Object-Oriented Programming: Polymorphism

Based on Chapter 12 of C++ How to Program, 10/e

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

OBJECTIVES

In this chapter you'll:

- See how polymorphism makes programming more convenient and systems more extensible.
- Understand the relationships among objects in an inheritance hierarchy.
- Use C++11's **`override`** keyword when overriding a base-class virtual function in a derived class.
- Use C++11's **`default`** keyword to autogenerate a `virtual` destructor.
- Use C++11's **`final`** keyword to indicate that a base-class virtual function cannot be overridden.
- Create an inheritance hierarchy with both abstract and concrete classes.
- Determine an object's type at runtime using runtime type information (RTTI), **`dynamic_cast`**, **`typeid`** and **`type_info`**.
- Understand how C++ can implement **`virtual`** functions and dynamic binding.
- Use **`virtual`** destructors to ensure that all appropriate destructors run on an object.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.1 Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies.
- "program in the *general*" rather than "program in the *specific*."
  - Write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.
- Polymorphism works off base-class pointer handles and base-class *reference handles*, but *not* off name handles.
- Relying on each object to know how to "do the right thing" in response to the same function call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.1 Introduction (cont.)

- Design and implement systems that are easily extensible.
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generally.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.1

Polymorphism enables you to deal in generalities *and let the execution-time environment concern itself with the* specifics. *You can direct a variety of objects to behave in manners appropriate to those objects* without even knowing their types—*as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer or a common base-class reference.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.2

Polymorphism promotes extensibility: *Software written to invoke polymorphic behavior is written independently of the specific types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3 Relationships Among Objects in an Inheritance Hierarchy

- How base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects.
  - An object of a derived class can be treated as an object of its base class.
- Despite the fact that the derived-class objects are of different types, the compiler allows this because each derived-class object *is an* object of its base class.
- We cannot treat a base-class object as an any of its derived classes.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects

- Fig. 12.1 reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Section 11.3.5.
- The first two are natural and straightforward—we aim a base-class pointer at a base-class object and invoke base-class functionality, and we aim a derived-class pointer at a derived-class object and invoke derived-class functionality.
- We demonstrate the relationship between derived classes and base classes (i.e., the *is-a* relationship of inheritance) by aiming a base-class pointer at a derived-class object and showing that the base-class functionality is indeed available in the derived-class object.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.1: fig12_01.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  #include <iomanip>
6  #include "CommissionEmployee.h"
7  #include "BasePlusCommissionEmployee.h"
8  using namespace std;
9
10 int main() {
11    // create base-class object
12    CommissionEmployee commissionEmployee{
13       "Sue", "Jones", "222-22-2222", 10000, .06};
14
15    // create derived-class object
16    BasePlusCommissionEmployee basePlusCommissionEmployee{
17       "Bob", "Lewis", "333-33-3333", 5000, .04, 300};
18
19    cout << fixed << setprecision(2); // set floating-point formatting
20
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 4.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
21      // output objects commissionEmployee and basePlusCommissionEmployee
22      cout << "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:\n"
23         << commissionEmployee.toString() // base-class toString
24         << "\n\n"
25         << basePlusCommissionEmployee.toString(); // derived-class toString
26
27      // natural: aim base-class pointer at base-class object
28      CommissionEmployee* commissionEmployeePtr{&commissionEmployee};
29      cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
30         << "\nBASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:\n"
31         << commissionEmployeePtr->toString(); // base version
32
33      // natural: aim derived-class pointer at derived-class object
34      BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
35         &basePlusCommissionEmployee}; // natural
36      cout << "\n\nCALLING TOSTRING WITH DERIVED-CLASS POINTER TO "
37         << "\nDERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
38         << "TOSTRING FUNCTION:\n"
39         << basePlusCommissionEmployeePtr->toString(); // derived version
40
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 4.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
41      // aim base-class pointer at derived-class object
42      commissionEmployeePtr = &basePlusCommissionEmployee;
43      cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
44         << "DERIVED-CLASS OBJECT\nINVOKES BASE-CLASS TOSTRING "
45         << "FUNCTION ON THAT DERIVED-CLASS OBJECT:\n"
46         << commissionEmployeePtr->toString() // base version
47         << endl;
48   }
```

```
DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 4.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO
DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS OBJECT
INVOKES BASE-CLASS TOSTRING FUNCTION ON THAT DERIVED-CLASS OBJECT:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 4 of 4.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects (cont.)

Aiming a Base-Class Pointer at a Base-Class Object

- Line 28 creates base-class pointer `commissionEmployeePtr` and initializes it with the address of base-class object `commissionEmployee`
- Line 31 uses this pointer to invoke member function `toString` on that `CommissionEmployee` object.
  - Invokes the version of `toString` defined in base class `CommissionEmployee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects (cont.)

Aiming a Derived-Class Pointer at a Derived-Class Object

- Lines 34-35 create `basePlusCommissionEmployeePtr` and initialize it with the address of derived-class object `basePlusCommissionEmployee`
- Line 39 uses this pointer to invoke member function `toString` on that `BasePlusCommissionEmployee`.
  - This invokes the version of `toString` defined in derived class `BasePlusCommissionEmployee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects (cont.)

Aiming a Base-Class Pointer at a Derived-Class Object

- Line 42 assigns the address of derived-class object `basePlusCommissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 46uses to invoke `toString`.
  - "crossover" allowed because of the *is an* relationship
  - *base class* `CommissionEmployee`'s `toString` member function is invoked (rather than `BasePlusCommissionEmployee`'s `toString` function).
- *The invoked functionality depends on the type of the pointer (or reference) used to invoke the function, not the type of the object for which the member function is called.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 12.2, we attempt to aim a derived-class pointer at a base-class object.
- C++ compiler generates an error.
- The compiler prevents this assignment, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.2: fig12_02.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main() {
7     CommissionEmployee commissionEmployee{
8        "Sue", "Jones", "222-22-2222", 10000, .06};
9
10    // aim derived-class pointer at base-class object
11    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
12    BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
13       &commissionEmployee};
14 }
```

Microsoft Visual C++ compiler error message:

```
c:\examples\ch12\fig12_02\fig12_02.cpp(13):
   error C2440: 'initializing': cannot convert from 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *'
```

Fig. 12.2 | Aiming a derived-class pointer at a base-class object.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

- Off a base-class pointer, the compiler allows us to invoke *only* base-class member functions.
- If a base-class pointer is aimed at a derived-class object, and an attempt is made to access a *derived-class-only member function*, a compilation error will occur.
- Figure 12.3 shows the consequences of attempting to invoke a derived-class-only member function off a base-class pointer.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.3: fig12_03.cpp
2  // Attempting to invoke derived-class-only member functions
3  // via a base-class pointer.
4  #include <string>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main() {
10    BasePlusCommissionEmployee basePlusCommissionEmployee{
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300};
12
13    // aim base-class pointer at derived-class object (allowed)
14    CommissionEmployee* commissionEmployeePtr{&basePlusCommissionEmployee};
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer (allowed)
18    string firstName{commissionEmployeePtr->getFirstName()};
19    string lastName{commissionEmployeePtr->getLastName()};
20    string ssn{commissionEmployeePtr->getSocialSecurityNumber()};
21    double grossSales{commissionEmployeePtr->getGrossSales()};
22    double commissionRate{commissionEmployeePtr->getCommissionRate()};
```

Fig. 12.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
23
24      // attempt to invoke derived-class-only member functions
25      // on derived-class object through base-class pointer (disallowed)
26      double baseSalary{commissionEmployeePtr->getBaseSalary()};
27      commissionEmployeePtr->setBaseSalary(500);
28  }
```

GNU C++ compiler error messages:

```
fig12_03.cpp:26:45: error: 'class CommissionEmployee' has no member named
'getBaseSalary'
    double baseSalary{commissionEmployeePtr->getBaseSalary()};
                                            ^
fig12_03.cpp:27:27: error: 'class CommissionEmployee' has no member named
'setBaseSalary'
    commissionEmployeePtr->setBaseSalary(500);
```

Fig. 12.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.3.3 Derived-Class Member-Function Calls via Base-Class Pointers (cont.)

Downcasting

- The compiler will allow access to derived-class-only members from a base-class pointer that is aimed at a derived-class object *if* we explicitly cast the base-class pointer to a derived-class pointer—known as downcasting.
- Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer.
- After a downcast, the program *can* invoke derived-class functions that are not in the base class.
- Section 12.8 demonstrates how to *safely* use downcasting.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.1 Why `virtual` Functions Are Useful

- Suppose that shape classes such as `Circle`, `Triangle`, `Rectangle` and `Square` are all derived from base class `Shape`.
  - Each of these classes might be endowed with the ability to *draw itself* via a member function `draw`, but the function for each shape is quite different.
  - In a program that draws a set of shapes, it would be useful to be able to treat all the shapes generally as objects of the base class `Shape`.
  - To draw any shape, we could simply use a base-class `Shape` pointer to invoke function `draw` and let the program determine dynamically (i.e., at runtime) which derived-class `draw` function to use, based on the type of the object to which the base-class `Shape` pointer points at any given time.
  - This is *polymorphic behavior*.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.3

With `virtual` *functions, the type of the object*—not *the type of the handle used to invoke the object's member function*—*determines which version of a* `virtual` *function to invoke.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.2 Declaring `virtual` Functions

- To enable this behavior, we declare `draw` in the base class as a `virtual` function, and we override `draw` in each of the derived classes to draw the appropriate shape.
- An overridden function in a derived class has the *same signature and return type* (i.e., *prototype*) as the function it overrides in its base class.
- If we declare the base-class function as `virtual`, we can *override* that function to enable *polymorphic behavior*.
- We declare a `virtual` function by preceding the function's prototype with the key-word `virtual` in the base class.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.4

Once a function is declared `virtual`*, it remains* `virtual` *all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared* `virtual` *when a derived class overrides it.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Good Programming Practice 12.1

Even though certain functions are implicitly `virtual` *because of a declaration made higher in the class hierarchy, for clarity explicitly declare these functions* `virtual` *at every level of the class hierarchy.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.5

When a derived class chooses not to override a `virtual` *function from its base class, the derived class simply inherits its base class's* `virtual` *function implementation.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.3 Invoking a `virtual` Function Through a Base-Class Pointer or Reference

- If a program invokes a `virtual` function through a base-class pointer to a derived-class object or a base-class reference to a derived-class, the program will choose the correct derived-class function dynamically (i.e., at execution time) *based on the object type—not the pointer or reference type*.
- Known as dynamic binding or late binding.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.4 Invoking a `virtual` Function Through an Object's Name

- When a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator, the function invocation is re-solved at compile time (this is called static binding) and the `virtual` function that is called is the one defined for (or inherited by) the class of that particular object—this is *not* polymorphic behavior.
- Dynamic binding with `virtual` functions occurs only off pointers (and, as we'll soon see, references).

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.4 `virtual` Functions in the `CommissionEmployee` Hierarchy

- Figures 12.4–12.5 are the headers for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively.
- We modified these to declare each class's `earnings` and `toString` member functions as `virtual`.
- Because functions `earnings` and `toString` are `virtual` in class `CommissionEmployee`, class `BasePlusCommissionEmployee`'s `earnings` and `toString` functions *override* class `CommissionEmployee`'s.
- In addition, class `BasePlusCommissionEmployee`'s `earnings` and `toString` functions are declare `override`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 12.1

To help prevent errors, apply C++11's `override` *keyword to the prototype of* every *derived-class function that overrides a base-class* `virtual` *function. This enables the compiler to check whether the base class has a* `virtual` *member function with the same signature. If not, the compiler generates an error. Not only does this ensure that you override the base-class function with the appropriate signature, it also prevents you from accidentally hiding a base-class function that has the same name and a different signature.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class with virtual earnings and toString functions.
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. 12.4 | CommissionEmployee class header declares earnings and toString as virtual. (Part 1 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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      virtual double earnings() const; // calculate earnings
29      virtual std::string toString() const; // 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. 12.4 | CommissionEmployee class header declares earnings and toString as virtual. (Part 2 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.5: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class CommissionEmployee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include <string> // C++ standard string class
7  #include "CommissionEmployee.h" // CommissionEmployee class declaration
8
9  class BasePlusCommissionEmployee : public CommissionEmployee {
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 setBaseSalary(double); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    virtual double earnings() const override; // calculate earnings
18    virtual std::string toString() const override; // string representation
19 private:
20    double baseSalary; // base salary
21 };
22
23 #endif
```

Fig. 12.5 | `BasePlusCommissionEmployee` class header declares `earnings` and `toString` functions as `virtual` and `override`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.4 `virtual` Functions in the `CommissionEmployee` Hierarchy

- We modified Fig. 12.1 to create the program of Fig. 12.6.
- Line 47 aims base-class pointer `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`.
- When line 54 invokes member function `toString` off the base-class pointer, the derived-class `BasePlusCommissionEmployee`'s `toString` member function is invoked.
- We see that declaring a member function `virtual` causes the program to dynamically determine which function to invoke *based on the type of object to which the handle points, rather than on the type of the handle*.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.6: fig12_06.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main() {
10    // create base-class object
11    CommissionEmployee commissionEmployee{
12       "Sue", "Jones", "222-22-2222", 10000, .06};
13
14    // create derived-class object
15    BasePlusCommissionEmployee basePlusCommissionEmployee{
16       "Bob", "Lewis", "333-33-3333", 5000, .04, 300};
17
18    cout << fixed << setprecision(2); // set floating-point formatting
19
```

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 1 of 5.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
20      // output objects using static binding
21      cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND DERIVED-CLASS "
22         << "\nOBJECTS WITH STATIC BINDING\n"
23         << commissionEmployee.toString() // static binding
24         << "\n\n"
25         << basePlusCommissionEmployee.toString(); // static binding
26
27      // output objects using dynamic binding
28      cout << "\n\nINVOKING TOSTRING FUNCTION ON BASE-CLASS AND "
29         << "\nDERIVED-CLASS OBJECTS WITH DYNAMIC BINDING";
30
31      // natural: aim base-class pointer at base-class object
32      const CommissionEmployee* commissionEmployeePtr{&commissionEmployee};
33      cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
34         << "\nTO BASE-CLASS OBJECT INVOKES BASE-CLASS "
35         << "TOSTRING FUNCTION:\n"
36         << commissionEmployeePtr->toString(); // base version
37
```

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 5.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
38      // natural: aim derived-class pointer at derived-class object
39      const BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
40         &basePlusCommissionEmployee}; // natural
41      cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS "
42         << "POINTER\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
43         << "TOSTRING FUNCTION:\n"
44         << basePlusCommissionEmployeePtr->toString(); // derived version
45
46      // aim base-class pointer at derived-class object
47      commissionEmployeePtr = &basePlusCommissionEmployee;
48      cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
49         << "\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
50         << "TOSTRING FUNCTION:\n";
51
52      // polymorphism; invokes BasePlusCommissionEmployee's toString
53      // via base-class pointer to derived-class object
54      cout<< commissionEmployeePtr->toString() << endl;
55   }
```

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 3 of 5.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
INVOKING TOSTRING FUNCTION ON BASE-CLASS AND DERIVED-CLASS
OBJECTS WITH STATIC BINDING
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO BASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 4 of 5.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300
```

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 5 of 5.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.6 `virtual` Destructors

- A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy.
- If a derived-class object with a non-virtual destructor is destroyed by applying the delete operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.
- The simple solution to this problem is to create a `public virtual destructor` in the base class.
- If a base class destructor is declared `virtual`, the destructors of any derived classes are *also* `virtual`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.6 `virtual` Destructors

- In class `CommissionEmployee`'s definition, we can define the `virtual` destructor as follows:

```
virtual ~CommissionEmployee() { }
```

- Now, if an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a *base-class pointer*, the destructor for the *appropriate class* is called based on the object to which the base-class pointer points.
- Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it's important for the destructors of both the derived and base classes to execute.
- The base-class destructor automatically executes after the derived-class destructor.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 12.2

If a class has `virtual` *functions,* always *provide a* `virtual` *destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base-class pointer.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 12.1

Constructors cannot be `virtual`*. Declaring a constructor* `virtual` *is a compilation error.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.6 `virtual` Destructors

- Destructor definition also may be written as follows:
  - `virtual ~CommissionEmployee() = default;`
- You can tell the compiler to explicitly generate the default version of a default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator or destructor by following the special member function's prototype with `= default`.
  - Useful, for example, when you explicitly define a constructor for a class and still want the compiler to generate a default constructor as well—in that case, add the following declaration to your class definition:
    - `ClassName() = default;`

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.7 C++11: final Member Functions and Classes

- In C++11, a base-class virtual function that's declared final in its prototype, as in

  ```
  virtual someFunction(parameters) final;
  ```

- *cannot* be overridden in any derived class
- Guarantees that the base class's `final` member function definition will be used by all base-class objects and by all objects of the base class's direct and indirect derived classes.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.4.7 C++11: final Member Functions and Classes

- You can declare a class as final to prevent it from being used as a base class, as in

```
class MyClass final // this class cannot be a base class
{
   // class body
};
```

- Attempting to override a `final` member function or inherit from a `final` base class results in a compilation error.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.5 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.
- Can distinguish among object types, then invoke an appropriate action for a particular object, similar to polymorphism.
- Using `switch` logic exposes programs to a variety of potential problems.
  - 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 modification of every `switch` statement; tracking these down can be time consuming and error prone.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.6

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.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.7

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.6 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.
- Cannot be used to instantiate objects, because they are *incomplete*—derived classes must define the "missing pieces."
- An abstract class is a base class from which other classes can inherit.
- Classes that can be used to instantiate objects are concrete classes.
- Such classes define *every* member function they declare.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.6 Abstract Classes and Pure `virtual` Functions (cont.)

- 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.6.1 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.6.1 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.8

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.9

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.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.6.1 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7 Case Study: Payroll System Using Polymorphism

- This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section 11.3. We use an abstract class and polymorphism to perform payroll calculations based on the type of employee.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7 Case Study: Payroll System Using Polymorphism (cont.)

- 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 and 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7 Case Study: Payroll System Using Polymorphism (cont.)

- The UML class diagram in Fig. 12.7 shows the inheritance hierarchy for our polymorphic employee payroll application.
- Abstract class name `Employee` is italicized, per UML convention.
- Abstract base class `Employee` declares the "interface" to the hierarchy—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`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

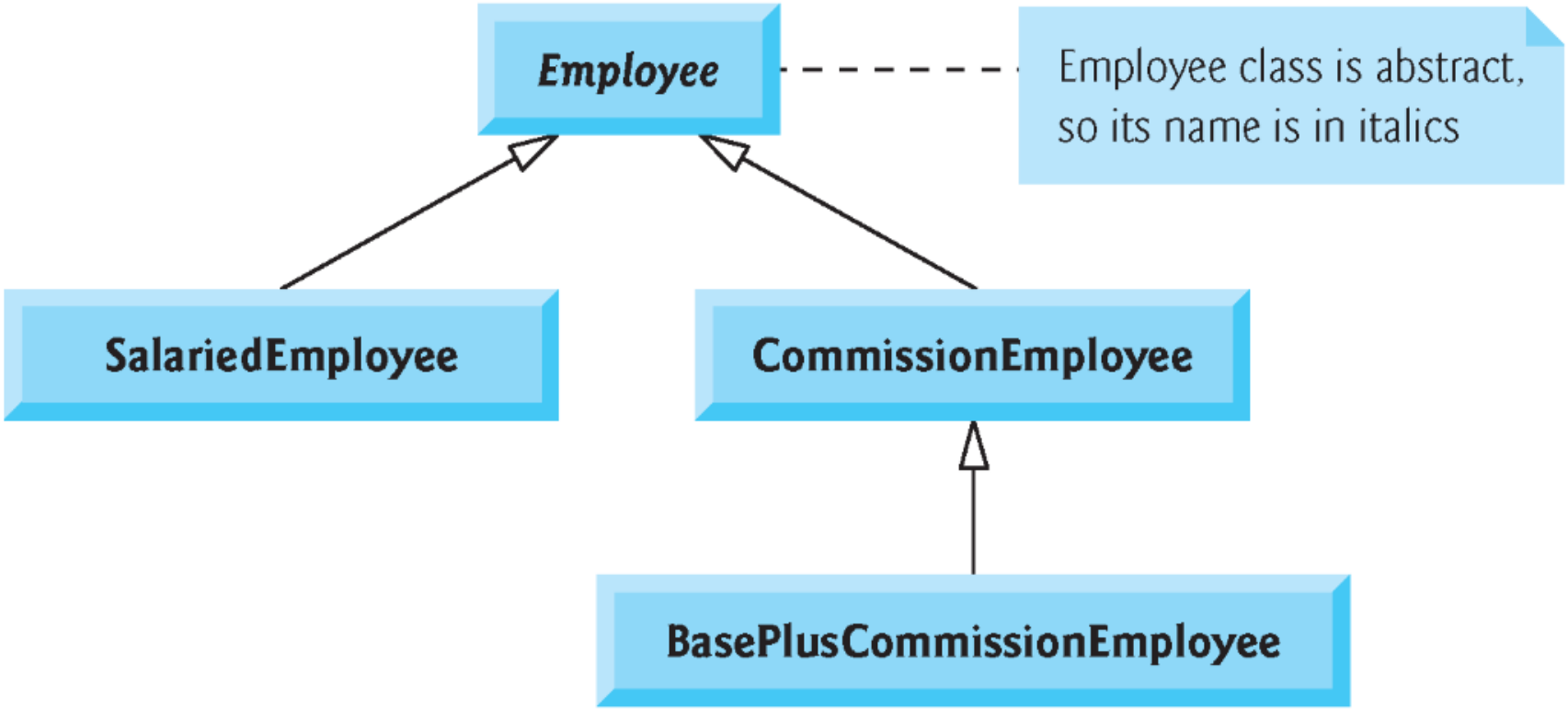

Fig. 12.7 | `Employee` hierarchy UML class diagram.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.10

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 derived class inherits one or more member functions from 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 by every derived class, but the individual derived classes provide their own implementations of the function(s).*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee`

- Class `Employee` (Figs. 12.9–12.10, discussed in further detail shortly) provides functions `earnings` and `toString`, 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee` (cont.)

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee` (cont.)

- Function `toString` in class `Employee` returns the first name, last name and social security number of the employee.
- Function `toString` could also call `earnings`, even though `toString` is a pure-`virtual` function in class `Employee`, because each concrete class is guaranteed to have an implementation of `earnings`.
- The diagram in Fig. 12.8 shows each of the five classes in the hierarchy down the left side and functions `earnings` and `toString` across the top.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee` (cont.)

- 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 `toString` 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

	earnings	toString
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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee` (cont.)

- 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
  - a `default` virtual destructor
  - *set* functions for first name, last name and social security number
  - *get* functions for first name, last name and social security number
  - pure `virtual` function `earnings`
  - `virtual` function `toString`

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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  public:
10    Employee(const std::string&, const std::string&, const std::string &);
11    virtual ~Employee() = default; // compiler generates virtual destructor
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. 12.9 | Employee abstract base class. (Part 1 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
21
22      // pure virtual function makes Employee an abstract base class
23      virtual double earnings() const = 0; // pure virtual
24      virtual std::string toString() const; // virtual
25   private:
26      std::string firstName;
27      std::string lastName;
28      std::string socialSecurityNumber;
29   };
30
31   #endif // EMPLOYEE_H
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.1 Creating Abstract Base Class `Employee` (cont.)

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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 <sstream>
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 // set first name
14 void Employee::setFirstName(const string& first) {firstName = first;}
15
16 // return first name
17 string Employee::getFirstName() const {return firstName;}
18
19 // set last name
20 void Employee::setLastName(const string& last) {lastName = last;}
21
22 // return last name
23 string Employee::getLastName() const {return lastName;}
```

Fig. 12.10 | Employee class implementation file. (Part 1 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
24
25   // set social security number
26   void Employee::setSocialSecurityNumber(const string& ssn) {
27      socialSecurityNumber = ssn; // should validate
28   }
29
30   // return social security number
31   string Employee::getSocialSecurityNumber() const {
32      return socialSecurityNumber;
33   }
34
35   // toString Employee's information (virtual, but not pure virtual)
36   string Employee::toString() const {
37      return getFirstName() + " "s + getLastName() +
38         "\nsocial security number: "s + getSocialSecurityNumber();
39   }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.2 Creating Concrete Derived Class `SalariedEmployee`

- Class `SalariedEmployee` (Figs. 12.11–12.12) derives from class `Employee`

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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 public:
11    SalariedEmployee(const std::string&, const std::string&,
12       const std::string&, double = 0.0);
13    virtual ~SalariedEmployee() = default; // virtual destructor
14
15    void setWeeklySalary(double); // set weekly salary
16    double getWeeklySalary() const; // return weekly salary
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
17
18      // keyword virtual signals intent to override
19      virtual double earnings() const override; // calculate earnings
20      virtual std::string toString() const override; // string representation
21   private:
22      double weeklySalary; // salary per week
23   };
24
25   #endif // SALARIED_H
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.2 Creating Concrete Derived Class `SalariedEmployee` (cont.)

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.2 Creating Concrete Derived Class `SalariedEmployee` (cont.)

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
24
25  // return salary
26  double SalariedEmployee::getWeeklySalary() const {return weeklySalary;}
27
28  // calculate earnings;
29  // override pure virtual function earnings in Employee
30  double SalariedEmployee::earnings() const {return getWeeklySalary();}
31
32  // return a string representation of SalariedEmployee's information
33  string SalariedEmployee::toString() const {
34     ostringstream output;
35     output << fixed << setprecision(2);
36     output << "salaried employee: "
37        << Employee::toString() // reuse abstract base-class function
38        << "\nweekly salary: " << getWeeklySalary();
39     return output.str();
40  }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.2 Creating Concrete Derived Class `SalariedEmployee` (cont.)

- Function `toString` of class `SalariedEmployee` overrides `Employee` function `toString`.
- If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would inherit the `Employee` version of `toString`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.3 Creating Concrete Derived Class `CommissionEmployee`

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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 public:
11    CommissionEmployee(const std::string&, const std::string&,
12       const std::string&, double = 0.0, double = 0.0);
13    virtual ~CommissionEmployee() = default; // virtual destructor
14
15    void setCommissionRate(double); // set commission rate
16    double getCommissionRate() const; // return commission rate
17
18    void setGrossSales(double); // set gross sales amount
19    double getGrossSales() const; // return gross sales amount
20
21    // keyword virtual signals intent to override
22    virtual double earnings() const override; // calculate earnings
23    virtual std::string toString() const override; // string representation
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
24   private:
25      double grossSales; // gross weekly sales
26      double commissionRate; // commission percentage
27   };
28
29   #endif // COMMISSION_H
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.14: CommissionEmployee.cpp
2  // CommissionEmployee class 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    : Employee(first, last, ssn) {
13    setGrossSales(sales);
14    setCommissionRate(rate);
15 }
16
17 // set gross sales amount
18 void CommissionEmployee::setGrossSales(double sales) {
19    if (sales < 0.0) {
20       throw invalid_argument("Gross sales must be >= 0.0");
21    }
22
23    grossSales = sales;
24 }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
25
26  // return gross sales amount
27  double CommissionEmployee::getGrossSales() const {return grossSales;}
28
29  // set commission rate
30  void CommissionEmployee::setCommissionRate(double rate) {
31     if (rate <= 0.0 || rate > 1.0) {
32        throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
33     }
34
35     commissionRate = rate;
36  }
37
38  // return commission rate
39  double CommissionEmployee::getCommissionRate() const {
40     return commissionRate;
41  }
42
43  // calculate earnings; override pure virtual function earnings in Employee
44  double CommissionEmployee::earnings() const {
45     return getCommissionRate() * getGrossSales();
46  }
47
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
48  // return a string representation of CommissionEmployee's information
49  string CommissionEmployee::toString() const {
50     ostringstream output;
51     output << fixed << setprecision(2);
52     output << "commission employee: " << Employee::toString()
53        << "\ngross sales: " << getGrossSales()
54        << "; commission rate: " << getCommissionRate();
55     return output.str();
56  }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.4 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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 public:
11    BasePlusCommissionEmployee(const std::string&, const std::string&,
12       const std::string&, double = 0.0, double = 0.0, double = 0.0);
13    virtual ~BasePlusCommissionEmployee() = default; // virtual destructor
14
15    void setBaseSalary(double); // set base salary
16    double getBaseSalary() const; // return base salary
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
17
18      // keyword virtual signals intent to override
19      virtual double earnings() const override; // calculate earnings
20      virtual std::string toString() const override; // string representation
21   private:
22      double baseSalary; // base salary per week
23   };
24
25   #endif // BASEPLUS_H
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.16: BasePlusCommissionEmployee.cpp
2  // 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    : CommissionEmployee(first, last, ssn, sales, rate) {
14    setBaseSalary(salary); // validate and store base salary
15 }
16
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 }
```

Fig. 12.16 | BasePlusCommissionEmployee class implementation file. (Part 1 of 2.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
25
26  // return base salary
27  double BasePlusCommissionEmployee::getBaseSalary() const {
28      return baseSalary;
29  }
30
31  // calculate earnings;
32  // override virtual function earnings in CommissionEmployee
33  double BasePlusCommissionEmployee::earnings() const {
34      return getBaseSalary() + CommissionEmployee::earnings();
35  }
36
37  // return a string representation of a BasePlusCommissionEmployee
38  string BasePlusCommissionEmployee::toString() const {
39     ostringstream output;
40     output << fixed << setprecision(2);
41     output << "base-salaried " << CommissionEmployee::toString()
42        << "; base salary: " << getBaseSalary();
43     return output.str();
44  }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.5 Demonstrating Polymorphic Processing

- Fig. 12.17 creates an object of each of the four concrete classes `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.
- Manipulates these objects, first with static binding, then polymorphically, using a `vector` of `Employee` pointers.
- Lines 20–25 create objects of each of the four concrete `Employee` derived classes.
- Lines 28–34 output each `Employee`'s information and earnings.
  - Each member-function invocation is an example of *static binding*—at *compile time*, because we are using *name*, the *compiler* can identify each object's type to determine which `toString` and `earnings` functions are called.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 12.17: fig12_17.cpp
2  // Processing Employee derived-class objects with static binding
3  // then 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    cout << fixed << setprecision(2); // set floating-point formatting
18
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 1 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
19      // create derived-class objects
20      SalariedEmployee salariedEmployee{
21         "John", "Smith", "111-11-1111", 800};
22      CommissionEmployee commissionEmployee{
23         "Sue", "Jones", "333-33-3333", 10000, .06};
24      BasePlusCommissionEmployee basePlusCommissionEmployee{
25         "Bob", "Lewis", "444-44-4444", 5000, .04, 300};
26
27      // output each Employee's information and earnings using static binding
28      cout << "EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING\n"
29         << salariedEmployee.toString()
30         << "\nearned $" << salariedEmployee.earnings() << "\n\n"
31         << commissionEmployee.toString()
32         << "\nearned $" << commissionEmployee.earnings() << "\n\n"
33         << basePlusCommissionEmployee.toString()
34         << "\nearned $" << basePlusCommissionEmployee.earnings() << "\n\n";
35
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 2 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
36      // create and initialize vector of three base-class pointers
37      vector<Employee *> employees{&salariedEmployee, &commissionEmployee,
38         &basePlusCommissionEmployee};
39
40      cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING\n\n";
41
42      // call virtualViaPointer to print each Employee's information
43      // and earnings using dynamic binding
44      cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS POINTERS\n";
45
46      for (const Employee* employeePtr : employees) {
47         virtualViaPointer(employeePtr);
48      }
49
50      // call virtualViaReference to print each Employee's information
51      // and earnings using dynamic binding
52      cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS REFERENCES\n";
53
54      for (const Employee* employeePtr : employees) {
55         virtualViaReference(*employeePtr); // note dereferencing
56      }
57   }
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 3 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
58
59   // call Employee virtual functions toString and earnings off a
60   // base-class pointer using dynamic binding
61   void virtualViaPointer(const Employee* const baseClassPtr) {
62      cout << baseClassPtr->toString()
63         << "\nearned $" << baseClassPtr->earnings() << "\n\n";
64   }
65
66   // call Employee virtual functions toString and earnings off a
67   // base-class reference using dynamic binding
68   void virtualViaReference(const Employee& baseClassRef) {
69      cout << baseClassRef.toString()
70         << "\nearned $" << baseClassRef.earnings() << "\n\n";
71   }
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 4 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 5 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
EMPLOYEES PROCESSED POLYMORPHICALLY VIA 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

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 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 6 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 7 of 7.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 37-38 create and initialize the `vector` `employees`, which contains three `Employee` pointers.
- The compiler allows these assignments, because a `SalariedEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 46–48 traverse `vector employees` and invoke function `virtualViaPointer` 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 `toString` and `earnings`
- 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 54–59 traverse `employees` and invoke function `virtualViaReference` for each `vector` element.
- Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a reference to the object obtained by dereferencing an `employees` element pointer
- Each call to `virtualViaReference` invokes `virtual` functions `toString` and `earnings` 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- How C++ can implement polymorphism, `virtual` functions and dynamic binding.
- Polymorphism is accomplished through three levels of pointers (i.e., "triple indirection").
- 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 (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 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 the `Employee` hierarchy.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

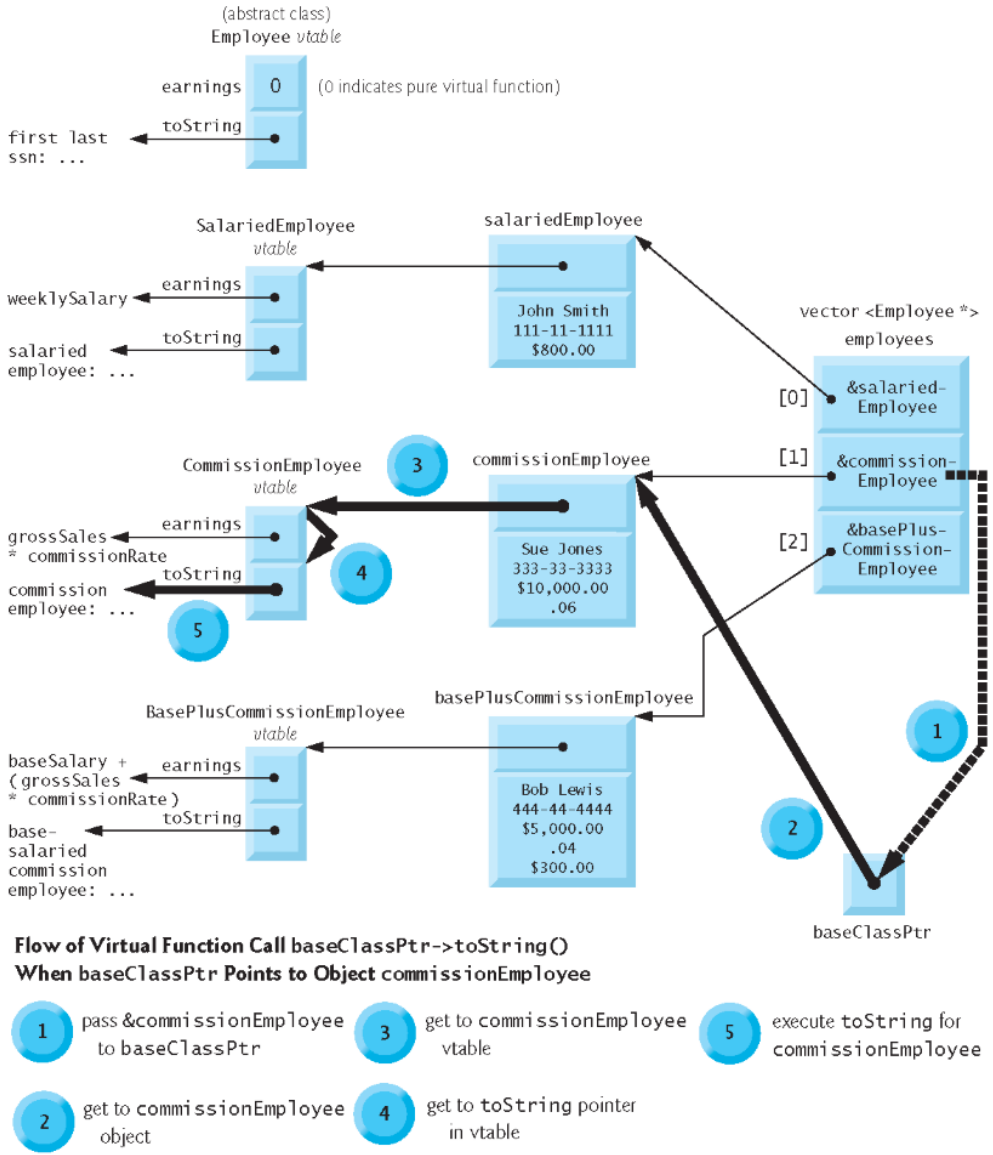

Fig. 12.18 | How `virtual` function calls work.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- In the `Employee` class *vtable*, the first function pointer is set to 0 (`nullptr`), because function `earnings` is a *pure* `virtual` function and therefore lacks an implementation.
- The second function pointer points to function `toString`, which returns the employee's full name and SSN.
- A class with at least one null pointer in its *vtable* is *abstract*.
- Classes without any null *vtable* pointers are concrete.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- 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 `toString`, so the corresponding function pointer points to the `SalariedEmployee` version of `toString`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- The `earnings` function pointer in the *vtable* for class `CommissionEmployee` points to `CommissionEmployee`'s `earnings` function.
- The `toString` function pointer points to the `CommissionEmployee` version of the function.
- As in class `SalariedEmployee`, both functions override the functions in class `Employee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- The `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee` points to the `BasePlusCommissionEmployee`'s `earnings` function.
- The `toString` function pointer points to the `BasePlusCommissionEmployee` version of the function.
- Both functions override the functions in class `CommissionEmployee`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- 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.
- *For each new object of a class with one or more* `virtual` *functions, the compiler attaches 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 (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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- Consider the call `baseClassPtr->toString()` in function `virtualViaPointer`
- Assume that `baseClassPtr` contains `employees[1]` (i.e., the address of object `commissionEmployee` in `employees`).
- The compiler determines that the call is indeed being made via a *base-class pointer* and that `toString` is a `virtual` function and that `toString` 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 (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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 (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 `toString` function pointer.
  5. *Dereference* the `toString` function pointer to form the "name" of the actual function to execute, and use the function call operator `()` to execute the appropriate `toString` function.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Performance Tip 12.1

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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.*

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- For the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployees` by adding 10 percent to their base salaries.
- When processing `Employees` polymorphically in Section 12.6.5, we did not worry about the "specifics."
- To adjust the base salaries of `BasePlusCommissionEmployees`, we have to determine the specific type of each `Employee` object at execution time, then act appropriately.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.9 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`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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    // set floating-point output formatting
17    cout << fixed << setprecision(2);
18
19    // create and initialize vector of three base-class pointers
20    vector<Employee*> employees{
21       new SalariedEmployee("John", "Smith", "111-11-1111", 800),
22       new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06),
23       new BasePlusCommissionEmployee(
24          "Bob", "Lewis", "444-44-4444", 5000, .04, 300)};
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
25
26      // polymorphically process each element in vector employees
27      for (Employee* employeePtr : employees) {
28         cout << employeePtr->toString() << endl; // output employee
29
30         // attempt to downcast pointer
31         BasePlusCommissionEmployee* derivedPtr =
32            dynamic_cast<BasePlusCommissionEmployee*>(employeePtr);
33
34         // determine whether element points to a BasePlusCommissionEmployee
35         if (derivedPtr != nullptr) { // true for "is a" relationship
36            double oldBaseSalary = derivedPtr->getBaseSalary();
37            cout << "old base salary: $" << oldBaseSalary << endl;
38            derivedPtr->setBaseSalary(1.10 * oldBaseSalary);
39            cout << "new base salary with 10% increase is: $"
40               << derivedPtr->getBaseSalary() << endl;
41         }
42
43         cout << "earned $" << employeePtr->earnings() << "\n\n";
44      }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
45
46      // release objects pointed to by vector's elements
47      for (const Employee* employeePtr : employees) {
48         // output class name
49         cout << "deleting object of "
50            << typeid(*employeePtr).name() << endl;
51
52         delete employeePtr;
53      }
54   }
```

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- Since we process the `Employees` 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 so they can receive the 10 percent salary increase.
- To accomplish this, we use operator `dynamic_cast` to determine whether the current `Employee`'s type is `BasePlusCommissionEmployee`.
- This is the *downcast* operation we referred to in Section 12.3.3.
- Lines 31–32 dynamically downcast `employeePtr` from type `Employee *` to type `BasePlusCommissionEmployee *`.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 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.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

12.8 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 31–32 *is not* `nullptr`, the object *is* the correct type, and the `if` statement performs the special processing required for the `BasePlusCommissionEmployee` object.

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

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

- Operator `typeid` 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` 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>`

©1992-2017 by Pearson Education, Inc. All Rights Reserved.

Portability Tip 12.1

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

©1992-2017 by Pearson Education, Inc. All Rights Reserved.