Object-Oriented Programming: Polymorphism II

OBJECTIVES

In this chapter you'll learn:

- How polymorphism makes programming more convenient and systems more extensible.
- The distinction between abstract and concrete classes and how to create abstract classes.
- To use runtime type information (RTTI).
- How C++ implements `virtual` functions and dynamic binding.
- How `virtual` destructors ensure that all appropriate destructors run on an object.

4 Type Fields and `switch` Statements

- One way to determine the type of an object is to use a `switch` statement to check the value of a field in the object.
- This allows us to distinguish among object types, then invoke an appropriate action for a particular object.
- Using `switch` logic exposes programs to a variety of potential problems.
  - For example, you might forget to include a type test when one is warranted, or might forget to test all possible cases in a `switch` statement.
  - When modifying a `switch`-based system by adding new types, you might forget to insert the new cases in *all* relevant `switch` statements.
  - Every addition or deletion of a class requires the modification of every `switch` statement in the system; tracking these statements down can be time consuming and error prone.

Software Engineering Observation 12.7

Polymorphic programming can eliminate the need for `switch` logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with `switch` logic.

Software Engineering Observation 12.8

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and simpler sequential code.

5 Abstract Classes and Pure `virtual` Functions

- There are cases in which it's useful to define *classes from which you never intend to instantiate any objects*.
- Such classes are called **abstract classes**.
- Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as **abstract base classes**.
- These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are ***incomplete***—derived classes must define the "missing pieces."

5 Abstract Classes and Pure `virtual` Functions

- An abstract class is a base class from which other classes can inherit.
- Classes that can be used to instantiate objects are called **concrete classes**.
- Such classes define *every* member function they declare.

5 Abstract Classes and Pure `virtual` Functions

- Abstract base classes are *too generic* to define real objects; we need to be *more specific* before we can think of instantiating objects.
- For example, if someone tells you to "draw the two-dimensional shape," what shape would you draw?
- Concrete classes provide the *specifics* that make it possible to instantiate objects.

5 Abstract Classes and Pure `virtual` Functions

- An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes.
- In some cases, abstract classes constitute the top few levels of the hierarchy.
- A good example of this is the shape hierarchy in Fig. 12.3, which begins with abstract base class `Shape`.

5 Abstract Classes and Pure `virtual` Functions

Pure Virtual Functions

- A class is made abstract by declaring one or more of its `virtual` functions to be "pure." A **pure `virtual` function** is specified by placing "`= 0`" in its declaration, as in

```
virtual void draw() const = 0; // pure virtual function
```

- The "`= 0`" is a **pure specifier**.
- Pure `virtual` functions typically do *not* provide implementations, though they can.

5 Abstract Classes and Pure `virtual` Functions

- Each *concrete* derived class *must override all* base-class pure `virtual` functions with concrete implementations of those functions; otherwise the derived class is also abstract.
- The difference between a `virtual` function and a pure `virtual` function is that a `virtual` function *has* an implementation and gives the derived class the *option* of overriding the function.
- By contrast, a pure `virtual` function does *not* have an implementation and *requires* the derived class to override the function for that derived class to be concrete; otherwise the derived class remains *abstract*.
- Pure `virtual` functions are used when it does *not* make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function.

Software Engineering Observation 12.9

An abstract class defines a common public interface for the various classes in a class hierarchy. An abstract class contains one or more pure `virtual` functions that concrete derived classes must override.

Common Programming Error 12.2

Failure to override a pure `virtual` function in a derived class makes that class abstract. Attempting to instantiate an object of an abstract class causes a compilation error.

Software Engineering Observation 12.10

An abstract class has at least one pure `virtual` function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

5 Abstract Classes and Pure `virtual` Functions

- Although we *cannot* instantiate objects of an abstract base class, we *can* use the abstract base class to declare *pointers* and *references* that can refer to objects of any *concrete* classes derived from the abstract class.
- Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

6 Case Study: Payroll System Using Polymorphism

- This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored previously. We use an abstract class and polymorphism to perform payroll calculations based on the type of employee.

6 Case Study: Payroll System Using Polymorphism

- We create an enhanced employee hierarchy to solve the following problem:

  A company pays its ***employees*** weekly. The employees are of three types:

  - ***Salaried employees*** are paid a fixed weekly salary regardless of the number of hours worked
  - ***Commission employees*** are paid a percentage of their sales
  - ***Base-salary-plus-commission employees*** receive a base salary plus a percentage of their sales.

  For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically.

- We use abstract class `Employee` to represent the general concept of an employee.

6 Case Study: Payroll System Using Polymorphism

- The UML class diagram in Fig. 12.7 shows the inheritance hierarchy for our polymorphic employee payroll application.
- The abstract class name `Employee` is italicized, as per the convention of the UML.
- Abstract base class `Employee` declares the "interface" to the hierarchy—that is, the set of member functions that a program can invoke on all `Employee` objects.
- Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` data members `firstName`, `lastName` and `socialSecurityNumber` appear in abstract base class `Employee`.

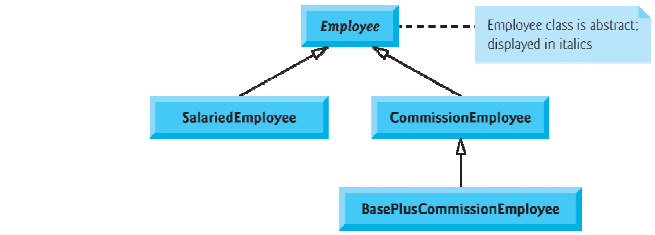


Fig. 12.7 | Employee hierarchy UML class diagram.

Software Engineering Observation 12.11

A derived class can inherit interface and/or implementation from a base class. Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy—each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions. Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy—a base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).

6.1 Creating Abstract Base Class `Employee`

- Class `Employee` (Figs. 12.9–12.10, discussed in further detail shortly) provides functions `earnings` and `print`, in addition to various *get* and *set* functions that manipulate `Employee`'s data members.
- An `earnings` function certainly applies generally to all employees, but each earnings calculation depends on the employee's class.
- So we declare `earnings` as pure `virtual` in base class `Employee` because a *default implementation does not make sense* for that function—there is not enough information to determine what amount `earnings` should return.
- Each derived class *overrides* `earnings` with an appropriate implementation.

6.1 Creating Abstract Base Class `Employee`

- To calculate an employee's earnings, the program assigns the address of an employee's object to a base class `Employee` pointer, then invokes the `earnings` function on that object.
- We maintain a `vector` of `Employee` pointers, each of which points to an `Employee` object (of course, there cannot be `Employee` objects, because `Employee` is an abstract class—because of inheritance, however, all objects of all concrete derived classes of `Employee` may nevertheless be thought of as `Employee` objects).
- The program iterates through the `vector` and calls function `earnings` for each `Employee` object.

6.1 Creating Abstract Base Class `Employee`

- C++ processes these function calls *polymorphically*.
- Including `earnings` as a pure `virtual` function in `Employee` forces every direct derived class of `Employee` that wishes to be a concrete class to override `earnings`.
- This enables the designer of the class hierarchy to demand that each derived class provide an appropriate pay calculation, if indeed that derived class is to be concrete.

6.1 Creating Abstract Base Class `Employee`

- Function `print` in class `Employee` displays the first name, last name and social security number of the employee.
- As we'll see, each derived class of `Employee` overrides function `print` to output the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information.
- Function `print` could also call `earnings`, even though `print` is a pure-`virtual` function in class `Employee`.
- The diagram in Fig. 12.8 shows each of the five classes in the hierarchy down the left side and functions `earnings` and `print` across the top.

6.1 Creating Abstract Base Class `Employee`

- For each class, the diagram shows the desired results of each function.
- Italic text represents where the values from a particular object are used in the `earnings` and `print` functions.
- Class `Employee` specifies "`= 0`" for function `earnings` to indicate that this is a pure `virtual` function.
- Each derived class overrides this function to provide an appropriate implementation.

	earnings	print
`Employee`	`= 0`	*firstName lastName* `social security number:` *SSN*
`Salaried-Employee`	*weeklySalary*	`salaried employee:` *firstName lastName* `social security number:` *SSN* `weekly salary:` *weeklySalary*
`Commission-Employee`	*commissionRate* * *grossSales*	`commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*; `commission rate:` *commissionRate*
`BasePlus-Commission-Employee`	(*commissionRate* * *grossSales*) + *baseSalary*	`base-salaried commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*; `commission rate:` *commissionRate*; `base salary:` *baseSalary*

Fig. 12.8 | Polymorphic interface for the `Employee` hierarchy classes.

6.1 Creating Abstract Base Class `Employee`

`Employee` Class Header

- Let's consider class `Employee`'s header (Fig. 12.9).
- The `public` member functions include a constructor that takes the first name, last name and social security number as arguments (lines 11-12); a virtual destructor (line 13); *set* functions that set the first name, last name and social security number (lines 15, 18 and 21, respectively); *get* functions that return the first name, last name and social security number (lines 16, 19 and 22, respectively); pure `virtual` function `earnings` (line 25) and `virtual` function `print` (line 26).

6.1 Creating Abstract Base Class `Employee`

`Employee` Class Member-Function Definitions

- Figure 12.10 contains the member-function definitions for class `Employee`.
- No implementation is provided for `virtual` function `earnings`.

```
1  // Fig. 12.9: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7
8  class Employee
9  {
10 public:
11    Employee( const std::string &, const std::string &,
12       const std::string & );
13    virtual ~Employee() { } // virtual destructor
14
15    void setFirstName( const std::string & ); // set first name
16    std::string getFirstName() const; // return first name
17
18    void setLastName( const std::string & ); // set last name
19    std::string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const std::string & ); // set SSN
22    std::string getSocialSecurityNumber() const; // return SSN
23
```

Fig. 12.9 | Employee abstract base class. (Part 1 of 2.)

```
24    // pure virtual function makes Employee an abstract base class
25    virtual double earnings() const = 0; // pure virtual
26    virtual void print() const; // virtual
27 private:
28    std::string firstName;
29    std::string lastName;
30    std::string socialSecurityNumber;
31 }; // end class Employee
32
33 #endif // EMPLOYEE_H
```

Fig. 12.9 | Employee abstract base class. (Part 2 of 2.)

```
1  // Fig. 12.10: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5  #include "Employee.h" // Employee class definition
6  using namespace std;
7
8  // constructor
9  Employee::Employee( const string &first, const string &last,
10    const string &ssn )
11    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
12 {
13    // empty body
14 } // end Employee constructor
15
16 // set first name
17 void Employee::setFirstName( const string &first )
18 {
19    firstName = first;
20 } // end function setFirstName
21
```

Fig. 12.10 | Employee class implementation file. (Part 1 of 3.)

```
22 // return first name
23 string Employee::getFirstName() const
24 {
25    return firstName;
26 } // end function getFirstName
27
28 // set last name
29 void Employee::setLastName( const string &last )
30 {
31    lastName = last;
32 } // end function setLastName
33
34 // return last name
35 string Employee::getLastName() const
36 {
37    return lastName;
38 } // end function getLastName
39
40 // set social security number
41 void Employee::setSocialSecurityNumber( const string &ssn )
42 {
43    socialSecurityNumber = ssn; // should validate
44 } // end function setSocialSecurityNumber
45
```

Fig. 12.10 | Employee class implementation file. (Part 2 of 3.)

```
46  // return social security number
47  string Employee::getSocialSecurityNumber() const
48  {
49     return socialSecurityNumber;
50  } // end function getSocialSecurityNumber
51
52  // print Employee's information (virtual, but not pure virtual)
53  void Employee::print() const
54  {
55     cout << getFirstName() << ' ' << getLastName()
56        << "\nsocial security number: " << getSocialSecurityNumber();
57  } // end function print
```

Fig. 12.10 | Employee class implementation file. (Part 3 of 3.)

6.2 Creating Concrete Derived Class `SalariedEmployee`

- Class `SalariedEmployee` (Figs. 12.11–12.12) derives from class `Employee` (line 9 of Fig. 12.11).

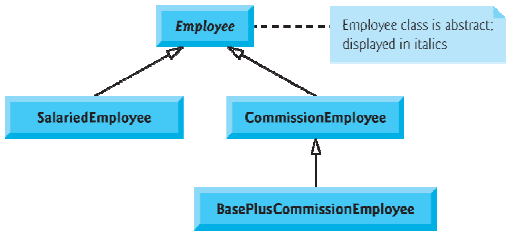


Fig. 12.7 | Employee hierarchy UML class diagram.


```
1  // Fig. 12.11: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include <string> // C++ standard string class
7  #include "Employee.h" // Employee class definition
8
9  class SalariedEmployee : public Employee
10 {
11 public:
12    SalariedEmployee( const std::string &, const std::string &,
13       const std::string &, double = 0.0 );
14    virtual ~SalariedEmployee() { } // virtual destructor
15
16    void setWeeklySalary( double ); // set weekly salary
17    double getWeeklySalary() const; // return weekly salary
18
```

Fig. 12.11 | SalariedEmployee class header. (Part 1 of 2.)

```
19     // keyword virtual signals intent to override
20     virtual double earnings() const override; // calculate earnings
21     virtual void print() const override; // print object
22  private:
23     double weeklySalary; // salary per week
24  }; // end class SalariedEmployee
25
26  #endif // SALARIED_H
```

Fig. 12.11 | SalariedEmployee class header. (Part 2 of 2.)

6.2 Creating Concrete Derived Class SalariedEmployee

SalariedEmployee Class Member-Function Definitions

- Figure 12.12 contains the member-function definitions for `SalariedEmployee`.
- The class's constructor passes the first name, last name and social security number to the `Employee` constructor (line 11) to initialize the `private` data members that are inherited from the base class, but not accessible in the derived class.
- Function `earnings` (line 33–36) overrides pure `virtual` function `earnings` in `Employee` to provide a *concrete* implementation that returns the `SalariedEmployee`'s weekly salary.

6.2 Creating Concrete Derived Class SalariedEmployee

- If we did not define `earnings`, class `SalariedEmployee` would be an *abstract* class.
- In class `SalariedEmployee`'s header, we declared member functions `earnings` and `print` as `virtual`
  - This is *redundant*.
- We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy.


```
1  // Fig. 12.12: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  #include <stdexcept>
5  #include "SalariedEmployee.h" // SalariedEmployee class definition
6  using namespace std;
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, double salary )
11    : Employee( first, last, ssn )
12 {
13    setWeeklySalary( salary );
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary( double salary )
18 {
19    if ( salary >= 0.0 )
20       weeklySalary = salary;
21    else
22       throw invalid_argument( "Weekly salary must be >= 0.0" );
23 } // end function setWeeklySalary
24
```

Fig. 12.12 | SalariedEmployee class implementation file. (Part 1 of 2.)

```
25 // return salary
26 double SalariedEmployee::getWeeklySalary() const
27 {
28    return weeklySalary;
29 } // end function getWeeklySalary
30
31 // calculate earnings;
32 // override pure virtual function earnings in Employee
33 double SalariedEmployee::earnings() const
34 {
35    return getWeeklySalary();
36 } // end function earnings
37
38 // print SalariedEmployee's information
39 void SalariedEmployee::print() const
40 {
41    cout << "salaried employee: ";
42    Employee::print(); // reuse abstract base-class print function
43    cout << "\nweekly salary: " << getWeeklySalary();
44 } // end function print
```

Fig. 12.12 | SalariedEmployee class implementation file. (Part 2 of 2.)

6.2 Creating Concrete Derived Class SalariedEmployee

- Function `print` of class `SalariedEmployee` (lines 39–44 of Fig. 12.12) overrides `Employee` function `print`.
- If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`.

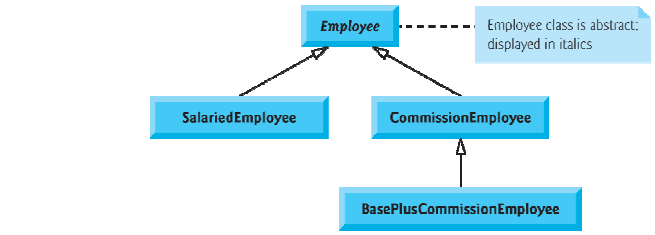


Fig. 12.7 | Employee hierarchy UML class diagram.

6.3 Creating Concrete Derived Class CommissionEmployee

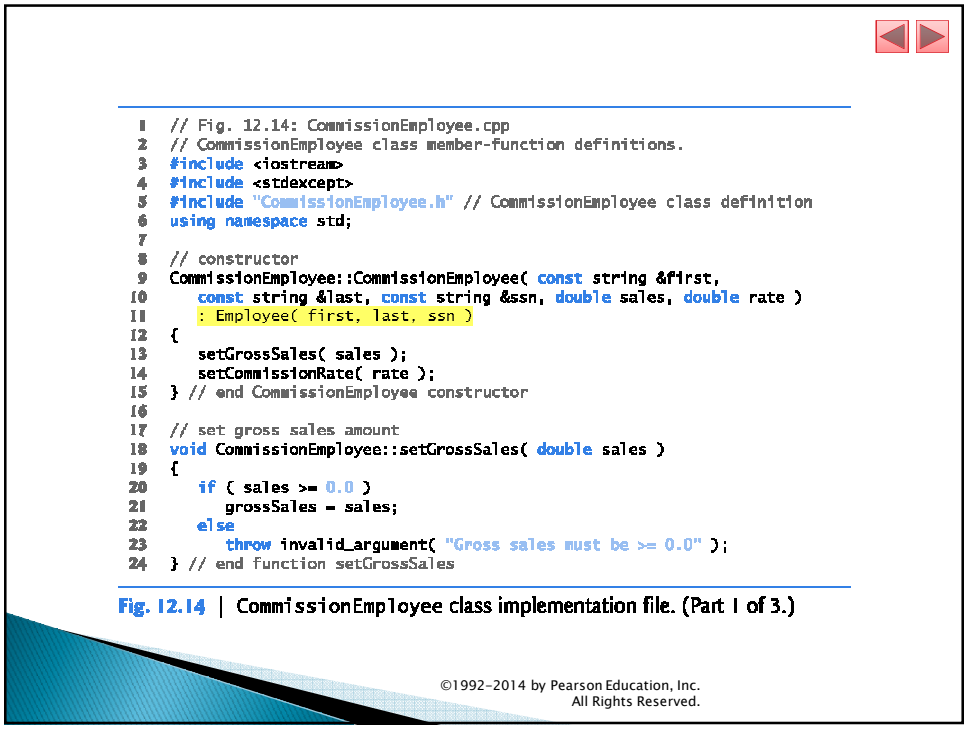
- Class `CommissionEmployee` (Figs. 12.13–12.14) derives from `Employee` (Fig. 12.13, line 9).
- The constructor passes the first name, last name and social security number to the `Employee` constructor (line 11) to initialize `Employee`'s `private` data members.
- Function `print` calls base-class function `print` (line 57) to display the `Employee`-specific information.

```
1  // Fig. 12.13: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  #include "Employee.h" // Employee class definition
8
9  class CommissionEmployee : public Employee
10 {
11 public:
12    CommissionEmployee( const std::string &, const std::string &,
13       const std::string &, double = 0.0, double = 0.0 );
14    virtual ~CommissionEmployee() { } // virtual destructor
15
16    void setCommissionRate( double ); // set commission rate
17    double getCommissionRate() const; // return commission rate
18
19    void setGrossSales( double ); // set gross sales amount
20    double getGrossSales() const; // return gross sales amount
21
```

Fig. 12.13 | CommissionEmployee class header. (Part 1 of 2.)

```
22    // keyword virtual signals intent to override
23    virtual double earnings() const override; // calculate earnings
24    virtual void print() const override; // print object
25 private:
26    double grossSales; // gross weekly sales
27    double commissionRate; // commission percentage
28 }; // end class CommissionEmployee
29
30 #endif // COMMISSION_H
```

Fig. 12.13 | CommissionEmployee class header. (Part 2 of 2.)



```
1  // Fig. 12.14: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  #include <stdexcept>
5  #include "CommissionEmployee.h" // CommissionEmployee class definition
6  using namespace std;
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, double sales, double rate )
11    : Employee( first, last, ssn )
12 {
13    setGrossSales( sales );
14    setCommissionRate( rate );
15 } // end CommissionEmployee constructor
16
17 // set gross sales amount
18 void CommissionEmployee::setGrossSales( double sales )
19 {
20    if ( sales >= 0.0 )
21       grossSales = sales;
22    else
23       throw invalid_argument( "Gross sales must be >= 0.0" );
24 } // end function setGrossSales
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 1 of 3.)

```
25
26 // return gross sales amount
27 double CommissionEmployee::getGrossSales() const
28 {
29    return grossSales;
30 } // end function getGrossSales
31
32 // set commission rate
33 void CommissionEmployee::setCommissionRate( double rate )
34 {
35    if ( rate > 0.0 && rate < 1.0 )
36       commissionRate = rate;
37    else
38       throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
39 } // end function setCommissionRate
40
41 // return commission rate
42 double CommissionEmployee::getCommissionRate() const
43 {
44    return commissionRate;
45 } // end function getCommissionRate
46
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 2 of 3.)

```
47 // calculate earnings; override pure virtual function earnings in Employee
48 double CommissionEmployee::earnings() const
49 {
50    return getCommissionRate() * getGrossSales();
51 } // end function earnings
52
53 // print CommissionEmployee's information
54 void CommissionEmployee::print() const
55 {
56    cout << "commission employee: ";
57    Employee::print(); // code reuse
58    cout << "\ngross sales: " << getGrossSales()
59       << "; commission rate: " << getCommissionRate();
60 } // end function print
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 3 of 3.)

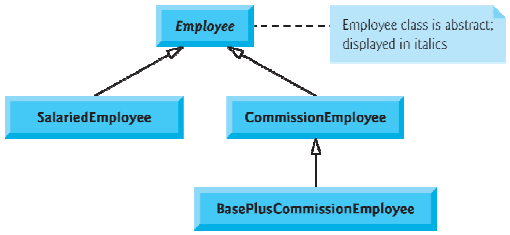


Fig. 12.7 | Employee hierarchy UML class diagram.

6.4 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

- Class `BasePlusCommissionEmployee` (Figs. 12.15–12.16) directly inherits from class `CommissionEmployee` (line 9 of Fig. 12.15) and therefore is an indirect derived class of class `Employee`.
- `BasePlusCommissionEmployee`'s `print` function (lines 40–45) outputs `"base-salaried"`, followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary.


```
1  // Fig. 12.15: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from CommissionEmployee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include <string> // C++ standard string class
7  #include "CommissionEmployee.h" // CommissionEmployee class definition
8
9  class BasePlusCommissionEmployee : public CommissionEmployee
10 {
11 public:
12    BasePlusCommissionEmployee( const std::string &, const std::string &,
13       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
14    virtual ~CommissionEmployee() { } // virtual destructor
15
16    void setBaseSalary( double ); // set base salary
17    double getBaseSalary() const; // return base salary
18
```

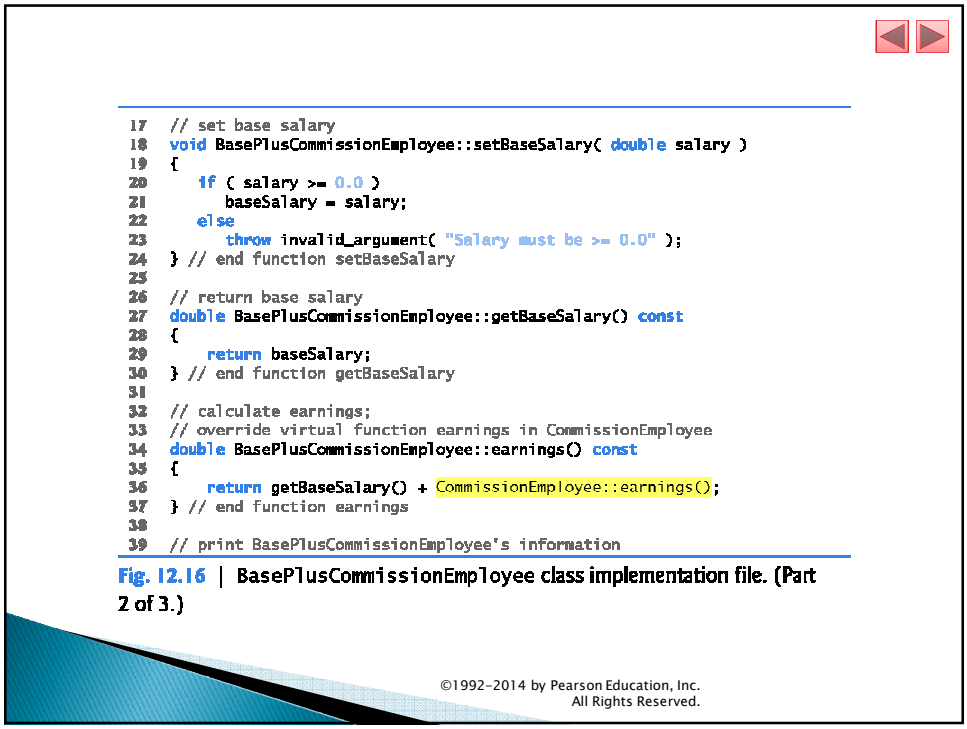
Fig. 12.15 | BasePlusCommissionEmployee class header. (Part 1 of 2.)

```
19    // keyword virtual signals intent to override
20    virtual double earnings() const override; // calculate earnings
21    virtual void print() const override; // print object
22 private:
23    double baseSalary; // base salary per week
24 }; // end class BasePlusCommissionEmployee
25
26 #endif // BASEPLUS_H
```

Fig. 12.15 | BasePlusCommissionEmployee class header. (Part 2 of 2.)

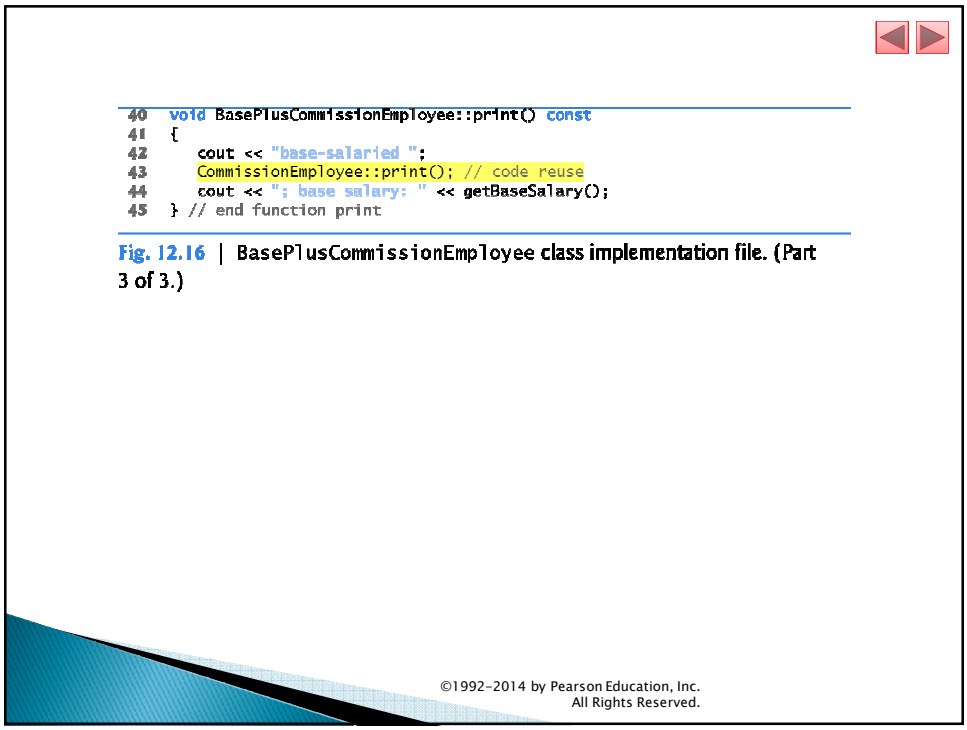

```
1  // Fig. 12.16: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  #include <stdexcept>
5  #include "BasePlusCommissionEmployee.h"
6  using namespace std;
7
8  // constructor
9  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate, double salary )
12    : CommissionEmployee( first, last, ssn, sales, rate )
13 {
14    setBaseSalary( salary ); // validate and store base salary
15 } // end BasePlusCommissionEmployee constructor
16
```

Fig. 12.16 | BasePlusCommissionEmployee class implementation file. (Part 1 of 3.)



```
17  // set base salary
18  void BasePlusCommissionEmployee::setBaseSalary( double salary )
19  {
20     if ( salary >= 0.0 )
21        baseSalary = salary;
22     else
23        throw invalid_argument( "Salary must be >= 0.0" );
24  } // end function setBaseSalary
25
26  // return base salary
27  double BasePlusCommissionEmployee::getBaseSalary() const
28  {
29     return baseSalary;
30  } // end function getBaseSalary
31
32  // calculate earnings;
33  // override virtual function earnings in CommissionEmployee
34  double BasePlusCommissionEmployee::earnings() const
35  {
36     return getBaseSalary() + CommissionEmployee::earnings();
37  } // end function earnings
38
39  // print BasePlusCommissionEmployee's information
```

Fig. 12.16 | BasePlusCommissionEmployee class implementation file. (Part 2 of 3.)



```
40  void BasePlusCommissionEmployee::print() const
41  {
42     cout << "base-salaried ";
43     CommissionEmployee::print(); // code reuse
44     cout << "; base salary: " << getBaseSalary();
45  } // end function print
```

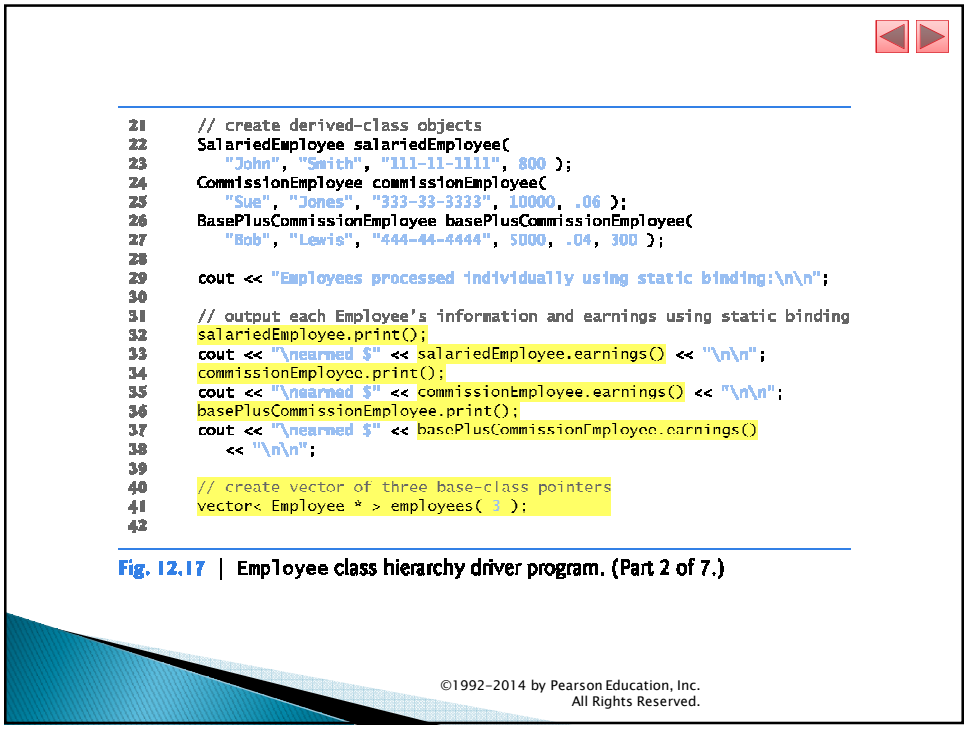
Fig. 12.16 | BasePlusCommissionEmployee class implementation file. (Part 3 of 3.)

6.5 Demonstrating Polymorphic Processing

- To test our `Employee` hierarchy, the program in Fig. 12.17 creates an object of each of the three concrete classes `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.
- The program manipulates these objects, first with static binding, then polymorphically, using a `vector` of `Employee` pointers.
- Lines 22–27 create objects of each of the three concrete `Employee` derived classes.
- Lines 32–38 output each `Employee`'s information and earnings.
- Each member-function invocation in lines 32–37 is an example of *static binding*—at *compile time*, because we are using *name handles* (not *pointers* or *references* that could be set at *execution time*), the *compiler* can identify each object's type to determine which `print` and `earnings` functions are called.

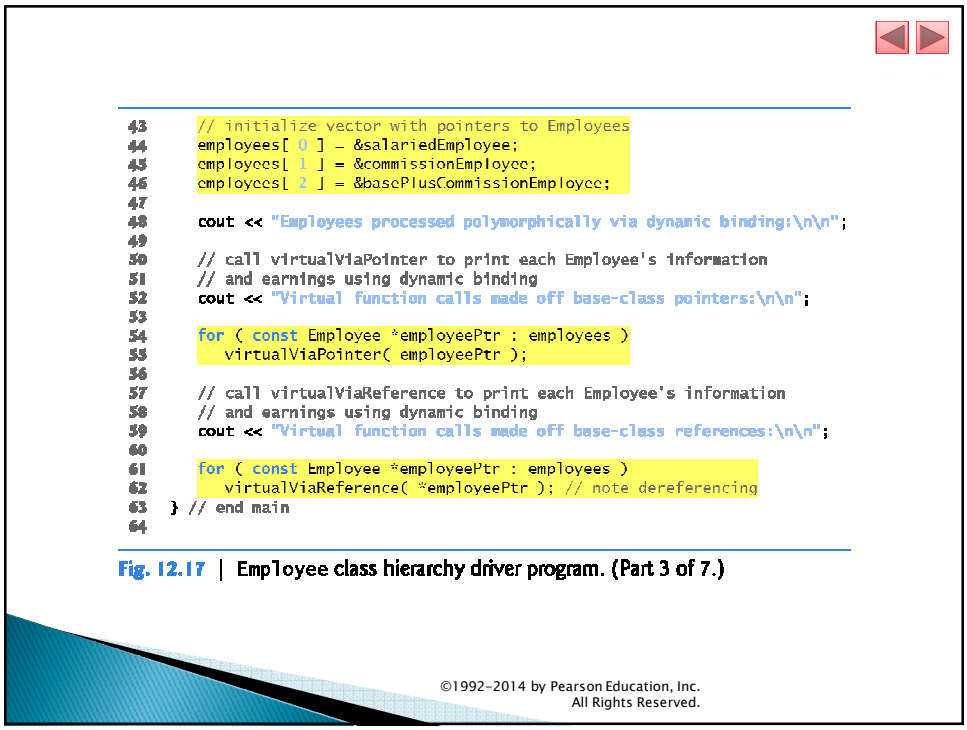
```
1  // Fig. 12.17: fig12_17.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>
7  #include "Employee.h"
8  #include "SalariedEmployee.h"
9  #include "CommissionEmployee.h"
10 #include "BasePlusCommissionEmployee.h"
11 using namespace std;
12
13 void virtualViaPointer( const Employee * const ); // prototype
14 void virtualViaReference( const Employee & ); // prototype
15
16 int main()
17 {
18    // set floating-point output formatting
19    cout << fixed << setprecision( 2 );
20
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 1 of 7.)



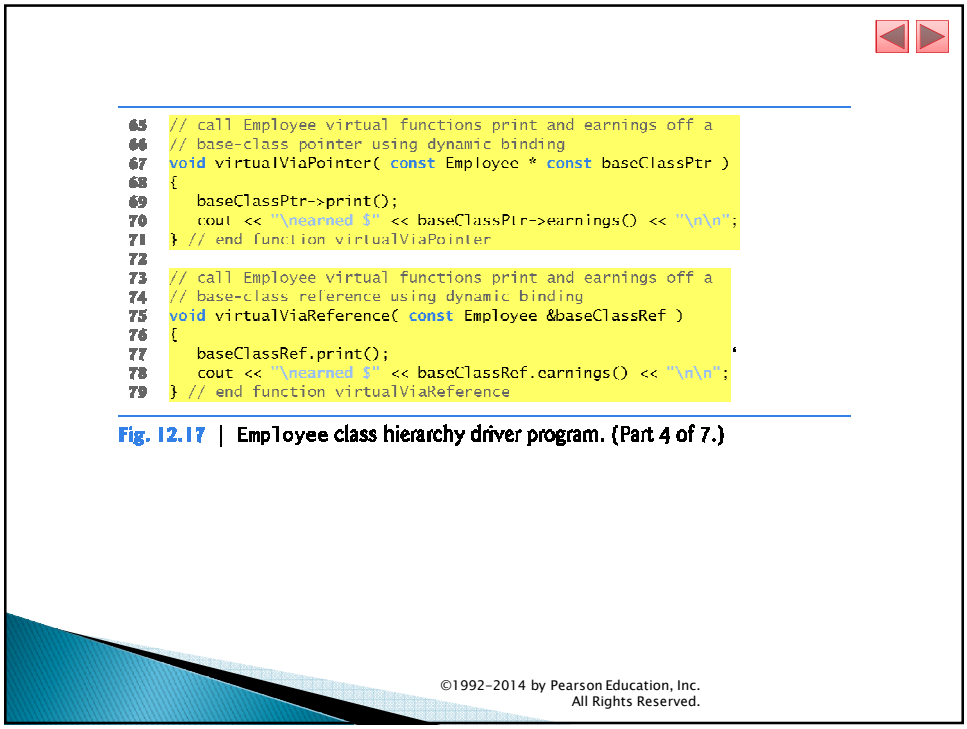
```
21      // create derived-class objects
22      SalariedEmployee salariedEmployee(
23         "John", "Smith", "111-11-1111", 800 );
24      CommissionEmployee commissionEmployee(
25         "Sue", "Jones", "333-33-3333", 10000, .06 );
26      BasePlusCommissionEmployee basePlusCommissionEmployee(
27         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
28
29      cout << "Employees processed individually using static binding:\n\n";
30
31      // output each Employee's information and earnings using static binding
32      salariedEmployee.print();
33      cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
34      commissionEmployee.print();
35      cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
36      basePlusCommissionEmployee.print();
37      cout << "\nearned $" << basePlusCommissionEmployee.earnings()
38         << "\n\n";
39
40      // create vector of three base-class pointers
41      vector< Employee * > employees( 3 );
42
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 2 of 7.)



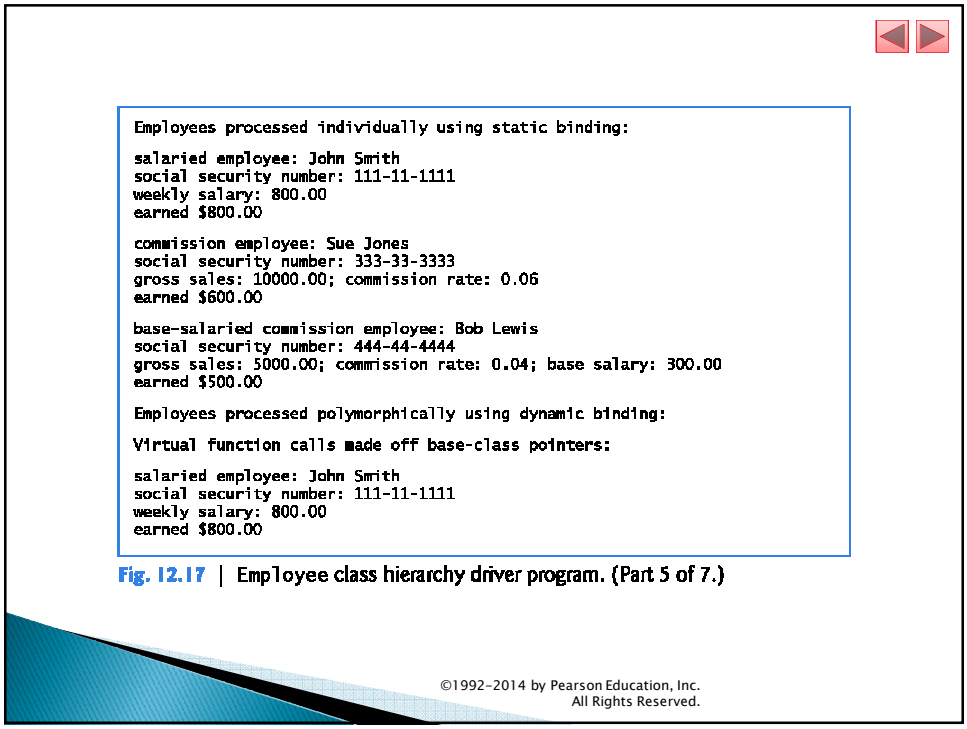
```
43      // initialize vector with pointers to Employees
44      employees[ 0 ] = &salariedEmployee;
45      employees[ 1 ] = &commissionEmployee;
46      employees[ 2 ] = &basePlusCommissionEmployee;
47
48      cout << "Employees processed polymorphically via dynamic binding:\n\n";
49
50      // call virtualViaPointer to print each Employee's information
51      // and earnings using dynamic binding
52      cout << "Virtual function calls made off base-class pointers:\n\n";
53
54      for ( const Employee *employeePtr : employees )
55         virtualViaPointer( employeePtr );
56
57      // call virtualViaReference to print each Employee's information
58      // and earnings using dynamic binding
59      cout << "Virtual function calls made off base-class references:\n\n";
60
61      for ( const Employee *employeePtr : employees )
62         virtualViaReference( *employeePtr ); // note dereferencing
63   } // end main
64
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 3 of 7.)



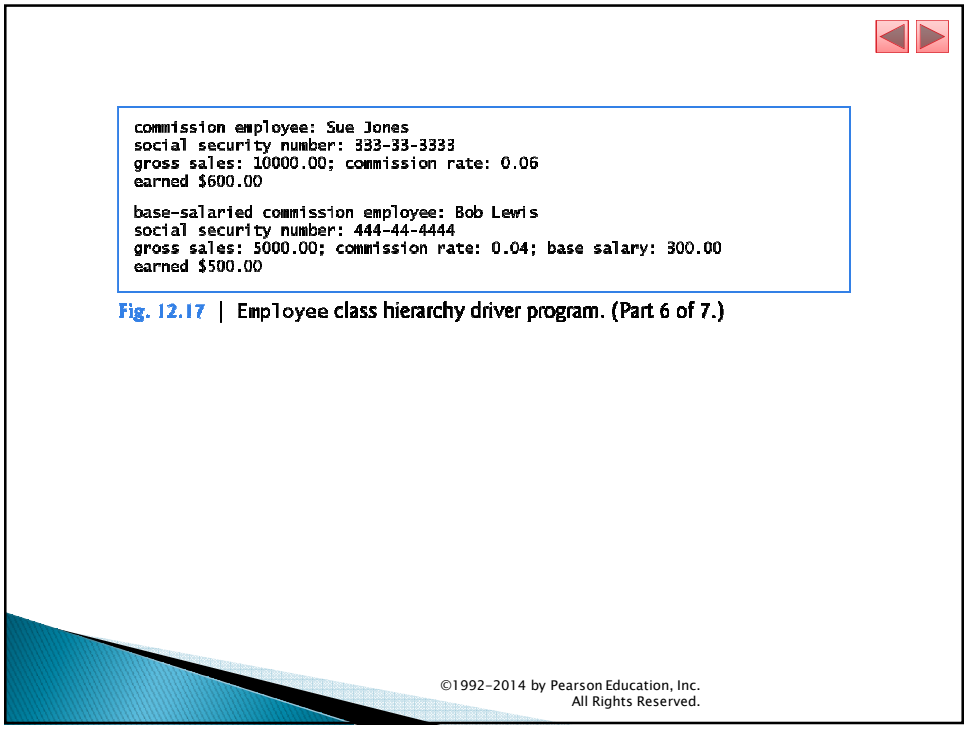
```
65 // call Employee virtual functions print and earnings off a
66 // base-class pointer using dynamic binding
67 void virtualViaPointer( const Employee * const baseClassPtr )
68 {
69    baseClassPtr->print();
70    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
71 } // end function virtualViaPointer
72
73 // call Employee virtual functions print and earnings off a
74 // base-class reference using dynamic binding
75 void virtualViaReference( const Employee &baseClassRef )
76 {
77    baseClassRef.print();
78    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
79 } // end function virtualViaReference
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 4 of 7.)



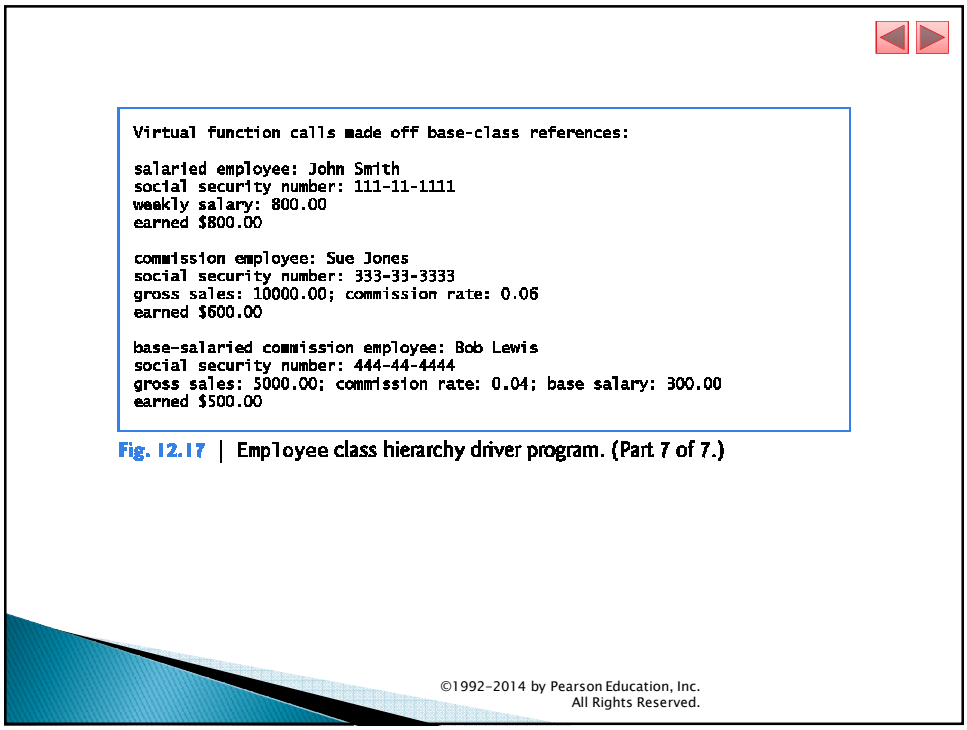
```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 5 of 7.)



```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 6 of 7.)



```
Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 12.17 | Employee class hierarchy driver program. (Part 7 of 7.)

6.5 Demonstrating Polymorphic Processing

- Line 41 creates the `vector` `employees`, which contains three `Employee` pointers.
- Line 44 aims `employees[ 0 ]` at object `salariedEmployee`.
- Line 45 aims `employees[ 1 ]` at object `commissionEmployee`.
- Line 46 aims `employees[ 2 ]` at object `basePlusCommissionEmployee`.
- The compiler allows these assignments, because a `SalariedEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`.

6.5 Demonstrating Polymorphic Processing

- Lines 54–55 traverse `vector` `employees` and invoke function `virtualViaPointer` (lines 67–71) for each element in `employees`.
- Function `virtualViaPointer` receives in parameter `baseClassPtr` (of type `const` `Employee` `*` `const`) the address stored in an `employees` element.
- Each call to `virtualViaPointer` uses `baseClassPtr` to invoke `virtual` functions `print` (line 69) and `earnings` (line 70).
- Note that function `virtualViaPointer` does not contain any `SalariedEmployee`, `CommissionEmployee` or `BasePlusCommissionEmployee` type information.
- The function knows only about base-class type `Employee`.
- The output illustrates that the appropriate functions for each class are indeed invoked and that each object's proper information is displayed.

6.5 Demonstrating Polymorphic Processing

- Lines 61–62 traverse `employees` and invoke function `virtualViaReference` (lines 75–79) for each `vector` element.
- Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee&`) a reference to the object obtained by dereferencing the pointer stored in each `employees` element (line 62).
- Each call to `virtualViaReference` invokes `virtual` functions `print` (line 77) and `earnings` (line 78) via `baseClassRef` to demonstrate that *polymorphic processing occurs with base-class references as well*.
- Each `virtual`-function invocation calls the function on the object to which `baseClassRef` refers at runtime.
- This is another example of ***dynamic binding***.
- The output produced using base-class references is identical to the output produced using base-class pointers.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- This section discusses how C++ can implement polymorphism, `virtual` functions and dynamic binding internally.
- This will give you a solid understanding of how these capabilities really work.
- More importantly, it will help you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time.
- You'll see that polymorphism is accomplished through three levels of pointers (i.e., "triple indirection").
- Then we'll show how an executing program uses these data structures to execute `virtual` functions and achieve the dynamic binding associated with polymorphism.
- Our discussion explains one possible implementation; this is not a language requirement.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- When C++ compiles a class that has one or more `virtual` functions, it builds a virtual function table (*vtable*) for that class.
- The *vtable* contains pointers to the class's `virtual` functions.
- Just as the name of a built-in array contains the address in memory of the array's first element, a pointer to a function contains the starting address in memory of the code that performs the function's task.
- An executing program uses the *vtable* to select the proper function implementation each time a `virtual` function of that class is called.
- The leftmost column of Fig. 12.18 illustrates the *vtables* for classes `Employee`, `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

***Employee Class* vtable**

- In the `Employee` class *vtable*, the first function pointer is set to 0 (i.e., the `nullptr`), because function `earnings` is a *pure* `virtual` function and therefore lacks an implementation.
- The second function pointer points to function `print`, which displays the employee's full name and social security number.
- Any class that has one or more null pointers in its *vtable* is an *abstract* class.
- Classes without any null *vtable* pointers are concrete classes.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

SalariedEmployeeClass vtable

- Class `SalariedEmployee` overrides function `earnings` to return the employee's weekly salary, so the function pointer points to the `earnings` function of class `SalariedEmployee`.
- `SalariedEmployee` also overrides `print`, so the corresponding function pointer points to the `SalariedEmployee` member function that prints `"salaried employee: "` followed by the employee's name, social security number and weekly salary.

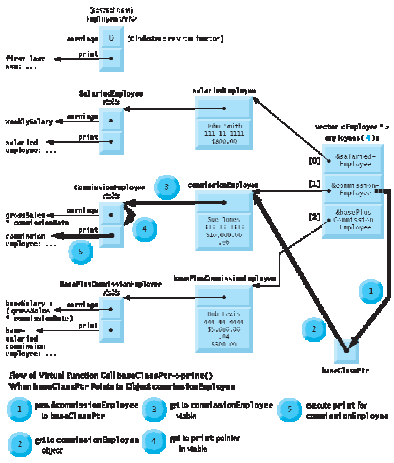


Fig. 12.18 | How `virtual` function calls work.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

***CommissionEmployee*Class vtable**

- The `earnings` function pointer in the *vtable* for class `CommissionEmployee` points to `CommissionEmployee`'s `earnings` function that returns the employee's gross sales multiplied by the commission rate.
- The `print` function pointer points to the `CommissionEmployee` version of the function, which prints the employee's type, name, social security number, commission rate and gross sales.
- As in class `SalariedEmployee`, both functions override the functions in class `Employee`.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

***BasePlusCommissionEmployee*Class vtable**

- The `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee` points to the `BasePlusCommissionEmployee`'s `earnings` function, which returns the employee's base salary plus gross sales multiplied by commission rate.
- The `print` function pointer points to the `BasePlusCommissionEmployee` version of the function, which prints the employee's base salary plus the type, name, social security number, commission rate and gross sales.
- Both functions override the functions in class `CommissionEmployee`.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

Three Levels of Pointers to Implement Polymorphism

- Polymorphism is accomplished through an elegant data structure involving three levels of pointers.
- We've discussed one level—the function pointers in the *vtable*.
- These point to the actual functions that execute when a `virtual` function is invoked.
- Now we consider the second level of pointers.
- *Whenever an object of a class with one or more* `virtual` *functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class.*
- This pointer is normally at the front of the object, but it isn't required to be implemented that way.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- In Fig. 12.18, these pointers are associated with the objects created in Fig. 12.17.
- Notice that the diagram displays each of the object's data member values.
- The third level of pointers simply contains the handles to the objects that receive the `virtual` function calls.
- The handles in this level may also be references.
- Fig. 12.18 depicts the `vector employees` that contains `Employee` pointers.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- Now let's see how a typical `virtual` function call executes.
- Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 69 of Fig. 12.17).
- Assume that `baseClassPtr` contains `employees[1]` (i.e., the address of object `commissionEmployee` in `employees`).
- When the compiler compiles this statement, it determines that the call is indeed being made via a *base-class pointer* and that `print` is a `virtual` function.
- The compiler determines that `print` is the *second* entry in each of the *vtables*.
- To locate this entry, the compiler notes that it will need to skip the first entry.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- Thus, the compiler compiles an offset or displacement of four bytes (four bytes for each pointer on today's popular 32-bit machines, and only one pointer needs to be skipped) into the table of machine-language object-code pointers to find the code that will execute the `virtual` function call.

7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The compiler generates code that performs the following operations.
  1. Select the i^{th} entry of `employees`, and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `commissionEmployee`.
  2. *Dereference* that pointer to get to the `commissionEmployee` object.
  3. *Dereference* `commissionEmployee`'s *vtable* pointer to get to the `CommissionEmployee` *vtable*.
  4. Skip the offset of four bytes to select the `print` function pointer.
  5. *Dereference* the `print` function pointer to form the "name" of the actual function to execute, and use the function call operator `()` to execute the appropriate `print` function.

Performance Tip 12.1

Polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient. You can use these capabilities with nominal impact on performance.

Performance Tip 12.2

Virtual functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that's nearly as efficient as hand-coded `switch`-based logic. Polymorphism's overhead is acceptable for most applications. In some situations—such as real-time applications with stringent performance requirements—polymorphism's overhead may be too high.

8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

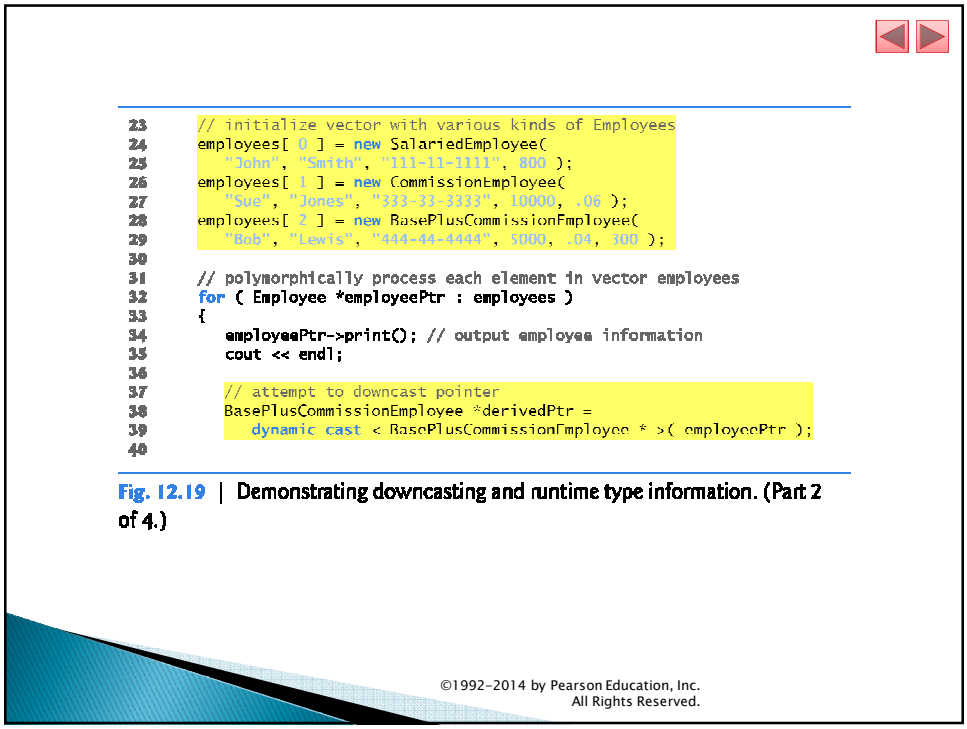
- Recall from the problem statement at the beginning of Section 6 that, for the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployee`s by adding 10 percent to their base salaries.
- When processing `Employee` objects polymorphically in Section 6.5, we did not need to worry about the "specifics."
- To adjust the base salaries of `BasePlusCommissionEmployee`s, we have to determine the specific type of each `Employee` object at execution time, then act appropriately.

8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- This section demonstrates the powerful capabilities of runtime type information (RTTI) and dynamic casting, which enable a program to determine an object's type at execution time and act on that object accordingly.
- Figure 12.19 uses the `Employee` hierarchy developed in Section 12.6 and increases by 10 percent the base salary of each `BasePlusCommissionEmployee`.

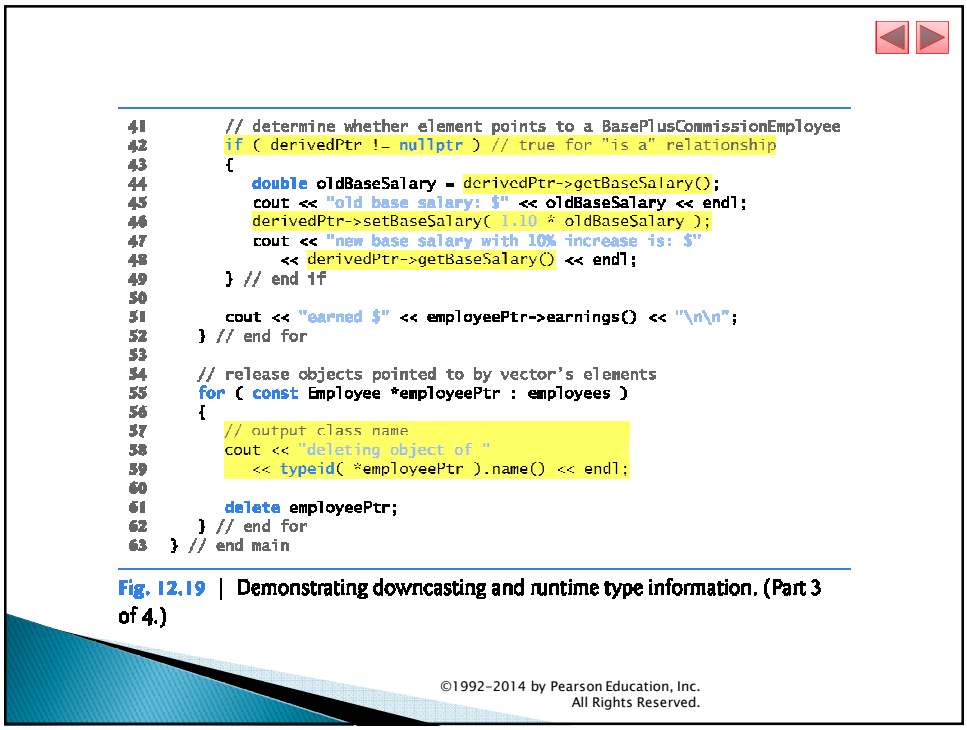

```
1  // Fig. 12.19: fig12_19.cpp
2  // Demonstrating downcasting and runtime type information.
3  // NOTE: You may need to enable RTTI on your compiler
4  // before you can compile this application.
5  #include <iostream>
6  #include <iomanip>
7  #include <vector>
8  #include <typeinfo>
9  #include "Employee.h"
10 #include "SalariedEmployee.h"
11 #include "CommissionEmployee.h"
12 #include "BasePlusCommissionEmployee.h"
13 using namespace std;
14
15 int main()
16 {
17    // set floating-point output formatting
18    cout << fixed << setprecision( 2 );
19
20    // create vector of three base-class pointers
21    vector < Employee * > employees( 3 );
22
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 1 of 4.)



```
23      // initialize vector with various kinds of Employees
24      employees[ 0 ] = new SalariedEmployee(
25         "John", "Smith", "111-11-1111", 800 );
26      employees[ 1 ] = new CommissionEmployee(
27         "Sue", "Jones", "333-33-3333", 10000, .06 );
28      employees[ 2 ] = new BasePlusCommissionEmployee(
29         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
30
31      // polymorphically process each element in vector employees
32      for ( Employee *employeePtr : employees )
33      {
34         employeePtr->print(); // output employee information
35         cout << endl;
36
37         // attempt to downcast pointer
38         BasePlusCommissionEmployee *derivedPtr =
39            dynamic_cast < BasePlusCommissionEmployee * >( employeePtr );
40
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 2 of 4.)



```
41         // determine whether element points to a BasePlusCommissionEmployee
42         if ( derivedPtr != nullptr ) // true for "is a" relationship
43         {
44            double oldBaseSalary = derivedPtr->getBaseSalary();
45            cout << "old base salary: $" << oldBaseSalary << endl;
46            derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
47            cout << "new base salary with 10% increase is: $"
48               << derivedPtr->getBaseSalary() << endl;
49         } // end if
50
51         cout << "earned $" << employeePtr->earnings() << "\n\n";
52      } // end for
53
54      // release objects pointed to by vector's elements
55      for ( const Employee *employeePtr : employees )
56      {
57         // output class name
58         cout << "deleting object of "
59            << typeid( *employeePtr ).name() << endl;
60
61         delete employeePtr;
62      } // end for
63   } // end main
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 3 of 4.)

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 4 of 4.)

12.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info` (cont.)

- Since we process the `Employee`s polymorphically, we cannot (with the techniques you've learned so far) be certain as to which type of `Employee` is being manipulated at any given time.
- `BasePlusCommissionEmployee` employees *must* be identified when we encounter them so they can receive the 10 percent salary increase.
- To accomplish this, we use operator `dynamic_cast` (line 39) to determine whether the current `Employee`'s type is `BasePlusCommissionEmployee`.
- This is the *downcast* operation we referred to in Section 12.3.3.
- Lines 38–39 dynamically downcast `employeePtr` from type `Employee *` to type `BasePlusCommissionEmployee *`.

8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info` (cont.)

- If `employeePtr` element points to an object that *is a* `BasePlusCommissionEmployee` object, then that object's address is assigned to derived-class pointer `derivedPtr`; otherwise, `nullptr` is assigned to `derivedPtr`.
- Note that `dynamic_cast` rather than `static_cast` is *required* here to perform type checking on the underlying object—a `static_cast` would simply cast the `Employee *` to a `BasePlusCommissionEmployee *` regardless of the underlying object's type.

8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info` (cont.)

- With a `static_cast`, the program would attempt to increase every `Employee`'s base salary, resulting in undefined behavior for each object that is not a `BasePlusCommissionEmployee`.
- If the value returned by the `dynamic_cast` operator in lines 38–39 *is not* `nullptr`, the object *is* the correct type, and the `if` statement (lines 42–49) performs the special processing required for the `BasePlusCommissionEmployee` object.

8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info` (cont.)

- Operator `typeid` (line 59) returns a *reference* to an object of class `type_info` that contains the information about the type of its operand, including the name of that type.
- When invoked, `type_info` member function `name` (line 59) returns a pointer-based string containing the `typeid` argument's type name (e.g., `"class BasePlusCommissionEmployee"`).
- To use `typeid`, the program must include header `<typeinfo>` (line 8).

Portability Tip 12.1

The string returned by `type_info` member function `name` may vary by compiler.