hinder software reusability, because it becomes difficult for a client to locate the most appropriate class of a huge class library. The alternative is to create fewer classes that provide more substantial functionality, but such classes might provide too much functionality.

Performance Tip 12.3

If classes produced through inheritance are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Inherit from the class whose functionality is "closest" to what is
Reading derived-class definitions can be confusing, because inherited members are not shown physically in the derived classes, but nevertheless are present. A similar problem exists when documenting derived-class members.

12.8. Wrap-Up

This chapter introduced inheritance—the ability to create a class by absorbing an existing class’s data members and member functions and embellishing them with new capabilities. Through a series of examples using an employee inheritance hierarchy, you learned the notions of base classes and derived classes and used public inheritance to create a derived class that inherits members from a base class. The chapter introduced the access specifier protected; derived-class member functions can access protected base-class members. You learned how to access redefined base-class members by qualifying their names with the base-class name and binary scope resolution operator (::). You also saw the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Finally, we explained the three types of inheritance—public, protected, and private—and the accessibility of base-class members in a derived class when using each type.

In Chapter 13, Object-Oriented Programming: Polymorphism, we build upon our discussion of inheritance by introducing polymorphism—an object-oriented concept that enables us to write programs that handle, in a more general manner, objects of a wide variety of classes related by inheritance. After studying Chapter 13, you will be familiar with classes, objects, encapsulation, inheritance and polymorphism—the essential aspects of object-oriented programming.

Summary

- Software reuse reduces program development time and cost.
- Inheritance is a form of software reuse in which the programmer creates a class that absorbs an existing class’s data and behaviors and enhances them with new capabilities. The existing class is called the base class, and the new class is referred to as the derived class.
- A direct base class is the one from which a derived class explicitly inherits (specified by the class name to the right of the : in the first line of a class definition). An indirect base class is inherited from two or more levels up in the class hierarchy.
- With single inheritance, a class is derived from one base class. With multiple inheritance, a class inherits from multiple (possibly unrelated) base classes.
A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors. A derived class can also customize behaviors inherited from the base class.

Every object of a derived class is also an object of that class's base class. However, a base-class object is not an object of that class's derived classes.

The is-a relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class.

The has-a relationship represents composition. In a has-a relationship, an object contains one or more objects of other classes as members, but does not disclose their behavior directly in its interface.

A derived class cannot access the private members of its base class directly; allowing this would violate the encapsulation of the base class. A derived class can, however, access the public and protected members of its base class directly.

A derived class can effect state changes in private base-class members, but only through non-private member functions provided in the base class and inherited into the derived class.

When a base-class member function is inappropriate for a derived class, that member function can be redefined in the derived class with an appropriate implementation.

Single-inheritance relationships form treelike hierarchical structures—a base class exists in a hierarchical relationship with its derived classes.

It is possible to treat base-class objects and derived-class objects similarly; the commonality shared between the object types is expressed in the data members and member functions of the base class.

A base class's public members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of that base class's derived classes, or, when using the binary scope resolution operator, whenever the class's name is in scope.

A base class's private members are accessible only within the definition of that base class or from friends of that class.

A base class's protected members have an intermediate level of protection between public and private access. A base class's protected members can be accessed by members and friends of that base class and by members and friends of any classes derived from that base class.

Unfortunately, protected data members often present two major problems. First, the derived-class object does not have to use a set function to change the value of the base-class's protected data. Second, derived-class member functions are more likely to depend on base-class implementation details.

When a derived-class member function redefines a base-class member function, the base-class member function can be accessed from the derived class by qualifying the base-class member function name with the base-class name and the binary scope resolution operator (::).
When an object of a derived class is instantiated, the base class’s constructor is called immediately (either explicitly or implicitly) to initialize the base-class data members in the derived-class object (before the derived-class data members are initialized).

Declaring data members `private`, while providing non-private member functions to manipulate and perform validation checking on this data, enforces good software engineering.

When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors: first the derived-class destructor is called, then the base-class destructor is called.

When deriving a class from a base class, the base class may be declared as either `public`, `protected`, or `private`.

When deriving a class from a `public` base class, `public` members of the base class become `public` members of the derived class, and `protected` members of the base class become `protected` members of the derived class.

When deriving a class from a `protected` base class, `public` and `protected` members of the base class become protected members of the derived class.

When deriving a class from a `private` base class, `public` and `protected` members of the base class become `private` members of the derived class.

Terminology

- **base class**
- **base-class constructor**
- **base-class default constructor**
- **base-class destructor**
- **base-class initializer**
- **brittle software**
- **class hierarchy**
- **composition**
- **customize software**
- **derived class**
- **derived-class constructor**
derived-class destructor

direct base class

fragile software

friend of a base class

friend of a derived class

has-a relationship

hierarchical relationship

indirect base class

inherit the members of an existing class

inheritance

is-a relationship

multiple inheritance

private base class

private inheritance

protected base class

protected inheritance

protected keyword

protected member of a class

public base class

public inheritance

qualified name

redefine a base-class member function

single inheritance

subclass

superclass
Self-Review Exercises

12.1 Fill in the blanks in each of the following statements:

a. __________ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

b. A base class's __________ members can be accessed only in the base-class definition or in derived-class definitions.

c. In a(n) __________ relationship, an object of a derived class also can be treated as an object of its base class.

d. In a(n) __________ relationship, a class object has one or more objects of other classes as members.

e. In single inheritance, a class exists in a(n) __________ relationship with its derived classes.

f. A base class's __________ members are accessible within that base class and anywhere that the program has a handle to an object of that base class or to an object of one of its derived classes.

g. A base class's protected access members have a level of protection between those of public and __________ access.

h. C++ provides for __________, which allows a derived class to inherit from many base classes, even if these base classes are unrelated.

i. When an object of a derived class is instantiated, the base class's __________ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

j. When deriving a class from a base class with public inheritance, public members of the base class become __________ members of the derived class, and protected members of the base class become __________ members of the derived class.

k. When deriving a class from a base class with protected inheritance, public members of the base class become __________ members of the derived class, and protected members of the base class become __________ members of the derived class.

12.2 State whether each of the following is true or false. If false, explain why.

a. Base-class constructors are not inherited by derived classes.
b. A has-a relationship is implemented via inheritance.

c. A Car class has an is-a relationship with the SteeringWheel and Brakes classes.

d. Inheritance encourages the reuse of proven high-quality software.

e. When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

Answers to Self-Review Exercises

12.1 a) Inheritance. b) protected. c) is-a or inheritance. d) has-a or composition or aggregation. e) hierarchical. f) public. g) private. h) multiple inheritance. i) constructor. j) public, protected. k) protected, protected.

12.2 a) True. b) False. A has-a relationship is implemented via composition. An is-a relationship is implemented via inheritance. c) False. This is an example of a has-a relationship. Class Car has an is-a relationship with class Vehicle. d) True. e) True.

Exercises

12.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class BasePlusCommissionEmployee of the CommissionEmployeeBasePlusCommissionEmployee hierarchy to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes CommissionEmployee and BasePlusCommissionEmployee, as well as for object-oriented programs in general. Which approach is more natural? Why?

12.4 Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

12.5 Some programmers prefer not to use protected access because they believe it breaks the encapsulation of the base class. Discuss the relative merits of using protected access vs. using private access in base classes.

12.6 Draw an inheritance hierarchy for students at a university similar to the hierarchy
shown in Fig. 12.2. Use `Student` as the base class of the hierarchy, then include classes `UndergraduateStudent` and `GraduateStudent` that derive from `Student`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might derive from `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might derive from `GraduateStudent`. After drawing the hierarchy, discuss the relationships that exist between the classes. [Note: You do not need to write any code for this exercise.]

**12.7** The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 12.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have base class `Shape` from which class `TwoDimensionalShape` and class `ThreeDimensionalShape` are derived. [Note: You do not need to write any code for this exercise.] We will use this hierarchy in the exercises of Chapter 13 to process a set of distinct shapes as objects of base-class `Shape`. (This technique, called polymorphism, is the subject of Chapter 13.)

**12.8** Draw an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the base class of the hierarchy. Make the hierarchy as deep as possible.

**12.9** (Package Inheritance Hierarchy) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use `Package` as the base class of the hierarchy, then include classes `TwoDayPackage` and `OvernightPackage` that derive from `Package`. Base class `Package` should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. `Package`'s constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. `Package` should provide a `public` member function `calculateCost` that returns a `double` indicating the cost associated with shipping the package. `Package`'s `calculateCost` function should determine the cost by multiplying the weight by the cost per ounce. Derived class `TwoDayPackage` should inherit the functionality of base class `Package`, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. `TwoDayPackage`'s constructor should receive a value to initialize this data member. `TwoDayPackage` should redefine member function `calculateCost` so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class `Package`'s `calculateCost` function. Class `OvernightPackage` should inherit directly from class `Package` and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. `OvernightPackage` should redefine member function `calculateCost` so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a test program that creates objects of each type of `Package` and tests member function `calculateCost`.

**12.10** (Account Inheritance Hierarchy) Create an inheritance hierarchy that a bank might
use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class `Account` and derived classes `SavingsAccount` and `CheckingAccount` that inherit from class `Account`. Base class `Account` should include one data member of type `double` to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to 0.0. If not, the balance should be set to 0.0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function `getBalance` should return the current balance.

Derived class `SavingsAccount` should inherit the functionality of an `Account`, but also include a data member of type `double` indicating the interest rate (percentage) assigned to the `Account`. `SavingsAccount`'s constructor should receive the initial balance, as well as an initial value for the `SavingsAccount`'s interest rate. `SavingsAccount` should provide a public member function `calculateInterest` that returns a `double` indicating the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance. [Note: `SavingsAccount` should inherit member functions `credit` and `debit` as is without redefining them.]

Derived class `CheckingAccount` should inherit from base class `Account` and include an additional data member of type `double` that represents the fee charged per transaction. `CheckingAccount`'s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class `CheckingAccount` should redefine member functions `credit` and `debit` so that they subtract the fee from the account balance whenever either transaction is performed successfully. `CheckingAccount`'s versions of these functions should invoke the base-class `Account` version to perform the updates to an account balance. `CheckingAccount`'s `debit` function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [Hint: Define `Account`'s `debit` function so that it returns a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the `SavingsAccount` object by first invoking its `calculateInterest` function, then passing the returned interest amount to the object's `credit` function.