Object-Oriented Programming: Inheritance

Based on Chapter 11 of C++ How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn what inheritance is.
- Understand the notions of base classes and derived classes and the relationships between them.
- Use the `protected` member access specifier.
- Use constructors and destructors in inheritance hierarchies.
- Understand the order in which constructors and destructors are called in inheritance hierarchies.
- Understand the differences between `public`, `protected` and `private` inheritance.
- Use inheritance to customize existing software.

11.1 Introduction

- Inheritance
  - A form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.
- You can designate that a 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.
- A derived class represents a *more specialized* group of objects.
- C++ offers `public`, `protected` and `private` inheritance.
- 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.

11.1 Introduction (cont.)

- With object-oriented programming, you 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.
- By contrast, the *has-a* relationship represents composition.

11.2 Base Classes and Derived Classes

- Figure 11.1 lists several simple examples of base classes and derived classes.
  - Base classes tend to be *more general* and derived classes tend to be *more specific*.
- 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.
- Inheritance relationships form class hierarchies.

11.2 Base Classes and Derived Classes (cont.)

- A base class exists in a hierarchical relationship with its derived classes.
- Although classes can exist independently, once they're 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*.

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Fig. 11.1 | Inheritance examples.

11.2.1 CommunityMember Class Hierarchy

- Let's develop a simple inheritance hierarchy with five levels (represented by the UML class diagram in Fig. 11.2).
- A university community has thousands of `CommunityMembers`.
- `Employees` are either `Faculty` or `Staff`.
- `Faculty` are either `Administrators` or `Teachers`.
- Some `Administrators`, however, are also `Teachers`.
- We've used *multiple inheritance* to form class `AdministratorTeacher`.

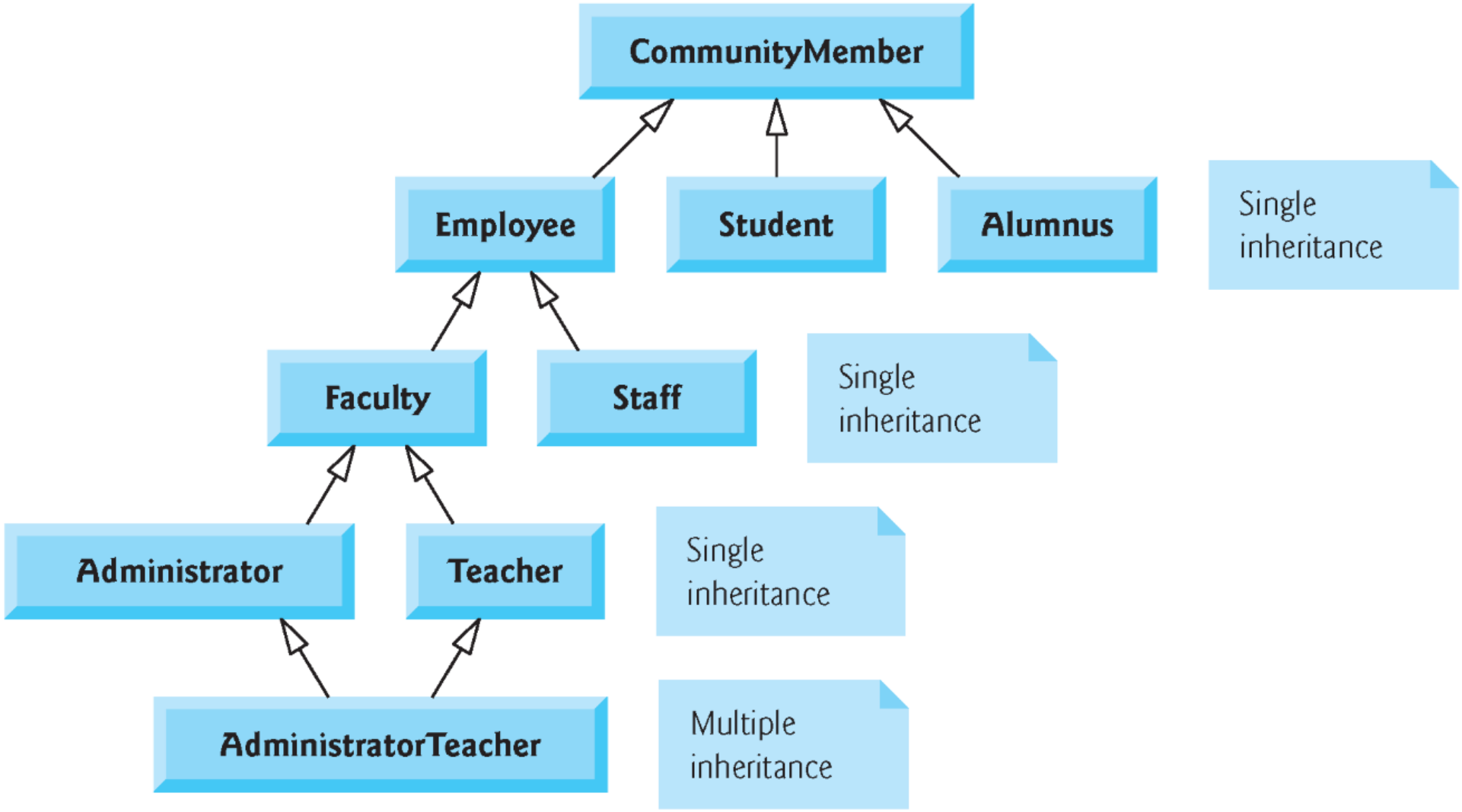

Fig. 11.2 | Inheritance hierarchy for university `CommunityMembers`.

11.2.1 CommunityMember Class Hierarchy

- With single inheritance, a class is derived from *one* base class.
- With multiple inheritance, a derived class inherits simultaneously from *two or more* (possibly unrelated) base classes.

11.2.1 CommunityMember Class Hierarchy

- Each arrow in the hierarchy (Fig. 11.2) represents an *is-a relationship.*
  - 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`.
  - `CommunityMember` is an indirect base class of all the other classes in the diagram.
- Starting from the bottom of the diagram, you can follow the arrows and apply the *is-a* relationship to the topmost base class.
  - An `AdministratorTeacher` *is an* `Administrator`, *is a* `Faculty` member, *is an* `Employee` and *is a* `CommunityMember`.

11.2.2 Shape Class Hierarchy

- Consider the `Shape` inheritance hierarchy in Fig. 11.3.
- Begins with base class `Shape`.
- Classes `TwoDimensionalShape` and `ThreeDimensionalShape` derive from base class `Shape`—`Shapes` are either `TwoDimensionalShapes` or `Three-DimensionalShapes`.
- The third level of this hierarchy contains some more specific types of `TwoDimensionalShapes` and `ThreeDimensionalShapes`.
- As in Fig. 11.2, 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.

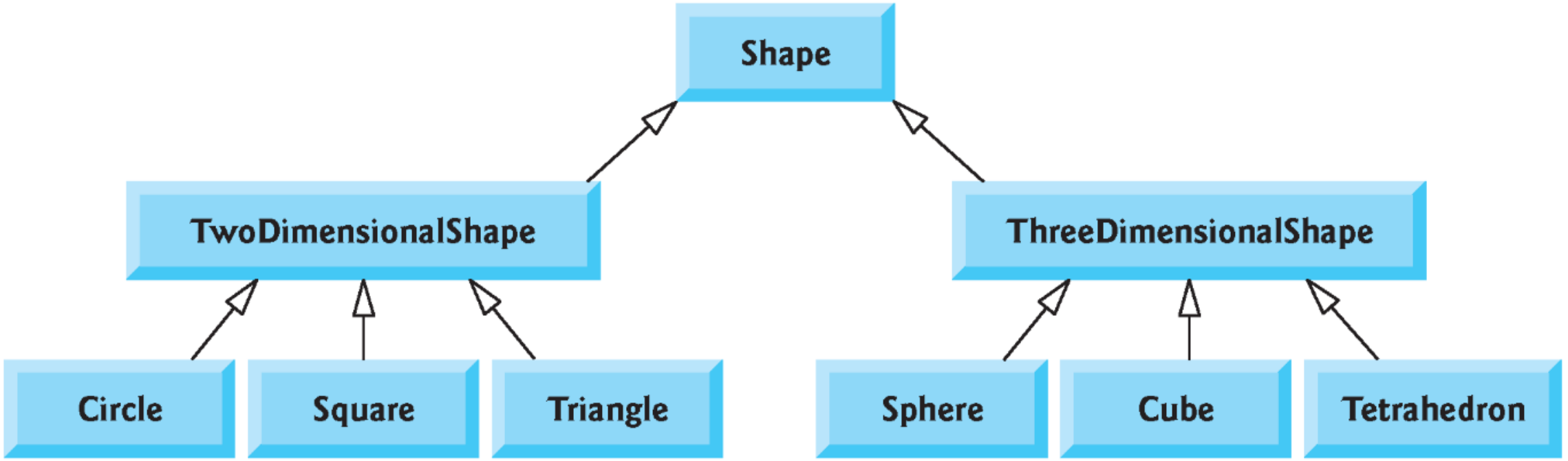

Fig. 11.3 | Inheritance hierarchy for Shapes.

11.3 Relationship between Base 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.

11.3.1 Creating and Using a CommissionEmployee Class

- `CommissionEmployee`'s class definition (Figs. 11.4–11.5).
- `CommissionEmployee`'s `public` services include a constructor and member functions `earnings` and `toString`.
- Also includes `public` *get* and *set* functions that manipulate the class's data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`.
  - `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.

```
1  // Fig. 11.4: 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
8  class CommissionEmployee {
9  public:
10    CommissionEmployee(const std::string&, const std::string&,
11       const std::string&, double = 0.0, double = 0.0);
12
13    void setFirstName(const std::string&); // set first name
14    std::string getFirstName() const; // return first name
15
16    void setLastName(const std::string&); // set last name
17    std::string getLastName() const; // return last name
18
19    void setSocialSecurityNumber(const std::string&); // set SSN
20    std::string getSocialSecurityNumber() const; // return SSN
```

Fig. 11.4 | CommissionEmployee class definition represents a commission employee. (Part 1 of 2.)

```
21
22      void setGrossSales(double); // set gross sales amount
23      double getGrossSales() const; // return gross sales amount
24
25      void setCommissionRate(double); // set commission rate (percentage)
26      double getCommissionRate() const; // return commission rate
27
28      double earnings() const; // calculate earnings
29      std::string toString() const; // create string representation
30  private:
31      std::string firstName;
32      std::string lastName;
33      std::string socialSecurityNumber;
34      double grossSales; // gross weekly sales
35      double commissionRate; // commission percentage
36  };
37
38  #endif
```

Fig. 11.4 | `CommissionEmployee` class definition represents a commission employee. (Part 2 of 2.)

```
1  // Fig. 11.5: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iomanip>
4  #include <stdexcept>
5  #include <sstream>
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7  using namespace std;
8
9  // constructor
10 CommissionEmployee::CommissionEmployee(const string& first,
11    const string& last, const string& ssn, double sales, double rate) {
12    firstName = first; // should validate
13    lastName = last; // should validate
14    socialSecurityNumber = ssn; // should validate
15    setGrossSales(sales); // validate and store gross sales
16    setCommissionRate(rate); // validate and store commission rate
17 }
18
19 // set first name
20 void CommissionEmployee::setFirstName(const string& first) {
21    firstName = first; // should validate
22 }
```

Fig. 11.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 1 of 4.)

```
23
24  // return first name
25  string CommissionEmployee::getFirstName() const {return firstName;}
26
27  // set last name
28  void CommissionEmployee::setLastName(const string& last) {
29     lastName = last; // should validate
30  }
31
32  // return last name
33  string CommissionEmployee::getLastName() const {return lastName;}
34
35  // set social security number
36  void CommissionEmployee::setSocialSecurityNumber(const string& ssn) {
37     socialSecurityNumber = ssn; // should validate
38  }
39
40  // return social security number
41  string CommissionEmployee::getSocialSecurityNumber() const {
42     return socialSecurityNumber;
43  }
44
```

Fig. 11.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 2 of 4.)

```
45  // set gross sales amount
46  void CommissionEmployee::setGrossSales(double sales) {
47     if (sales < 0.0) {
48        throw invalid_argument("Gross sales must be >= 0.0");
49     }
50
51     grossSales = sales;
52  }
53
54  // return gross sales amount
55  double CommissionEmployee::getGrossSales() const {return grossSales;}
56
57  // set commission rate
58  void CommissionEmployee::setCommissionRate(double rate) {
59     if (rate <= 0.0 || rate >= 1.0) {
60        throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
61     }
62
63     commissionRate = rate;
64  }
```

Fig. 11.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 3 of 4.)

```
65
66  // return commission rate
67  double CommissionEmployee::getCommissionRate() const {
68     return commissionRate;
69  }
70
71  // calculate earnings
72  double CommissionEmployee::earnings() const {
73     return commissionRate * grossSales;
74  }
75
76  // return string representation of CommissionEmployee object
77  string CommissionEmployee::toString() const {
78     ostringstream output;
79     output << fixed << setprecision(2); // two digits of precision
80     output << "commission employee: " << firstName << " " << lastName
81        << "\nsocial security number: " << socialSecurityNumber
82        << "\ngross sales: " << grossSales
83        << "\ncommission rate: " << commissionRate;
84     return output.str();
85  }
```

Fig. 11.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 4 of 4.)

11.3.1 Creating and Using a `CommissionEmployee` Class (cont.)

`CommissionEmployee` Constructor

- 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.
  - Later in this section, we'll return to using member-initializer lists in the constructors.

11.3.1 Creating and Using a `CommissionEmployee` Class (cont.)

`CommissionEmployee` Member Functions `earnings` and `toString`

- Member function `earnings` calculates a `CommissionEmployee`'s earnings.
- Member function `toString` displays the values of a `CommissionEmployee` object's data members.

Testing Class `CommissionEmployee`

- Figure 11.6 tests class `CommissionEmployee`.

```
1  // Fig. 11.6: fig11_06.cpp
2  // CommissionEmployee class test program.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h" // CommissionEmployee class definition
6  using namespace std;
7
8  int main() {
9     // instantiate a CommissionEmployee object
10    CommissionEmployee employee{"Sue", "Jones", "222-22-2222", 10000, .06};
11
12    // get commission employee data
13    cout << fixed << setprecision(2); // set floating-point formatting
14    cout << "Employee information obtained by get functions: \n"
15       << "\nFirst name is " << employee.getFirstName()
16       << "\nLast name is " << employee.getLastName()
17       << "\nSocial security number is "
18       << employee.getSocialSecurityNumber()
19       << "\nGross sales is " << employee.getGrossSales()
20       << "\nCommission rate is " << employee.getCommissionRate() << endl;
```

Fig. 11.6 | CommissionEmployee class test program. (Part 1 of 3.)

```
21
22      employee.setGrossSales(8000); // set gross sales
23      employee.setCommissionRate(.1); // set commission rate
24      cout << "\nUpdated employee information from function toString: \n\n"
25         << employee.toString();
26
27      // display the employee's earnings
28      cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
29   }
```

Fig. 11.6 | CommissionEmployee class test program. (Part 2 of 3.)

```
Employee information obtained by get functions:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information from function toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10

Employee's earnings: $800.00
```

Fig. 11.6 | `CommissionEmployee` class test program. (Part 3 of 3.)

11.3.2 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. 11.7–11.8), which contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary.

```
1  // Fig. 11.7: 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
9  class BasePlusCommissionEmployee {
10 public:
11    BasePlusCommissionEmployee(const std::string&, const std::string&,
12       const std::string&, double = 0.0, 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. 11.7 | BasePlusCommissionEmployee class header. (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      void setBaseSalary(double); // set base salary
30      double getBaseSalary() const; // return base salary
31
32      double earnings() const; // calculate earnings
33      std::string toString() const; // create string representation
34  private:
35      std::string firstName;
36      std::string lastName;
37      std::string socialSecurityNumber;
38      double grossSales; // gross weekly sales
39      double commissionRate; // commission percentage
40      double baseSalary; // base salary
41  };
42
43  #endif
```

Fig. 11.7 | BasePlusCommissionEmployee class header. (Part 2 of 2.)

```
1  // Fig. 11.8: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iomanip>
4  #include <stdexcept>
5  #include <sstream>
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string& first, const string& last, const string& ssn,
12    double sales, double rate, double salary) {
13    firstName = first; // should validate
14    lastName = last; // should validate
15    socialSecurityNumber = ssn; // should validate
16    setGrossSales(sales); // validate and store gross sales
17    setCommissionRate(rate); // validate and store commission rate
18    setBaseSalary(salary); // validate and store base salary
19 }
20
```

Fig. 11.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 6.)

```
21   // set first name
22   void BasePlusCommissionEmployee::setFirstName(const string& first) {
23      firstName = first; // should validate
24   }
25
26   // return first name
27   string BasePlusCommissionEmployee::getFirstName() const {
28      return firstName;
29   }
30
31   // set last name
32   void BasePlusCommissionEmployee::setLastName(const string& last) {
33      lastName = last; // should validate
34   }
35
36   // return last name
37   string BasePlusCommissionEmployee::getLastName() const {return lastName;}
38
```

Fig. 11.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 6.)

```
39   // set social security number
40   void BasePlusCommissionEmployee::setSocialSecurityNumber(
41      const string& ssn) {
42      socialSecurityNumber = ssn; // should validate
43   }
44
45   // return social security number
46   string BasePlusCommissionEmployee::getSocialSecurityNumber() const {
47      return socialSecurityNumber;
48   }
49
50   // set gross sales amount
51   void BasePlusCommissionEmployee::setGrossSales(double sales) {
52      if (sales < 0.0) {
53         throw invalid_argument("Gross sales must be >= 0.0");
54      }
55
56      grossSales = sales;
57   }
```

Fig. 11.8 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 3 of 6.)

```
58
59   // return gross sales amount
60   double BasePlusCommissionEmployee::getGrossSales() const {
61      return grossSales;
62   }
63
64   // set commission rate
65   void BasePlusCommissionEmployee::setCommissionRate(double rate) {
66      if (rate <= 0.0 || rate >= 1.0) {
67         throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
68      }
69
70      commissionRate = rate;
71   }
72
73   // return commission rate
74   double BasePlusCommissionEmployee::getCommissionRate() const {
75      return commissionRate;
76   }
77
```

Fig. 11.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 6.)

```
78  // set base salary
79  void BasePlusCommissionEmployee::setBaseSalary(double salary) {
80     if (salary < 0.0) {
81        throw invalid_argument("Salary must be >= 0.0");
82     }
83
84     baseSalary = salary;
85  }
86
87  // return base salary
88  double BasePlusCommissionEmployee::getBaseSalary() const {
89     return baseSalary;
90  }
91
92  // calculate earnings
93  double BasePlusCommissionEmployee::earnings() const {
94     return baseSalary + (commissionRate * grossSales);
95  }
```

Fig. 11.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 6.)

```
96
97  // return string representation of BasePlusCommissionEmployee object
98  string BasePlusCommissionEmployee::toString() const {
99     ostringstream output;
100    output << fixed << setprecision(2); // two digits of precision
101    output << "base-salaried commission employee: " << firstName << ' '
102       << lastName << "\nsocial security number: " << socialSecurityNumber
103       << "\ngross sales: " << grossSales
104       << "\ncommission rate: " << commissionRate
105       << "\nbase salary: " << baseSalary;
106    return output.str();
107 }
```

Fig. 11.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 6 of 6.)

11.3.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance (cont.)

Defining Class BasePlusCommissionEmployee

- The `BasePlusCommissionEmployee` header (Fig. 11.7) specifies class `BasePlusCommissionEmployee`'s `public` services, which include the `BasePlusCommissionEmployee` constructor and member functions `earnings` and `toString`.
- Lines 14–30 declare `public` *get* and *set* functions for the class's `private` data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` *and* `baseSalary`.

11.3.2 Creating a `BasePlusCommissionEmployee` Class Without Using Inheritance (cont.)

- Note the similarity between this class and class `CommissionEmployee` (Figs. 11.4–11.5)—in this example, we won't yet exploit that similarity.
- Class `BasePlusCommissionEmployee`'s `earnings` member function computes the earnings of a base-salaried commission employee.

Testing Class `BasePlusCommissionEmployee`

- Figure 11.9 tests class `BasePlusCommissionEmployee`.

```
1  // Fig. 11.9: fig11_09.cpp
2  // BasePlusCommissionEmployee class test program.
3  #include <iostream>
4  #include <iomanip>
5  #include "BasePlusCommissionEmployee.h"
6  using namespace std;
7
8  int main() {
9     // instantiate BasePlusCommissionEmployee object
10    BasePlusCommissionEmployee employee{"Bob", "Lewis", "333-33-3333",
11       5000, .04, 300};
12
13    // get commission employee data
14    cout << fixed << setprecision(2); // set floating-point formatting
15    cout << "Employee information obtained by get functions: \n"
16       << "\nFirst name is " << employee.getFirstName()
17       << "\nLast name is " << employee.getLastName()
18       << "\nSocial security number is "
19       << employee.getSocialSecurityNumber()
20       << "\nGross sales is " << employee.getGrossSales()
21       << "\nCommission rate is " << employee.getCommissionRate()
22       << "\nBase salary is " << employee.getBaseSalary() << endl;
```

Fig. 11.9 | BasePlusCommissionEmployee class test program. (Part 1 of 3.)

```
23      employee.setBaseSalary(1000); // set base salary
24      cout << "\nUpdated employee information from function toString: \n\n"
25         << employee.toString();
26
27      // display the employee's earnings
28      cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
29   }
```

Fig. 11.9 | BasePlusCommissionEmployee class test program. (Part 2 of 3.)

```
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 from function toString:

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. 11.9 | `BasePlusCommissionEmployee` class test program. (Part 3 of 3.)

11.3.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance (cont.)

Exploring the Similarities Between Class BasePlusCommissionEmployee and Class CommissionEmployee

- Most of the code for class `BasePlusCommissionEmployee` (Figs. 11.7–11.8) is similar, if not identical, to the code for class `CommissionEmployee` (Figs. 11.4–11.5).
- In class `BasePlusCommissionEmployee`, `private` data members `firstName` and `lastName` and member functions `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`.
- Both classes contain `private` data members `socialSecurityNumber`, `commissionRate` and `grossSales`, as well as *get* and *set* functions to manipulate these members.

11.3.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance (cont.)

- 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 `toString` member function is *nearly identical* to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s `toString` also outputs the value of data member `baseSalary`.

11.3.2 Creating a `BasePlusCommissionEmployee` Class Without Using Inheritance (cont.)

- 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 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.

Software Engineering Observation 11.1

Copying and pasting code from one class to another can spread many physical copies of the same code and can spread errors throughout a system, creating a code-maintenance nightmare. 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 11.2

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, you 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.*

11.3.3 Creating a `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy

- Now we create and test a new `BasePlusCommissionEmployee` class (Figs. 11.10–11.11) that derives from `CommissionEmployee` (Figs. 11.4–11.5).
- In this example, a `BasePlusCommissionEmployee` object *is a* `CommissionEmployee` (because inheritance passes on the capabilities of class `CommissionEmployee`), but class `BasePlusCommissionEmployee` also has data member `baseSalary` (Fig. 11.10, line 21).
- The colon (`:`) in line 10 of the class definition indicates inheritance.
- Keyword `public` indicates the *type of inheritance*.
- As a derived class (formed with `public` inheritance), `BasePlusCommissionEmployee` inherits all the members of class `CommissionEmployee`, except for the constructor—each class provides its own constructors that are specific to the class.

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

- Destructors, too, are not inherited
- Thus, the `public` services of `BasePlusCommissionEmployee` include its constructor and the `public` member functions inherited from class *`CommissionEmployee`—although we cannot see these inherited member functions* in `BasePlusCommissionEmployee`'s source code, they're nevertheless a part of derived class `BasePlusCommissionEmployee`.
- The derived class's `public` services also include member functions `setBaseSalary`, `getBaseSalary`, `earnings` and `toString`.

```
1  // Fig. 11.10: 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  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9
10 class BasePlusCommissionEmployee : public CommissionEmployee {
11 public:
12    BasePlusCommissionEmployee(const std::string&, const std::string&,
13       const std::string&, double = 0.0, double = 0.0, double = 0.0);
14
15    void setBaseSalary(double); // set base salary
16    double getBaseSalary() const; // return base salary
```

Fig. 11.10 | `BasePlusCommissionEmployee` class definition indicating inheritance relationship with class `CommissionEmployee`. (Part 1 of 2.)

```
17
18      double earnings() const; // calculate earnings
19      std::string toString() const; // create string representation
20   private:
21      double baseSalary; // base salary
22   };
23
24   #endif
```

Fig. 11.10 | `BasePlusCommissionEmployee` class definition indicating inheritance relationship with class `CommissionEmployee`. (Part 2 of 2.)

```
1  // Fig. 11.11: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iomanip>
4  #include <sstream>
5  #include <stdexcept>
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string& first, const string& last, const string& ssn,
12    double sales, double rate, double salary)
13    // explicitly call base-class constructor
14    : CommissionEmployee(first, last, ssn, sales, rate) {
15    setBaseSalary(salary); // validate and store base salary
16 }
17
```

Fig. 11.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 1 of 4.)

```
18  // set base salary
19  void BasePlusCommissionEmployee::setBaseSalary(double salary) {
20     if (salary < 0.0) {
21        throw invalid_argument("Salary must be >= 0.0");
22     }
23
24     baseSalary = salary;
25  }
26
27  // return base salary
28  double BasePlusCommissionEmployee::getBaseSalary() const {
29     return baseSalary;
30  }
31
32  // calculate earnings
33  double BasePlusCommissionEmployee::earnings() const {
34     // derived class cannot access the base class's private data
35     return baseSalary + (commissionRate * grossSales);
36  }
```

Fig. 11.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 2 of 4.)

```
37
38   // returns string representation of BasePlusCommissionEmployee object
39   string BasePlusCommissionEmployee::toString() const {
40      ostringstream output;
41      output << fixed << setprecision(2); // two digits of precision
42
43      // derived class cannot access the base class's private data
44      output << "base-salaried commission employee: " << firstName << ' '
45         << lastName << "\nsocial security number: " << socialSecurityNumber
46         << "\ngross sales: " << grossSales
47         << "\ncommission rate: " << commissionRate
48         << "\nbase salary: " << baseSalary;
49      return output.str();
50   }
```

Fig. 11.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 3 of 4.)

Compilation Errors from the Clang/LLVM Compiler in Xcode 7.2

```
BasePlusCommissionEmployee.cpp:34:25:
   'commissionRate' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:34:42:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:42:55:
   'firstName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:10:
   'lastName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:54:
   'socialSecurityNumber' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:44:31:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:45:35:
   'commissionRate' is a private member of 'CommissionEmployee'
```

Fig. 11.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 4 of 4.)

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

- Figure 11.11 shows `BasePlusCommissionEmployee`'s member-function implementations.
- The constructor introduces base-class initializer syntax, which uses a member initializer to pass arguments to the base-class constructor.
- C++ requires that a derived-class constructor call its base-class constructor to initialize the base-class data members that are inherited into the derived class.
- 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.

Common Programming Error 11.1

When a derived-class constructor calls a base-class constructor, the arguments passed to the base-class constructor must be consistent with the number and types of parameters specified in one of the base-class constructors; otherwise, a compilation error occurs.

Performance Tip 11.1

In a derived-class constructor, invoking base-class constructors and initializing member objects 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.

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

Compilation Errors from Accessing Base-Class `private` Members

- The compiler generates errors for line 35 of Fig. 11.11 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.
- We used red text in Fig. 11.11 to indicate erroneous code.
- The compiler issues additional errors in lines 44–47 of `BasePlus-Commission-Employee`'s `toString` member function for the same reason.
- 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.*

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

Preventing the Errors in `BasePlusCommissionEmployee`

- We purposely included the erroneous code in Fig. 11.11 to emphasize 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`.

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, 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.
  - 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 9.3, we discussed the linking process for creating an executable `Time` 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.

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 30–35) 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  public:
10    CommissionEmployee(const std::string&, const std::string&,
11       const std::string&, double = 0.0, double = 0.0);
12
13    void setFirstName(const std::string&); // set first name
14    std::string getFirstName() const; // return first name
15
16    void setLastName(const std::string&); // set last name
17    std::string getLastName() const; // return last name
18
19    void setSocialSecurityNumber(const std::string&); // set SSN
20    std::string getSocialSecurityNumber() const; // return SSN
21
```

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

```
22      void setGrossSales(double); // set gross sales amount
23      double getGrossSales() const; // return gross sales amount
24
25      void setCommissionRate(double); // set commission rate
26      double getCommissionRate() const; // return commission rate
27
28      double earnings() const; // calculate earnings
29      std::string toString() const; // return string representation
30   protected:
31      std::string firstName;
32      std::string lastName;
33      std::string socialSecurityNumber;
34      double grossSales; // gross weekly sales
35      double commissionRate; // commission percentage
36   };
37
38   #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 `toString` 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` is considerably shorter than the code for the noninherited version of the class, 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 from function toString:

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.*

11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

- We now reexamine our hierarchy once more, this time using the best software engineering practices.
- Class `CommissionEmployee` now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `private` (as shown previously in lines 31–35 of Fig. 11.4).

11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data (cont.)

Changes to Class CommissionEmployee's Member Function Definitions

- In the `CommissionEmployee` constructor implementation (Fig. 11.14, lines 10–15), we use member initializers (line 12) to set the values of members `firstName`, `lastName` and `socialSecurityNumber`.
- Though we do not do so here, derived-class `BasePlusCommissionEmployee` (Fig. 11.15) can invoke non-`private` base-class member functions (`setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber` and `getSocialSecurityNumber`) to manipulate these data members, , as can any client code of class `BasePlusCommissionEmployee` (such as `main`).

```
1  // Fig. 11.14: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iomanip>
4  #include <stdexcept>
5  #include <sstream>
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7  using namespace std;
8
9  // constructor
10 CommissionEmployee::CommissionEmployee(const string &first,
11    const string &last, const string &ssn, double sales, double rate)
12    : firstName(first), lastName(last), socialSecurityNumber(ssn) {
13    setGrossSales(sales); // validate and store gross sales
14    setCommissionRate(rate); // validate and store commission rate
15 }
16
17 // set first name
18 void CommissionEmployee::setFirstName(const string& first) {
19    firstName = first; // should validate
20 }
```

Fig. 11.14 | CommissionEmployee class implementation file: CommissionEmployee class uses member functions to manipulate its private data. (Part 1 of 5.)

```
21
22   // return first name
23   string CommissionEmployee::getFirstName() const {return firstName;}
24
25   // set last name
26   void CommissionEmployee::setLastName(const string& last) {
27      lastName = last; // should validate
28   }
29
30   // return last name
31   string CommissionEmployee::getLastName() const {return lastName;}
32
```

Fig. 11.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 2 of 5.)

```
33  // set social security number
34  void CommissionEmployee::setSocialSecurityNumber(const string& ssn) {
35     socialSecurityNumber = ssn; // should validate
36  }
37
38  // return social security number
39  string CommissionEmployee::getSocialSecurityNumber() const {
40     return socialSecurityNumber;
41  }
42
43  // set gross sales amount
44  void CommissionEmployee::setGrossSales(double sales) {
45     if (sales < 0.0) {
46        throw invalid_argument("Gross sales must be >= 0.0");
47     }
48
49     grossSales = sales;
50  }
51
52  // return gross sales amount
53  double CommissionEmployee::getGrossSales() const {return grossSales;}
```

Fig. 11.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 3 of 5.)

```
54
55   // set commission rate
56   void CommissionEmployee::setCommissionRate(double rate) {
57      if (rate <= 0.0 || rate >= 1.0) {
58         throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
59      }
60
61      commissionRate = rate;
62   }
63
64   // return commission rate
65   double CommissionEmployee::getCommissionRate() const {
66      return commissionRate;
67   }
68
69   // calculate earnings
70   double CommissionEmployee::earnings() const {
71      return getCommissionRate() * getGrossSales();
72   }
73
```

Fig. 11.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 4 of 5.)

```
74   // return string representation of CommissionEmployee object
75   string CommissionEmployee::toString() const {
76      ostringstream output;
77      output << fixed << setprecision(2); // two digits of precision
78      output << "commission employee: "
79         << getFirstName() << ' ' << getLastName()
80         << "\nsocial security number: " << getSocialSecurityNumber()
81         << "\ngross sales: " << getGrossSales()
82         << "\ncommission rate: " << getCommissionRate();
83      return output.str();
84   }
```

Fig. 11.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 5 of 5.)

Performance Tip 11.2

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 perform many optimizations implicitly (such as inlining set *and* get *member-function calls). You should write code that adheres to proper software engineering principles, and leave optimization to the compiler. A good rule is, "Do not second-guess the compiler."*

11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data (cont.)

Changes to Class BasePlusCommissionEmployee's Member Function Definitions

- Class `BasePlusCommissionEmployee` has several changes to its member-function implementations (Fig. 11.15) that distinguish it from the previous version of the class (Figs. 11.10–11.11).
- Member functions `earnings` (Fig. 11.15, lines 32–34) and `toString` (lines 37–42) each invoke `getBaseSalary` to obtain the base salary value.

```
1  // Fig. 11.15: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <stdexcept>
4  #include <sstream>
5  #include "BasePlusCommissionEmployee.h"
6  using namespace std;
7
8  // constructor
9  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
10    const string& first, const string& last, const string& ssn,
11    double sales, double rate, double salary)
12    // explicitly call base-class constructor
13    : CommissionEmployee(first, last, ssn, sales, rate) {
14    setBaseSalary(salary); // validate and store base salary
15 }
16
```

Fig. 11.15 | `BasePlusCommissionEmployee` class that inherits from class `CommissionEmployee` but cannot directly access the class's `private` data. (Part 1 of 3.)

```
17   // set base salary
18   void BasePlusCommissionEmployee::setBaseSalary(double salary) {
19      if (salary < 0.0) {
20         throw invalid_argument("Salary must be >= 0.0");
21      }
22
23      baseSalary = salary;
24   }
25
26   // return base salary
27   double BasePlusCommissionEmployee::getBaseSalary() const {
28      return baseSalary;
29   }
30
```

Fig. 11.15 | `BasePlusCommissionEmployee` class that inherits from class `CommissionEmployee` but cannot directly access the class's `private` data. (Part 2 of 3.)

```
31   // calculate earnings
32   double BasePlusCommissionEmployee::earnings() const {
33      return getBaseSalary() + CommissionEmployee::earnings();
34   }
35
36   // return string representation of BasePlusCommissionEmployee object
37   string BasePlusCommissionEmployee::toString() const {
38      ostringstream output;
39      output << "base-salaried " << CommissionEmployee::toString()
40         << "\nbase salary: " << getBaseSalary();
41      return output.str();
42   }
```

Fig. 11.15 | `BasePlusCommissionEmployee` class that inherits from class `CommissionEmployee` but cannot directly access the class's `private` data. (Part 3 of 3.)

11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data (cont.)

BasePlusCommissionEmployee* Member Function *earnings

- Class `BasePlusCommissionEmployee`'s `earnings` function (Fig. 11.15, lines 32–34) redefines class `CommissionEmployee`'s `earnings` member function (Fig. 11.14, lines 70–72) to calculate the earnings of a base-salaried commission employee. It also calls `CommissionEmployee`'s `earnings` function.
  - 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.
  - Good software engineering practice: 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.

Common Programming Error 11.2

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.*

11.3.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Data (cont.)

BasePlusCommissionEmployee Member Function *toString*

- `BasePlusCommissionEmployee`'s `toString` function (Fig. 11.15, lines 37–42) redefines class `CommissionEmployee`'s `toString` function (Fig. 11.14, lines 75–84) to output the appropriate base-salaried commission employee information.
- By using inheritance and by calling member functions that hide the data and ensure consistency, we've efficiently and effectively constructed a well-engineered class.

11.4 Constructors and Destructors in Derived Classes

- 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).
- 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 most derived-class constructor's body finishes executing *last*.
- Each base-class constructor initializes the base-class data members that the derived-class object inherits.

Software Engineering Observation 11.6

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 it contains more than two levels.

11.4 Constructors and Destructors in Derived Classes (cont.)

- 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 11.7

Suppose that we create an object of a derived class where both the base class and the derived class contain (via composition) 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 body executes, then the constructors for the derived class's member objects execute, then the derived class's constructor body executes. Destructors for derived-class objects are called in the reverse of the order in which their corresponding constructors are called.

11.4 Constructors and Destructors in Derived Classes (cont.)

- Base-class constructors, destructors and overloaded assignment operators (Chapter 10) are *not* inherited by derived classes.
- Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

11.4 Constructors and Destructors in Derived Classes (cont.)

C++11: Inheriting Base Class Constructors

- Sometimes a derived class's constructors simply specify the same parameters as the base class's constructors.
- For such cases, C++11 allows you to specify that a derived class should *inherit* a base class's constructors.
- To do this explicitly include a using declaration of the following form *anywhere* in the derived-class definition:

```
using BaseClass::BaseClass;
```

- In the preceding declaration, `BaseClass` is the base class's name.

11.4 Constructors and Destructors in Derived Classes (cont.)

- When you inherit constructors:
  - Each inherited constructor has the *same* access specifier (`public`, `protected` or `private`) as its corresponding base-class constructor.
  - The default, copy and move constructors are *not* inherited.
  - If a constructor is *deleted* in the base class by placing `= delete` in its prototype, the corresponding constructor in the derived class is *also* deleted.
  - If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.
  - A given base-class constructor is not inherited if a constructor that you explicitly define in the derived class has the same parameter list.
  - A base-class constructor's default arguments are *not* inherited. Instead, the compiler generates overloaded constructors in the derived class.

11.5 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.
- Figure 11.16 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.
- 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.

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Fig. 11.16 | Summary of base-class member accessibility in a derived class.