11.3.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy (cont.)

- For example, line 37 could have invoked `getCommissionRate` and `getGrossSales` to access `CommissionEmployee`'s `private` data members `commissionRate` and `grossSales`, respectively.
- Similarly, lines 44–47 could have used appropriate *get* member functions to retrieve the values of the base class's data members.

11.3.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy (cont.)

Including the Base-Class Header in the Derived-Class Header with `#include`

- We `#include` the base class's header in the derived class's header (line 8 of Fig. 11.10).
- This is necessary for three reasons.
  - The derived class uses the base class's name in line 10, so we must tell the compiler that the base class exists.
  - The compiler uses a class definition to determine the size of an object of that class. A client program that creates an object of a class `#include`s the class definition to enable the compiler to reserve the proper amount of memory for the object.
  - The compiler must determine whether the derived class uses the base class's inherited members properly.

11.3.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy (cont.)

Linking Process in an Inheritance Hierarchy

- In Section 3.7, we discussed the linking process for creating an executable `GradeBook` application.
- 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.
- The code is also linked with the object code for any C++ Standard Library classes used in the classes or the client code.

11.3.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

- In this section, we introduce the access specifier `protected`.
- 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.
- A base class's `protected` members can be accessed within the body of that base class, by members and `friend`s of that base class, and by members and `friend`s of any classes derived from that base class.

11.3.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data (cont.)

Defining Base Class CommissionEmployee with protected Data

- Class `CommissionEmployee` (Fig. 11.12) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `protected` (lines 31–36) rather than `private`.
- The member-function implementations are identical to those in Fig. 11.5.

```
1  // Fig. 11.12: 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
8  class CommissionEmployee
9  {
10 public:
11    CommissionEmployee( const std::string &, const std::string &,
12       const std::string &, double = 0.0, double = 0.0 );
13
14    void setFirstName( const std::string & ); // set first name
15    std::string getFirstName() const; // return first name
16
17    void setLastName( const std::string & ); // set last name
18    std::string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const std::string & ); // set SSN
21    std::string getSocialSecurityNumber() const; // return SSN
22
```

Fig. 11.12 | CommissionEmployee class definition that declares protected data to allow access by derived classes. (Part 1 of 2.)

```
23      void setGrossSales( double ); // set gross sales amount
24      double getGrossSales() const; // return gross sales amount
25
26      void setCommissionRate( double ); // set commission rate
27      double getCommissionRate() const; // return commission rate
28
29      double earnings() const; // calculate earnings
30      void print() const; // print CommissionEmployee object
31   protected:
32      std::string firstName;
33      std::string lastName;
34      std::string socialSecurityNumber;
35      double grossSales; // gross weekly sales
36      double commissionRate; // commission percentage
37   }; // end class CommissionEmployee
38
39   #endif
```

Fig. 11.12 | `CommissionEmployee` class definition that declares `protected` data to allow access by derived classes. (Part 2 of 2.)

11.3.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data (cont.)

- BasePlusCommissionEmployee inherits from class CommissionEmployee in Fig. 11.12.
- 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. 11.11 (lines 34–38 and 41–49, respectively).
- Objects of a derived class also can access protected members in any of that derived class's *indirect* base classes.

11.3.4 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Data (cont.)

Testing the Modified `BasePlusCommissionEmployee` Class

- To test the updated class hierarchy, we reused the test program from Fig. 11.9.
- As shown in Fig. 11.13, the output is identical to that of Fig. 11.9.
- The code for class `BasePlusCommissionEmployee`, which is 74 lines, is considerably shorter than the code for the noninherited version of the class, which is 161 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`.
  - Makes the source code easier to maintain, modify and debug.

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Fig. 11.13 | `protected` base-class data can be accessed from derived class.

11.3.4 `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Data (cont.)

Notes on Using `protected` Data

- 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.

Software Engineering Observation 11.3

In most cases, it's 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.

11.3.4 `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Data (cont.)

- Using `protected` data members creates two serious problems.
  - The derived-class object does not have to use a member function to set the value of the base class's `protected` data member.
  - Derived-class member functions are more likely to be written so that they depend on the base-class implementation. 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.
- Such software is said to be fragile or brittle, because a small change in the base class can "break" derived-class implementation.

Software Engineering Observation 11.4

It's appropriate to use the `protected` access specifier when a base class should provide a service (i.e., a non-`private` member function) only to its derived classes and `friend`s.

Software Engineering Observation 11.5

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