[Page 686]

Chapter 13. Object-Oriented Programming: Polymorphism

One Ring to rule them all, One Ring to find them, One Ring to bring them all and in the darkness bind them.

John Ronald Reuel Tolkien

The silence often of pure innocence

Persuades when speaking fails.

William Shakespeare

General propositions do not decide concrete cases.

Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

Alfred North Whitehead

OBJECTIVES

In this chapter you will learn:

- What polymorphism is, how it makes programming more convenient, and how it makes systems more extensible and maintainable.
- To declare and use `virtual` functions to effect polymorphism.
- The distinction between abstract and concrete classes.
- To declare pure `virtual` functions to create abstract classes.
- How to use run-time type information (RTTI) with downcasting, `dynamic_cast`, `typeid` and `type_info`.
- How C++ implements `virtual` functions and dynamic binding "under the hood."
- How to use `virtual` destructors to ensure that all appropriate destructors run on an object.

[Page 687]

Outline


[Page 687 (continued)]

13.1. Introduction

In Chapters 912, we discussed key object-oriented programming technologies including classes, objects, encapsulation, operator overloading and inheritance. We now continue our study of OOP by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects of classes that are part of the same class hierarchy as if they are all objects of the hierarchy's base class. As we will soon see, polymorphism works off base-class pointer handles and base-class reference handles, but not off name handles.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation. Imagine that each of these classes inherits from base class `Animal`, which contains a function `move` and maintains an animal's current location. Each derived class implements function `move`. Our program maintains a `vector` of pointers to objects of the various `Animal` derived classes. To simulate the animals' movements, the program sends each object the same message once per secondnamely, `move`. However, each specific type of `Animal` responds to a `move` message in its own unique waya `Fish` might swim two feet, a `Frog` might jump three feet and a `Bird` might fly ten feet. The program issues the same message (i.e., `move`) to each animal object generically, but each object knows how to modify its location appropriately for its specific type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same function call is the key concept of polymorphism. The same message (in this case, `move`) sent to a variety of objects has "many forms" of resultshence the term polymorphism.

[Page 688]

With polymorphism, we can design and implement systems that are easily extensiblenew 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 generically. 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 the programmer adds to the hierarchy. For example, if we create class `Tortoise` that inherits from class `Animal` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that process each `Animal` generically can remain the same.

We begin with a sequence of small, focused examples that lead up to an understanding of `virtual` functions and dynamic bindingpolymorphism's two underlying technologies. We then present a case study that revisits Chapter 12's `Employee` hierarchy. In the case study, we define a common "interface" (i.e., set of functionality) for all the classes in the hierarchy. This common functionality among employees is defined in a so-called abstract base class, `Employee`, from which classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and class `BaseCommissionEmployee` inherits indirectly. We will soon see what makes a class "abstract" or its opposite"concrete."

In this hierarchy, every employee has an `earnings` function to calculate the employee's weekly pay. These `earnings` functions vary by employee typefor instance, `SalariedEmployees` are paid a fixed weekly salary regardless of the number of hours worked, while `HourlyEmployees` are paid by the hour and receive overtime pay. We show how to process each employee "in the general"that is, using base-class pointers to call the `earnings` function of several derived-class objects. This way, the programmer needs to be concerned with only one type of function call, which can be used to execute several different functions based on the objects referred to by the base-class pointers.

A key feature of this chapter is its (optional) detailed discussion of polymorphism, `virtual` functions and dynamic binding "under the hood," which uses a detailed diagram to explain how polymorphism can be implemented in C++.

Occasionally, when performing polymorphic processing, we need to program "in the specific," meaning that operations need to be performed on a specific type of object in a hierarchythe operation cannot be generally applied to several types of objects. We reuse our `Employee` hierarchy to demonstrate the powerful capabilities of **run-time type information (RTTI)** and **dynamic casting**, which enable a program to

determine the type of an object at execution time and act on that object accordingly. We use these capabilities to determine whether a particular employee object is a `BasePlusCommissionEmployee`, then give that employee a 10 percent bonus on his or her base salary.


[Page 689]

13.2. Polymorphism Examples

In this section, we discuss several polymorphism examples. With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked. This gives the programmer tremendous expressive capability. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. Therefore, any operation (such as calculating the perimeter or the area) that can be performed on an object of class `Quadrilateral` also can be performed on an object of class `Rectangle`. Such operations also can be performed on other kinds of `Quadrilaterals`, such as `Squares`, `Parallelograms` and `TRapezoids`. The polymorphism occurs when a program invokes a `virtual` function through a base-class (i.e., `Quadrilateral`) pointer or referenceC++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated. You will see a code example that illustrates this process in Section 13.3.

As another example, suppose that we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venutian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each of these classes inherits from the common base class `SpaceObject`, which contains member function `draw`. Each derived class implements this function in a manner appropriate for that class. A screen-manager program maintains a container (e.g., a `vector`) that holds `SpaceObject` pointers to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same messagenamely, `draw`. Each type of object responds in a unique way. For example, a `Martian` object might draw itself in red with the appropriate number of antennae. A `SpaceShip` object might draw itself as a silver flying saucer. A `LaserBeam` object might draw itself as a bright red beam across the screen. Again, the same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A polymorphic screen manager facilitates adding new classes to a system with minimal modifications to its code. Suppose that we want to add objects of class `Mercurian` to our video game. To do so, we must build a class `Mercurian` that inherits from `SpaceObject`, but provides its own definition of member function `draw`. Then, when pointers to objects of class `Mercurian` appear in the container, the programmer does not need to modify the code for the screen manager. The screen manager invokes member function `draw` on every object in the container, regardless of the object's type, so the new `Mercurian` objects simply "plug right in." Thus, without modifying the system (other than to build and include the classes themselves), programmers can use polymorphism to accommodate additional classes, including ones that were not even envisioned when the system was created.

Software Engineering Observation 13.1

With `virtual` functions and polymorphism, you can deal in generalities and let the executiontime 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).

Software Engineering Observation 13.2

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the 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.

[Page 690]

13.3. Relationships Among Objects in an Inheritance Hierarchy

Section 12.4 created an employee class hierarchy, in which class `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. The Chapter 12 examples manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using the objects' names to invoke their member functions. We now examine the relationships among classes in a hierarchy more closely. The next several sections present a series of examples that demonstrate 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. Toward the end of this section, we demonstrate how to get polymorphic behavior from base-class pointers aimed at derived-class objects.

In Section 13.3.1, we assign the address of a derived-class object to a base-class pointer, then show that invoking a function via the base-class pointer invokes the base-class functionalityi.e., the type of the handle determines which function is called. In Section 13.3.2, we assign the address of a base-class object to a derived-class pointer, which results in a compilation error. We discuss the error message and investigate why the compiler does not allow such an assignment. In Section 13.3.3, we assign the address of a derived-class object to a base-class pointer, then examine how the base-class pointer can be used to invoke only the base-class functionalitywhen we attempt to invoke derived-class member functions through the base-class pointer, compilation errors occur. Finally, in Section 13.3.4, we introduce `virtual` functions and polymorphism by declaring a base-class function as `virtual`. We then assign a derived-class object to the base-class pointer and use that pointer to invoke derived-class functionalityprecisely the capability we need to achieve polymorphic behavior.

A key concept in these examples is to demonstrate that an object of a derived class can be treated as an object of its base class. This enables various interesting manipulations. For example, a program can create an array of base-class pointers that point to objects of many derived-class types. 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. However, we cannot treat a base-class object as an object of any of its derived classes. For example, a `CommissionEmployee` is not a `BasePlusCommissionEmployee` in the hierarchy defined in Chapter 12a `CommissionEmployee` does not have a `baseSalary` data member and does not have member functions `setBaseSalary` and `getBaseSalary`. The is-a relationship applies only from a derived class to its direct and indirect base classes.

13.3.1. Invoking Base-Class Functions from Derived-Class Objects

The example in Figs. 13.113.5 demonstrates three ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects. The first two are straightforwardwe 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). Then, 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).

Class `CommissionEmployee` (Figs. 13.113.2), which we discussed in Chapter 12, is used to represent employees who are paid a percentage of their sales. Class `BasePlusCommissionEmployee` (Figs. 13.313.4), which we also discussed in Chapter 12, is used to represent employees who receive a base salary plus a percentage of their sales. Each `BasePlusCommissionEmployee` object is a `CommissionEmployee` that also

has a base salary. Class `BasePlusCommissionEmployee`'s earnings member function (lines 3235 of Fig. 13.4) redefines class `CommissionEmployee`'s earnings member function (lines 7982 of Fig. 13.2) to include the object's base salary. Class `BasePlusCommissionEmployee`'s print member function (lines 3846 of Fig. 13.4) redefines class `CommissionEmployee`'s print member function (lines 8592 of Fig. 13.2) to display the same information as the `print` function in class `CommissionEmployee`, as well as the employee's base salary.

[Page 691]

Figure 13.1. **`CommissionEmployee` class header file.**

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

[Page 692]

Figure 13.2. **`CommissionEmployee` class implementation file.**
(This item is displayed on pages 692 - 693 in the print version)

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
```

```
 4  using std::cout;
 5
 6  #include "CommissionEmployee.h" // CommissionEmployee class definition
 7
 8  // constructor
 9  CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13  {
14     setGrossSales( sales ); // validate and store gross sales
15     setCommissionRate( rate ); // validate and store commission rate
16  } // end CommissionEmployee constructor
17
18  // set first name
19  void CommissionEmployee::setFirstName( const string &first )
20  {
21     firstName = first; // should validate
22  } // end function setFirstName
23
24  // return first name
25  string CommissionEmployee::getFirstName() const
26  {
27     return firstName;
28  } // end function getFirstName
29
30  // set last name
31  void CommissionEmployee::setLastName( const string &last )
32  {
33     lastName = last; // should validate
34  } // end function setLastName
35
36  // return last name
37  string CommissionEmployee::getLastName() const
38  {
39     return lastName;
40  } // end function getLastName
41
42  // set social security number
43  void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44  {
45     socialSecurityNumber = ssn; // should validate
46  } // end function setSocialSecurityNumber
47
48  // return social security number
49  string CommissionEmployee::getSocialSecurityNumber() const
50  {
51     return socialSecurityNumber;
52  } // end function getSocialSecurityNumber
53
54  // set gross sales amount
55  void CommissionEmployee::setGrossSales( double sales )
56  {
57     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58  } // end function setGrossSales
59
60  // return gross sales amount
61  double CommissionEmployee::getGrossSales() const
62  {
63     return grossSales;
64  } // end function getGrossSales
65
66  // set commission rate
67  void CommissionEmployee::setCommissionRate( double rate )
68  {
69     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70  } // end function setCommissionRate
71
```

```
72  // return commission rate
73  double CommissionEmployee::getCommissionRate() const
74  {
75     return commissionRate;
76  } // end function getCommissionRate
77
78  // calculate earnings
79  double CommissionEmployee::earnings() const
80  {
81     return getCommissionRate() * getGrossSales();
82  } // end function earnings
83
84  // print CommissionEmployee object
85  void CommissionEmployee::print() const
86  {
87     cout << "commission employee: "
88        << getFirstName() << ' ' << getLastName()
89        << "\nsocial security number: " << getSocialSecurityNumber()
90        << "\ngross sales: " << getGrossSales()
91        << "\ncommission rate: " << getCommissionRate();
92  }  // end function print
```

[Page 693]

Figure 13.3. `BasePlusCommissionEmployee` **class header file.**
(This item is displayed on pages 693 - 694 in the print version)

```
1  // Fig. 13.3: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

[Page 694]

Figure 13.4. `BasePlusCommissionEmployee` **class implementation file.**
(This item is displayed on pages 694 - 695 in the print version)

```
1  // Fig. 13.4: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

[Page 695]

In Fig. 13.5, lines 1920 create a `CommissionEmployee` object and line 23 creates a pointer to a `CommissionEmployee` object; lines 2627 create a `BasePlusCommissionEmployee` object and line 30 creates a pointer to a `BasePlusCommissionEmployee` object. Lines 37 and 39 use each object's name (`CommissionEmployee` and `BasePlusCommissionEmployee`, respectively) to invoke each object's `print` member function. Line 42 assigns the address of base-class object `CommissionEmployee` to base-class pointer `CommissionEmployeePtr`, which line 45 uses to invoke member function `print` on that `CommissionEmployee` object. This invokes the version of `print` defined in base class `CommissionEmployee`. Similarly, line 48 assigns the address of derived-class object `BasePlusCommissionEmployee` to derived-class pointer `BasePlusCommissionEmployeePtr`, which line 52 uses to invoke member function `print` on that `BasePlusCommissionEmployee` object. This invokes the version of `print` defined in derived class `BasePlusCommissionEmployee`. Line 55 then assigns the address of derived-class object `BasePlusCommissionEmployee` to base-class pointer `CommissionEmployeePtr`,

which line 59 uses to invoke member function `print`. The C++ compiler allows this "crossover" because an object of a derived class is an object of its base class. Note that despite the fact that the base class `CommissionEmployee` pointer points to a derived class `BasePlusCommissionEmployee` object, the base class `CommissionEmployee`'s `print` member function is invoked (rather than `BasePlusCommissionEmployee`'s `print` function). The output of each `print` member-function invocation in this program reveals that the invoked functionality depends on the type of the handle (i.e., the pointer or reference type) used to invoke the function, not the type of the object to which the handle points. In Section 13.3.4, when we introduce `virtual` functions, we demonstrate that it is possible to invoke the object type's functionality, rather than invoke the handle type's functionality. We will see that this is crucial to implementing polymorphic behaviorthe key topic of this chapter.

[Page 697]

Figure 13.5. Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers.
(This item is displayed on pages 695 - 697 in the print version)

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18    // create base-class object
19    CommissionEmployee commissionEmployee(
20       "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22    // create base-class pointer
23    CommissionEmployee *commissionEmployeePtr = 0;
24
25    // create derived-class object
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29    // create derived-class pointer
30    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
31
32    // set floating-point output formatting
33    cout << fixed << setprecision( 2 );
34
35    // output objects commissionEmployee and basePlusCommissionEmployee
36    cout << "Print base-class and derived-class objects:\n\n";
37    commissionEmployee.print(); // invokes base-class print
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // invokes derived-class print
40
41    // aim base-class pointer at base-class object and print
42    commissionEmployeePtr = &commissionEmployee; // perfectly natural
43    cout << "\n\n\nCalling print with base-class pointer to "
44       << "\nbase-class object invokes base-class print function:\n\n";
45    commissionEmployeePtr->print(); // invokes base-class print
46
47    // aim derived-class pointer at derived-class object and print
48    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
```

```
49     cout << "\n\n\nCalling print with derived-class pointer to "
50        << "\nderived-class object invokes derived-class "
51        << "print function:\n\n";
52     basePlusCommissionEmployeePtr->print(); // invokes derived-class print
53
54     // aim base-class pointer at derived-class object and print
55     commissionEmployeePtr = &basePlusCommissionEmployee;
56     cout << "\n\n\nCalling print with base-class pointer to "
57        << "derived-class object\ninvokes base-class print "
58        << "function on that derived-class object:\n\n";
59     commissionEmployeePtr->print(); // invokes base-class print
60     cout << endl;
61     return 0;
62  } // end main
```

```
Print 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

Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling print with derived-class pointer to
derived-class object invokes derived-class print 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 print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

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

[Page 698]

13.3.2. Aiming Derived-Class Pointers at Base-Class Objects

In Section 13.3.1, we assigned the address of a derived-class object to a base-class pointer and explained that the C++ compiler allows this assignment, because a derived-class object is a base-class object. We take the

opposite approach in Fig. 13.6, as we aim a derived-class pointer at a base-class object. [Note: This program uses classes `CommissionEmployee` and `BasePlusCommissionEmployee` of Figs. 13.113.4] Lines 89 of Fig. 13.6 create a `CommissionEmployee` object, and line 10 creates a `BasePlusCommissionEmployee` pointer. Line 14 attempts to assign the address of base-class object `commissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, but the C++ compiler generates an error. The compiler prevents this assignment, because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`. Consider the consequences if the compiler were to allow this assignment. Through a `BasePlusCommissionEmployee` pointer, we can invoke every `BasePlusCommissionEmployee` member function, including `setBaseSalary`, for the object to which the pointer points (i.e., the base-class object `commissionEmployee`). However, the `CommissionEmployee` object does not provide a `setBaseSalary` member function, nor does it provide a `baseSalary` data member to set. This could lead to problems, because member function `setBaseSalary` would assume that there is a `baseSalary` data member to set at its "usual location" in a `BasePlusCommissionEmployee` object. This memory does not belong to the `CommissionEmployee`, object so member function `setBaseSalary` might overwrite other important data in memory, possibly data that belongs to a different object.

Figure 13.6. Aiming a derived-class pointer at a base-class object.
(This item is displayed on pages 698 - 699 in the print version)

```
1  // Fig. 13.6: fig13_06.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  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

Borland C++ command-line compiler error messages:

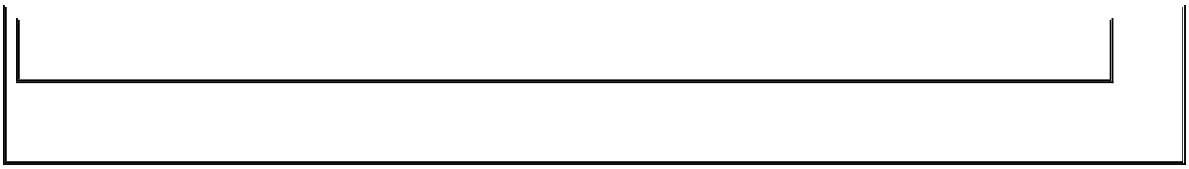
```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *' in function main()
```

GNU C++ compiler error messages:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
   `BasePlusCommissionEmployee*'
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
   '=' : cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
         Cast from base to derived requires dynamic_cast or static_cast
```



[Page 699]

13.3.3. Derived-Class Member-Function Calls via Base-Class Pointers

Off a base-class pointer, the compiler allows us to invoke only bases-class member functions. Thus, 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 13.7 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer. [Note: We are again using classes `CommissionEmployee` and `BasePlusCommissionEmployee` of Figs. 13.113.4] Line 9 creates `commissionEmployeePtr`a pointer to a `CommissionEmployee` objectand lines 1011 create a `BasePlusCommissionEmployee` object. Line 14 aims `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`. Recall from Section 13.3.1 that the C++ compiler allows this, because a `BasePlusCommissionEmployee` is a `CommissionEmployee` (in the sense that a `BasePlusCommissionEmployee` object contains all the functionality of a `CommissionEmployee` object). Lines 1822 invoke base-class member functions `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `getGrossSales` and `getCommissionRate` off the base-class pointer. All of these calls are legitimate, because `BasePlusCommissionEmployee` inherits these member functions from `CommissionEmployee`. We know that `commissionEmployeePtr` is aimed at a `BasePlusCommissionEmployee` object, so in lines 2627 we attempt to invoke `BasePlusCommissionEmployee` member functions `getBaseSalary` and `setBaseSalary`. The C++ compiler generates errors on both of these lines, because these are not member functions of base-class `CommissionEmployee`. The handle can invoke only those functions that are members of that handle's associated class type. (In this case, off a `CommissionEmployee *`, we can invoke only `CommissionEmployee` member functions `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `print`.)

[Page 700]

Figure 13.7. Attempting to invoke derived-class-only functions via a base-class pointer.
(This item is displayed on pages 699 - 700 in the print version)

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer
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();
23
24     // attempt to invoke derived-class-only member functions
25     // on derived-class object through base-class pointer
26     double baseSalary = commissionEmployeePtr->getBaseSalary();
27     commissionEmployeePtr->setBaseSalary( 500 );
28     return 0;
29  } // end main
```

Borland C++ command-line compiler error messages:

```
Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
   'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
   'CommissionEmployee' in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
   each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)
```

It turns out that the C++ compiler does 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 pointera technique known as **downcasting**. As you learned in Section 13.3.1, it is possible to aim a base-class pointer at a derived-class object. However, as we demonstrated in Fig. 13.7, a base-class pointer can be used to invoke only the functions declared in the base class. Downcasting allows a program to perform 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. We will show you a concrete example of downcasting in Section 13.8.

[Page 701]

Software Engineering Observation 13.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it is acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to send that derived-class object messages that do not appear in the base class.

13.3.4. Virtual Functions

In Section 13.3.1, we aimed a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, then invoked member function `print` through that pointer. Recall that the type of the handle determines which class's functionality to invoke. In that case, the `CommissionEmployee` pointer invoked the `CommissionEmployee` member function `print` on the `BasePlusCommissionEmployee` object, even though the pointer was aimed at a `BasePlusCommissionEmployee` object that has its own customized `print` function. With `virtual` functions, the type of the object being pointed to, not the type of the handle, determines which version of a `virtual` function to invoke.

First, we consider why `virtual` functions are useful. Suppose that a set of 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`. Although each class has its own `draw` function, 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 generically as objects of the base class `Shape`. Then, 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.

To enable this kind of 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. From an implementation perspective, overriding a function is no different than redefining one (which is the approach we have been using until now). 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 do not declare the base-class function as `virtual`, we can redefine that function. By contrast, 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 keyword `virtual` in the base class. For example,

```
virtual void draw() const;
```

would appear in base class `Shape`. The preceding prototype declares that function `draw` is a `virtual` function that takes no arguments and returns nothing. The function is declared `const` because a `draw` function typically would not make changes to the `Shape` object on which it is invoked. Virtual functions do not necessarily have to be `const` functions.

Software Engineering Observation 13.4

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 class overrides it.

[Page 702]

Good Programming Practice 13.1

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

of the hierarchy to promote program clarity.

Error-Prevention Tip 13.1

When a programmer browses a class hierarchy to locate a class to reuse, it is possible that a function in that class will exhibit `virtual` function behavior even though it is not explicitly declared `virtual`. This happens when the class inherits a `virtual` function from its base class, and it can lead to subtle logic errors. Such errors can be avoided by explicitly declaring all `virtual` functions `virtual` tHRoughout the inheritance hierarchy.

Software Engineering Observation 13.5

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.

If the program invokes a `virtual` function through a base-class pointer to a derived-class object (e.g., `shapePtr->draw ()`), the program will choose the correct derived-class `draw` function dynamically (i.e., at execution time) based on the object typenot the pointer type. Choosing the appropriate function to call at execution time (rather than at compile time) is known as **dynamic binding** or **late binding**.

When a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator (e.g., `squareObject.draw()`), the function invocation is resolved 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 objectthis is not polymorphic behavior. Thus, dynamic binding with `virtual` functions occurs only off pointer (and, as we will soon see, reference) handles.

Now let's see how `virtual` functions can enable polymorphic behavior in our employee hierarchy. Figures 13.813.9 are the header files for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively. Note that the only difference between these files and those of Fig. 13.1 and Fig. 13.3 is that we specify each class's `earnings` and `print` member functions as `virtual` (lines 3031 of Fig. 13.8 and lines 2122 of Fig. 13.9). Because functions `earnings` and `print` are `virtual` in class `CommissionEmployee`, class `BasePlusCommissionEmployee`'s `earnings` and `print` functions override class `CommissionEmployee`'s. Now, if we aim a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, and the program uses that pointer to call either function `earnings` or `print`, the `BasePlusCommissionEmployee` object's corresponding function will be invoked. There were no changes to the member-function implementations of classes `CommissionEmployee` and `BasePlusCommissionEmployee`, so we reuse the versions of Fig. 13.2 and Fig. 13.4.

Figure 13.8. `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`.
(This item is displayed on page 703 in the print version)

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
```

```
13        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // set first name
16     string getFirstName() const; // return first name
17
18     void setLastName( const string & ); // set last name
19     string getLastName() const; // return last name
20
21     void setSocialSecurityNumber( const string & ); // set SSN
22     string getSocialSecurityNumber() const; // return SSN
23
24     void setGrossSales( double ); // set gross sales amount
25     double getGrossSales() const; // return gross sales amount
26
27     void setCommissionRate( double ); // set commission rate
28     double getCommissionRate() const; // return commission rate
29
30     virtual double earnings() const; // calculate earnings
31     virtual void print() const; // print CommissionEmployee object
32  private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // gross weekly sales
37     double commissionRate; // commission percentage
38  }; // end class CommissionEmployee
39
40  #endif
```

Figure 13.9. **`BasePlusCommissionEmployee`** **class header file declares** **`earnings`** **and** **`print`** **functions as** **`virtual`**.
(This item is displayed on pages 703 - 704 in the print version)

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

We modified Fig. 13.5 to create the program of Fig. 13.10. Lines 4657 demonstrate again that a

`CommissionEmployee` pointer aimed at a `CommissionEmployee` object can be used to invoke `CommissionEmployee` functionality, and a `BasePlusCommissionEmployee` pointer aimed at a `BasePlusCommissionEmployee` object can be used to invoke `BasePlusCommissionEmployee` functionality. Line 60 aims base-class pointer `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`. Note that when line 67 invokes member function `print` off the base-class pointer, the derived-class `BasePlusCommissionEmployee`'s `print` member function is invoked, so line 67 outputs different text than line 59 does in Fig. 13.5 (when member function `print` was not declared `virtual`). 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. The decision about which function to call is an example of polymorphism. Note again that when `commissionEmployeePtr` points to a `CommissionEmployee` object (line 46), class `CommissionEmployee`'s `print` function is invoked, and when `CommissionEmployeePtr` points to a `BasePlusCommissionEmployee` object, class `BasePlusCommissionEmployee`'s `print` function is invoked. Thus, the same message`print`, in this casesent (off a base-class pointer) to a variety of objects related by inheritance to that base class, takes on many formsthis is polymorphic behavior.

[Page 704]

Figure 13.10. Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object.

(This item is displayed on pages 704 - 706 in the print version)

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // create base-class object
18    CommissionEmployee commissionEmployee(
19       "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21    // create base-class pointer
22    CommissionEmployee *commissionEmployeePtr = 0;
23
24    // create derived-class object
25    BasePlusCommissionEmployee basePlusCommissionEmployee(
26       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28    // create derived-class pointer
29    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31    // set floating-point output formatting
32    cout << fixed << setprecision( 2 );
33
34    // output objects using static binding
35    cout << "Invoking print function on base-class and derived-class "
36       << "\nobjects with static binding\n\n";
37    commissionEmployee.print(); // static binding
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // static binding
40
41    // output objects using dynamic binding
```

```
42     cout << "\n\n\nInvoking print function on base-class and "
43       << "derived-class \nobjects with dynamic binding";
44
45     // aim base-class pointer at base-class object and print
46     commissionEmployeePtr = &commissionEmployee;
47     cout << "\n\nCalling virtual function print with base-class pointer"
48        << "\nto base-class object invokes base-class "
49        << "print function:\n\n";
50     commissionEmployeePtr->print(); // invokes base-class print
51
52     // aim derived-class pointer at derived-class object and print
53     basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54     cout << "\n\nCalling virtual function print with derived-class "
55        << "pointer\nto derived-class object invokes derived-class "
56        << "print function:\n\n";
57     basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59     // aim base-class pointer at derived-class object and print
60     commissionEmployeePtr = &basePlusCommissionEmployee;
61     cout << "\n\nCalling virtual function print with base-class pointer"
62        << "\nto derived-class object invokes derived-class "
63        << "print function:\n\n";
64
65     // polymorphism; invokes BasePlusCommissionEmployee's print;
66     // base-class pointer to derived-class object
67     commissionEmployeePtr->print();
68     cout << endl;
69     return 0;
70  } // end main
```

```
Invoking print 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: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print 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 virtual function print with base-class pointer
to derived-class object invokes derived-class print 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
```

[Page 707]

13.3.5. Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

Now that you have seen a complete application that processes diverse objects polymorphically, we summarize what you can and cannot do with base-class and derived-class objects and pointers. Although a derived-class object also is a base-class object, the two objects are nevertheless different. As discussed previously, derived-class objects can be treated as if they are base-class objects. This is a logical relationship, because the derived class contains all the members of the base class. However, base-class objects cannot be treated as if they are derived-class objectsthe derived class can have additional derived-class-only members. For this reason, aiming a derived-class pointer at a base-class object is not allowed without an explicit castsuch an assignment would leave the derived-class-only members undefined on the base-class object. The cast relieves the compiler of the responsibility of issuing an error message. In a sense, by using the cast you are saying, "I know that what I'm doing is dangerous and I take full responsibility for my actions."

In the current section and in Chapter 12, we have discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:

1. Aiming a base-class pointer at a base-class object is straightforwardcalls made off the base-class pointer simply invoke base-class functionality.

2. Aiming a derived-class pointer at a derived-class object is straightforwardcalls made off the derived-class pointer simply invoke derived-class functionality.

3. Aiming a base-class pointer at a derived-class object is safe, because the derived-class object is an object of its base class. However, this pointer can be used to invoke only base-class member functions. If the programmer attempts to refer to a derived-class-only member through the base-class pointer, the compiler reports an error. To avoid this error, the programmer must cast the base-class pointer to a derived-class pointer. The derived-class pointer can then be used to invoke the derived-class object's complete functionality. However, this techniquecalled downcastingis a potentially dangerous operation. Section 13.8 demonstrates how to safely use downcasting.

4. Aiming a derived-class pointer at a base-class object generates a compilation error. The is-a relationship applies only from a derived class to its direct and indirect base classes, and not vice versa. A base-class object does not contain the derived-class-only members that can be invoked off a derived-class pointer.

Common Programming Error 13.1

After aiming a base-class pointer at a derived-class object, attempting to reference derived-class-only members with the base-class pointer is a compilation error.

Common Programming Error 13.2

Treating a base-class object as a derived-class object can cause errors.


[Page 707 (continued)]

13.4. Type Fields and `switch` Statements

One way to determine the type of an object that is incorporated in a larger program is to use a `switch` statement. This allows us to distinguish among object types, then invoke an appropriate action for a particular object. For example, in a hierarchy of shapes in which each shape object has a `shapeType` attribute, a `switch` statement could check the object's `shapeType` to determine which `print` function to call.

[Page 708]

However, using `switch` logic exposes programs to a variety of potential problems. For example, the programmer 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, the programmer 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 13.6

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

Software Engineering Observation 13.7

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and more simple, sequential code. This simplification facilitates testing, debugging and program maintenance.


[Page 708 (continued)]

13.5. Abstract Classes and Pure `virtual` Functions

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it is useful to define classes from which the programmer never intends 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 will soon see, abstract classes are incompletederived classes must define the "missing pieces." We build programs with abstract classes in Section 13.6.

The purpose of an abstract class is to provide an appropriate base class from which other classes can inherit. Classes that can be used to instantiate objects are called **concrete classes**. Such classes provide implementations of every member function they define. We could have an abstract base class `TwoDimensionalShape` and derive such concrete classes as `Square`, `Circle` and `triangle`. We could also have an abstract base class `THReeDimensionalShape` and derive such concrete classes as `Cube`, `Sphere` and `Cylinder`. 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 reasonable to instantiate objects.

An inheritance hierarchy does not need to contain any abstract classes, but, as we will see, many good 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`. On the next level of the hierarchy we have two more abstract base classes, namely, `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy defines concrete classes for two-dimensional shapes (namely, `Circle`, `Square` and `TRiangle`) and for three-dimensional shapes (namely, `Sphere, Cube` and `TeTRahedron`).

[Page 709]

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 known as a **pure specifier**. Pure `virtual` functions do not provide implementations. Every concrete derived class must override all base-class pure `virtual` functions with concrete implementations of those functions. 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 provide 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 the programmer wants all concrete derived classes to implement the function. Returning to our earlier example of space objects, it does not make sense for the base class `SpaceObject` to have an implementation for function `draw` (as there is no way to draw a generic space object without having more information about what type of space object is being drawn). An example of a function that would be defined as `virtual` (and not pure `virtual`) would be one that returns a name for the object. We can name a generic `SpaceObject` (for instance, as `"space object"`), so a default implementation for this function can be provided, and the function does not need to be pure `virtual`. The function is still declared `virtual`, however, because it is expected that derived classes will override this function to provide more specific names for the derived-class objects.

Software Engineering Observation 13.8

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 13.3

Attempting to instantiate an object of an abstract class causes a compilation error.

Common Programming Error 13.4

Failure to override a pure `virtual` function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Software Engineering Observation 13.9

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.

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.

Let us consider another application of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that the programmer will add to the system after writing the screen manager. The system might need to display various shapes, such as `Circles`, `triangles` or `Rectangles`, which are derived from abstract base class `Shape`. The screen manager uses `Shape` pointers to manage the objects that are displayed. To draw any object (regardless of the level at which that object's class appears in the inheritance hierarchy), the screen manager uses a base-class pointer to the object to invoke the object's `draw` function, which is a pure `virtual` function in base class `Shape`; therefore, each concrete derived class must implement function `draw`. Each `Shape` object in the inheritance hierarchy knows how to draw itself. The screen manager does not have to worry about the type of each object or whether the screen manager has ever encountered objects of that type.

[Page 710]

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the systemplace some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the `public` functions) offered by the device drivers are provided as pure `virtual` functions in the abstract base class. The implementations of these pure `virtual` functions are provided in the derived classes that correspond to the specific types of device drivers. This architecture also allows new devices to be added to a system easily, even after the operating system has been defined. The user can just plug in the device and install its new device driver. The operating system "talks" to this new device through its device driver, which has the same `public` member functions as all other device driversthose defined in the abstract base device driver class.

It is common in object-oriented programming to define an **iterator class** that can traverse all the objects in a

container (such as an array). For example, a program can print a list of objects in a `vector` by creating an iterator object, then using the iterator to obtain the next element of the list each time the iterator is called. Iterators often are used in polymorphic programming to traverse an array or a linked list of pointers to objects from various levels of a hierarchy. The pointers in such a list are all base-class pointers. (Chapter 23, Standard Template Library (STL), presents a thorough treatment of iterators.) A list of pointers to objects of base class `TwoDimensionalShape` could contain pointers to objects of classes `Square`, `Circle`, `triangle` and so on. Using polymorphism to send a `draw` message, off a `TwoDimensionalShape *` pointer, to each object in the list would draw each object correctly on the screen.

[Page 710 (continued)]

13.6. Case Study: Payroll System Using Polymorphism

This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section 12.4. In this example, we use an abstract class and polymorphism to perform payroll calculations based on the type of employee. We create an enhanced employee hierarchy to solve the following problem:

[Page 711]

> A company pays its employees weekly. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, 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. The classes that derive directly from `Employee` are `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee`. Class `BasePlusCommissionEmployee`derived from `CommissionEmployee`represents the last employee type. The UML class diagram in Fig. 13.11 shows the inheritance hierarchy for our polymorphic employee payroll application. Note that abstract class name `Employee` is italicized, as per the convention of the UML.

Figure 13.11. `Employee` hierarchy UML class diagram.

[View full size image]

Abstract base class `Employee` declares the "interface" to the hierarchythat 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`.

Software Engineering Observation 13.10

A derived class can inherit interface or implementation from a base class. Hierarchies designed for **implementation inheritance** tend to have their functionality high in the hierarchyeach 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 hierarchya 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).

The following sections implement the `Employee` class hierarchy. The first five each implement one of the abstract or concrete classes. The last section implements a test program that builds objects of all these classes and processes the objects polymorphically.

[Page 712]

13.6.1. Creating Abstract Base Class `Employee`

Class `Employee` (Figs. 13.1313.14, 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 generically 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 functionthere is not enough information to determine what amount `earnings` should return. Each derived class overrides `earnings` with an appropriate implementation. 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 classbecause of inheritance, however, all objects of all 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. 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.

Function `print` in class `Employee` displays the first name, last name and social security number of the employee. As we will 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.

The diagram in Fig. 13.12 shows each of the five classes in the hierarchy down the left side and functions `earnings` and `print` across the top. For each class, the diagram shows the desired results of each function. Note that 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. We do not list base class `Employee`'s get and set functions because they are not overridden in any of the derived classeseach of these functions is inherited and used "as is" by each of the derived classes.

Figure 13.12. Polymorphic interface for the Employee hierarchy classes.
(This item is displayed on page 713 in the print version)

[View full size image]

Let us consider class `Employee`'s header file (Fig. 13.13). The `public` member functions include a constructor that takes the first name, last name and social security number as arguments (line 12); set functions that set the first name, last name and social security number (lines 14, 17 and 20, respectively); get functions that return the first name, last name and social security number (lines 15, 18 and 21, respectively); pure `virtual` function `earnings` (line 24) and `virtual` function `print` (line 25).

Figure 13.13. `Employee` class header file.

(This item is displayed on pages 713 - 714 in the print version)

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
22
23    // pure virtual function makes Employee abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
```

```
29     string socialSecurityNumber;
30  }; // end class Employee
31
32  #endif // EMPLOYEE_H
```

Recall that we declared `earnings` as a pure `virtual` function because we first must know the specific `Employee` type to determine the appropriate `earnings` calculations. Declaring this function as pure `virtual` indicates that each concrete derived class must provide an appropriate `earnings` implementation and that a program can use base-class `Employee` pointers to invoke function `earnings` polymorphically for any type of `Employee`.

Figure 13.14 contains the member-function implementations for class `Employee`. No implementation is provided for `virtual` function `earnings`. Note that the `Employee` constructor (lines 1015) does not validate the social security number. Normally, such validation should be provided. An exercise in Chapter 12 asks you to validate a social security number to ensure that it is in the form `###-##-####`, where each # represents a digit.

[Page 714]

Figure 13.14. `Employee` class implementation file.
(This item is displayed on pages 714 - 715 in the print version)

```
 1  // Fig. 13.14: Employee.cpp
 2  // Abstract-base-class Employee member-function definitions.
 3  // Note: No definitions are given for pure virtual functions.
 4  #include <iostream>
 5  using std::cout;
 6
 7  #include "Employee.h" // Employee class definition
 8
 9  // constructor
10  Employee::Employee( const string &first, const string &last,
11     const string &ssn )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13  {
14     // empty body
15  } // end Employee constructor
16
17  // set first name
18  void Employee::setFirstName( const string &first )
19  {
20     firstName = first;
21  } // end function setFirstName
22
23  // return first name
24  string Employee::getFirstName() const
25  {
26     return firstName;
27  } // end function getFirstName
28
29  // set last name
30  void Employee::setLastName( const string &last )
31  {
32     lastName = last;
33  } // end function setLastName
34
35  // return last name
36  string Employee::getLastName() const
37  {
38     return lastName;
39  } // end function getLastName
```

```
40
41 // set social security number
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50    return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56    cout << getFirstName() << ' ' << getLastName()
57       << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print
```

[Page 715]

Note that `virtual` function `print` (Fig. 13.14, lines 5458) provides an implementation that will be overridden in each of the derived classes. Each of these functions will, however, use the abstract class's version of `print` to print information common to all classes in the `Employee` hierarchy.

13.6.2. Creating Concrete Derived Class `SalariedEmployee`

Class `SalariedEmployee` (Figs. 13.1513.16) derives from class `Employee` (line 8 of Fig. 13.15). The `public` member functions include a constructor that takes a first name, a last name, a social security number and a weekly salary as arguments (lines 1112); a set function to assign a new nonnegative value to data member `weeklySalary` (lines 14); a get function to return `weeklySalary`'s value (line 15); a `virtual` function `earnings` that calculates a `SalariedEmployee`'s earnings (line 18) and a `virtual` function `print` that outputs the employee's type, namely, `"salaried employee:"` followed by employee-specific information produced by base class `Employee`'s `print` function and `SalariedEmployee`'s `getWeeklySalary` function (line 19).

Figure 13.15. `SalariedEmployee` class header file.
(This item is displayed on page 716 in the print version)

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
```

```
21     double weeklySalary; // salary per week
22  }; // end class SalariedEmployee
23
24  #endif // SALARIED_H
```

Figure 13.16 contains the member-function implementations 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 3033) overrides pure `virtual` function `earnings` in `Employee` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` would be an abstract class, and any attempt to instantiate an object of the class would result in a compilation error (and, of course, we want `SalariedEmployee` here to be a concrete class). Note that in class `SalariedEmployee`'s header file, we declared member functions `earnings` and `print` as `virtual` (lines 1819 of Fig. 13.15)actually, placing the `virtual` keyword before these member functions is redundant. We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy. Recall from Good Programming Practice 13.1 that explicitly declaring such functions `virtual` at every level of the hierarchy can promote program clarity.

[Page 717]

Figure 13.16. `SalariedEmployee` class implementation file.
(This item is displayed on pages 716 - 717 in the print version)

```
1  // Fig. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
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    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25    return weeklySalary;
26 } // end function getWeeklySalary
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
```

```
38     cout << "salaried employee: ";
39     Employee::print(); // reuse abstract base-class print function
40     cout << "\nweekly salary: " << getWeeklySalary();
41  } // end function print
```

Function `print` of class `SalariedEmployee` (lines 3641 of Fig. 13.16) overrides `Employee` function `print`. If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`. In that case, `SalariedEmployee`'s `print` function would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To print a `SalariedEmployee`'s complete information, the derived class's `print` function outputs `"salaried employee:"` followed by the base-class `Employee`-specific information (i.e., first name, last name and social security number) printed by invoking the base class's `print` using the scope resolution operator (line 39)this is a nice example of code reuse. The output produced by `SalariedEmployee`'s `print` function contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` function.

13.6.3. Creating Concrete Derived Class `HourlyEmployee`

Class `HourlyEmployee` (Figs. 13.1713.18) also derives from class `Employee` (line 8 of Fig. 13.17). The `public` member functions include a constructor (lines 1112) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked; set functions that assign new values to data members `wage` and `hours`, respectively (lines 14 and 17); get functions to return the values of `wage` and `hours`, respectively (lines 15 and 18); a `virtual` function `earnings` that calculates an `HourlyEmployee`'s earnings (line 21) and a `virtual` function `print` that outputs the employee's type, namely, `"hourly employee:"` and employee-specific information (line 22).

[Page 718]

Figure 13.17. `HourlyEmployee` class header file.

```
 1  // Fig. 13.17: HourlyEmployee.h
 2  // HourlyEmployee class definition.
 3  #ifndef HOURLY_H
 4  #define HOURLY_H
 5
 6  #include "Employee.h" // Employee class definition
 7
 8  class HourlyEmployee : public Employee
 9  {
10  public:
11     HourlyEmployee( const string &, const string &,
12        const string &, double = 0.0, double = 0.0 );
13
14     void setWage( double ); // set hourly wage
15     double getWage() const; // return hourly wage
16
17     void setHours( double ); // set hours worked
18     double getHours() const; // return hours worked
19
20     // keyword virtual signals intent to override
21     virtual double earnings() const; // calculate earnings
22     virtual void print() const; // print HourlyEmployee object
23  private:
24     double wage; // wage per hour
25     double hours; // hours worked for week
26  }; // end class HourlyEmployee
27
28  #endif // HOURLY_H
```

[Page 719]

Figure 13.18 contains the member-function implementations for class `HourlyEmployee`. Lines 1821 and 3034 define set functions that assign new values to data members `wage` and `hours`, respectively. Function `setWage` (lines 1821) ensures that `wage` is nonnegative, and function `setHours` (lines 3034) ensures that data member `hours` is between 0 and 168 (the total number of hours in a week). Class `HourlyEmployee`'s get functions are implemented in lines 2427 and 3740. We do not declare these functions `virtual`, so classes derived from class `HourlyEmployee` cannot override them (although derived classes certainly can redefine them). Note that the `HourlyEmployee` constructor, like the `SalariedEmployee` constructor, passes the first name, last name and social security number to the base class `Employee` constructor (line 11) to initialize the inherited `private` data members declared in the base class. In addition, `HourlyEmployee`'s `print` function calls base-class function `print` (line 56) to output the `Employee`-specific information (i.e., first name, last name and social security number)this is another nice example of code reuse.

[Page 720]

Figure 13.18. **`HourlyEmployee` class implementation file.**
(This item is displayed on pages 718 - 719 in the print version)

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8  // constructor
9  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10    const string &ssn, double hourlyWage, double hoursWorked )
11    : Employee( first, last, ssn )
12 {
13    setWage( hourlyWage ); // validate hourly wage
14    setHours( hoursWorked ); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26    return wage;
27 } // end function getWage
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
```

```
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

13.6.4. Creating Concrete Derived Class `CommissionEmployee`

Class `CommissionEmployee` (Figs. 13.1913.20) derives from class `Employee` (line 8 of Fig. 13.19). The member-function implementations (Fig. 13.20) include a constructor (lines 915) that takes a first name, a last name, a social security number, a sales amount and a commission rate; set functions (lines 1821 and 3033) to assign new values to data members `commissionRate` and `grossSales`, respectively; get functions (lines 2427 and 3639) that retrieve the values of these data members; function `earnings` (lines 4346) to calculate a `CommissionEmployee`'s earnings; and function `print` (lines 4955), which outputs the employee's type, namely, `"commission employee:"` and employee-specific information. The `CommissionEmployee`'s constructor also 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 52) to display the `Employee`-specific information (i.e., first name, last name and social security number).

[Page 722]

Figure 13.19. `CommissionEmployee` class header file.
(This item is displayed on page 720 in the print version)

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
```

```
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

Figure 13.20. **`CommissionEmployee`** **class implementation file.**

(This item is displayed on pages 720 - 721 in the print version)

```
 1 // Fig. 13.20: CommissionEmployee.cpp
 2 // CommissionEmployee class member-function definitions.
 3 #include <iostream>
 4 using std::cout;
 5
 6 #include "CommissionEmployee.h" // CommissionEmployee class definition
 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 commission rate
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26    return commissionRate;
27 } // end function getCommissionRate
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45    return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
```

```
54         << "; commission rate: " << getCommissionRate();
55   } // end function print
```

13.6.5. Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployee` (Figs. 13.2113.22) directly inherits from class `CommissionEmployee` (line 8 of Fig. 13.21) and therefore is an indirect derived class of class `Employee`. Class `BasePlusCommissionEmployee`'s member-function implementations include a constructor (lines 1016 of Fig. 13.22) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes the first name, last name, social security number, sales amount and commission rate to the `CommissionEmployee` constructor (line 13) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a set function (lines 1922) to assign a new value to data member `baseSalary` and a get function (lines 2528) to return `baseSalary`'s value. Function `earnings` (lines 3235) calculates a `BasePlusCommissionEmployee`'s earnings. Note that line 34 in function `earnings` calls base-class `CommissionEmployee`'s `earnings` function to calculate the commission-based portion of the employee's earnings. This is a nice example of code reuse. `BasePlusCommissionEmployee`'s `print` function (lines 3843) outputs `"base-salaried"`, followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary. The resulting output begins with `"base-salaried commission employee:"` followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `print` displays the employee's first name, last name and social security number by invoking the `print` function of its base class (i.e., `Employee`)yet another example of code reuse. Note that `BasePlusCommissionEmployee`'s `print` initiates a chain of functions calls that spans all three levels of the `Employee` hierarchy.

[Page 723]

Figure 13.21. `BasePlusCommissionEmployee` **class header file.**
(This item is displayed on page 722 in the print version)

```
 1  // Fig. 13.21: BasePlusCommissionEmployee.h
 2  // BasePlusCommissionEmployee class derived from Employee.
 3  #ifndef BASEPLUS_H
 4  #define BASEPLUS_H
 5
 6  #include "CommissionEmployee.h" // CommissionEmployee class definition
 7
 8  class BasePlusCommissionEmployee : public CommissionEmployee
 9  {
10  public:
11     BasePlusCommissionEmployee( const string &, const string &,
12        const 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     // keyword virtual signals intent to override
18     virtual double earnings() const; // calculate earnings
19     virtual void print() const; // print BasePlusCommissionEmployee object
20  private:
21     double baseSalary; // base salary per week
22  }; // end class BasePlusCommissionEmployee
23
24  #endif // BASEPLUS_H
```

Figure 13.22. `BasePlusCommissionEmployee` **class implementation file.**

```
1  // Fig. 13.22: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
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 {
15    setBaseSalary( salary ); // validate and store base salary
16 } // end BasePlusCommissionEmployee constructor
17
18 // set base salary
19 void BasePlusCommissionEmployee::setBaseSalary( double salary )
20 {
21    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
22 } // end function setBaseSalary
23
24 // return base salary
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27     return baseSalary;
28 } // end function getBaseSalary
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41    CommissionEmployee::print(); // code reuse
42    cout << "; base salary: " << getBaseSalary();
43 } // end function print
```

[Page 724]

13.6.6. Demonstrating Polymorphic Processing

To test our `Employee` hierarchy, the program in Fig. 13.23 creates an object of each of the four concrete classes `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`. The program manipulates these objects, first with static binding, then polymorphically, using a `vector` of `Employee` pointers. Lines 3138 create objects of each of the four concrete `Employee` derived classes. Lines 4351 output each `Employee`'s information and earnings. Each member-function invocation in lines 4351 is an example of static bindingat 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.

[Page 727]

Figure 13.23. `Employee` class hierarchy driver program.

(This item is displayed on pages 724 - 727 in the print version)

```
// Fig. 13.23: fig13_23.cpp
// Processing Employee derived-class objects individually
// and polymorphically using dynamic binding.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include <vector>
using std::vector;

// include definitions of classes in Employee hierarchy
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

void virtualViaPointer( const Employee * const ); // prototype
void virtualViaReference( const Employee & ); // prototype

int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create derived-class objects
   SalariedEmployee salariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   HourlyEmployee hourlyEmployee(
      "Karen", "Price", "222-22-2222", 16.75, 40 );
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   cout << "Employees processed individually using static binding:\n\n";

   // output each Employee's information and earnings using static binding
   salariedEmployee.print();
   cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
   hourlyEmployee.print();
   cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
   commissionEmployee.print();
   cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
   basePlusCommissionEmployee.print();
   cout << "\nearned $" << basePlusCommissionEmployee.earnings()
      << "\n\n";

   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

   // initialize vector with Employees
   employees[ 0 ] = &salariedEmployee;
   employees[ 1 ] = &hourlyEmployee;
   employees[ 2 ] = &commissionEmployee;
   employees[ 3 ] = &basePlusCommissionEmployee;

   cout << "Employees processed polymorphically via dynamic binding:\n\n";

   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class pointers:\n\n";
```

```
67
68     for ( size_t i = 0; i < employees.size(); i++ )
69        virtualViaPointer( employees[ i ] );
70
71     // call virtualViaReference to print each Employee's information
72     // and earnings using dynamic binding
73     cout << "Virtual function calls made off base-class references:\n\n";
74
75     for ( size_t i = 0; i < employees.size(); i++ )
76        virtualViaReference( *employees[ i ] ); // note dereferencing
77
78     return 0;
79  } // end main
80
81  // call Employee virtual functions print and earnings off a
82  // base-class pointer using dynamic binding
83  void virtualViaPointer( const Employee * const baseClassPtr )
84  {
85     baseClassPtr->print();
86     cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87  } // end function virtualViaPointer
88
89  // call Employee virtual functions print and earnings off a
90  // base-class reference using dynamic binding
91  void virtualViaReference( const Employee &baseClassRef )
92  {
93     baseClassRef.print();                                    '
94     cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95  } // end function virtualViaReference
```

```
Employees processed individually using static binding:

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00
```

```
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

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

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
```

Line 54 allocates `vector employees`, which contains four `Employee` pointers. Line 57 aims `employees[0]` at object `salariedEmployee`. Line 58 aims `employees[1]` at object `hourlyEmployee`. Line 59 aims `employees[2]` at object `commissionEmployee`. Line 60 aims `employee[3]` at object `basePlusCommissionEmployee`. The compiler allows these assignments, because a `SalariedEmployee` is an `Employee`, an `HourlyEmployee` is an `Employee`, a `CommissionEmployee` is an `Employee` and a `BasePlusCommissionEmployee` is an `Employee`. Therefore, we can assign the addresses of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to base-class `Employee` pointers (even though `Employee` is an abstract class).

The `for` statement at lines 6869 traverses `vector employees` and invokes function `virtualViaPointer` (lines 8387) 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 85) and `earnings` (line 86). Note that function `virtualViaPointer` does not contain any `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` or `BasePlusCommissionEmployee` type information. The function knows only about base-class type `Employee`. Therefore, at compile time, the compiler cannot know which concrete class's functions to call through `baseClassPtr`. Yet at execution time, each virtual-function invocation calls the function on the object to which `baseClassPtr` points at that time. The output illustrates that the appropriate functions for each class are indeed invoked and that each object's proper information is displayed. For instance, the weekly salary is displayed for the `SalariedEmployee`, and the gross sales are displayed for the `CommissionEmployee` and `BasePlusCommissionEmployee`. Also note that obtaining the earnings of each `Employee` polymorphically in line 86 produces the same results as obtaining these employees' earnings via static binding in lines 44, 46, 48 and 50. All `virtual` function calls to `print` and `earnings` are resolved at runtime with dynamic binding.

Finally, another `for` statement (lines 7576) traverses `employees` and invokes function `virtualViaReference` (lines 9195) for each element in the `vector`. Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a reference formed by dereferencing the pointer stored in each `employees` element (line 76). Each call to `virtualViaReference` invokes `virtual` functions `print` (line 93) and `earnings` (line 94) via reference `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.

[Page 728]

[Page 728 (continued)]

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

C++ makes polymorphism easy to program. It is certainly possible to program for polymorphism in non-object-oriented languages such as C, but doing so requires complex and potentially dangerous pointer manipulations. 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 polymorphismin terms of additional memory consumption and processor time. This will help you determine when to use polymorphism and when to avoid it. As you will see in Chapter 23, Standard Template Library (STL), the STL components were implemented without polymorphism and `virtual` functionsthis was done to avoid the associated execution-time overhead and achieve optimal performance to meet the unique requirements of the STL.

First, we will explain the data structures that the C++ compiler builds at compile time to support polymorphism at execution time. You will see that polymorphism is accomplished through three levels of pointers (i.e., "triple indirection"). Then we will show how an executing program uses these data structures to execute `virtual` functions and achieve the dynamic binding associated with polymorphism. Note that our discussion explains one possible implementation; this is not a language requirement.

When C++ compiles a class that has one or more `virtual` functions, it builds a **virtual function table (vtable)** for that class. 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. 13.24 illustrates the vtables for classes `Employee`, `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.

Figure 13.24. How `virtual` function calls work.
(This item is displayed on page 729 in the print version)

[View full size image]

In the vtable for class `Employee`, the first function pointer is set to 0 (i.e., the null pointer). This is done 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. [Note: We have abbreviated the output of each `print` function in this figure to conserve space.] Any class that has one or more null pointers in its vtable is an abstract class. Classes without any null vtable pointers (such as `SalariedEmployee, HourlyEmployee, CommissionEmployee` and `BasePlusCommissionEmployee`) are concrete classes.

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 `"salariedemployee:"` followed by the employee's name, social security number and weekly salary.

The `earnings` function pointer in the vtable for class `HourlyEmployee` points to `HourlyEmployee`'s `earnings` function that returns the employee's `wage` multiplied by the number of `hours` worked. Note that to

conserve space, we have omitted the fact that hourly employees receive time-and-a-half pay for overtime hours worked. The `print` function pointer points to the `HourlyEmployee` version of the function, which prints `"hourly employee:"`, the employee's name, social security number, hourly wage and hours worked. Both functions override the functions in class `Employee`.

[Page 730]

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 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 `HourlyEmployee`, both functions override the functions in class `Employee`.

The `earnings` function pointer in the vtable for class `BasePlusCommissionEmployee` points to `BasePlusCommissionEmployee`'s `earnings` function that 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`.

Notice that in our `Employee` case study, each concrete class provides its own implementation for `virtual` functions `earnings` and `print`. You have already learned that each class which inherits directly from abstract base class `Employee` must implement `earnings` in order to be a concrete class, because `earnings` is a pure `virtual` function. These classes do not need to implement function `print`, however, to be considered concrete`print` is not a pure `virtual` function and derived classes can inherit class `Employee`'s implementation of `print`. Furthermore, class `BasePlusCommissionEmployee` does not have to implement either function `print` or `earnings`both function implementations can be inherited from class `CommissionEmployee`. If a class in our hierarchy were to inherit function implementations in this manner, the vtable pointers for these functions would simply point to the function implementation that was being inherited. For example, if `BasePlusCommissionEmployee` did not override `earnings`, the `earnings` function pointer in the vtable for class `BasePlusCommissionEmployee` would point to the same `earnings` function as the vtable for class `CommissionEmployee` points to.

Polymorphism is accomplished through an elegant data structure involving three levels of pointers. We have discussed one levelthe 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 is not required to be implemented that way. In Fig. 13.24, these pointers are associated with the objects created in Fig. 13.23 (one object for each of the types `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`). Notice that the diagram displays each of the object's data member values. For example, the `salariedEmployee` object contains a pointer to the `SalariedEmployee` vtable; the object also contains the values `John Smith`, `111-11-1111` and `$800.00`.

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. Note that Fig. 13.24 depicts the `vector employees` that contains `Employee` pointers.

Now let us see how a typical `virtual` function call executes. Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 85 of Fig. 13.23). Assume that `baseClassPtr` contains `employees[ 1 ]` (i.e., the address of object `hourlyEmployee` 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.

[Page 731]

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. 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.

The compiler generates code that performs the following operations [Note: The numbers in the list correspond to the circled numbers in Fig. 13.24]:

1. Select the i^{th} entry of `employees` (in this case, the address of object `hourlyEmployee`), and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `hourlyEmployee`.

2. Dereference that pointer to get to the `hourlyEmployee` objectwhich, as you recall, begins with a pointer to the `HourlyEmployee` vtable.

3. Dereference `hourlyEmployee`'s vtable pointer to get to the `HourlyEmployee` 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, which in this case prints the employee's type, name, social security number, hourly wage and hours worked.

The data structures of Fig. 13.24 may appear to be complex, but this complexity is managed by the compiler and hidden from you, making polymorphic programming straightforward. The pointer dereferencing operations and memory accesses that occur on every `virtual` function call require some additional execution time. The vtables and the vtable pointers added to the objects require some additional memory. You now have enough information to determine whether `virtual` functions are appropriate for your programs.

Performance Tip 13.1

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

Performance Tip 13.2

Virtual functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that runs as efficiently as hand-coded `switch`-based logic. The overhead of polymorphism is acceptable for most applications. But in some situationsreal-time applications with stringent performance requirements, for examplethe overhead of polymorphism may be too high.

Software Engineering Observation 13.11

Dynamic binding enables independent software vendors (ISVs) to distribute software without revealing proprietary secrets. Software distributions can consist of only header files and object filesno source code needs to be revealed. Software developers can then use inheritance to derive new classes from those provided by the ISVs. Other software that worked with the classes the ISVs provided will still work with the

derived classes and will use the overridden `virtual` functions provided in these classes (via dynamic binding).

[Page 732]

13.8. Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

Recall from the problem statement at the beginning of Section 13.6 that, for the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployees` by adding 10 percent to their base salaries. When processing `Employee` objects polymorphically in Section 13.6.6, we did not need to worry about the "specifics." Now, however, to adjust the base salaries of `BasePlusCommissionEmployees`, we have to determine the specific type of each `Employee` object at execution time, then act appropriately. This section demonstrates the powerful capabilities of run-time type information (RTTI) and dynamic casting, which enable a program to determine the type of an object at execution time and act on that object accordingly.

Some compilers, such as Microsoft Visual C++ .NET, require that RTTI be enabled before it can be used in a program. Consult your compiler's documentation to determine whether your compiler has similar requirements. To enable RTTI in Visual C++ .NET, select the Project menu and then select the properties option for the current project. In the Property Pages dialog box that appears, select Configuration Properties > C/C++ > Language. Then choose Yes (/GR) from the combo box next to Enable Run-Time Type Info. Finally, click OK to save the settings.

The program in Fig. 13.25 uses the `Employee` hierarchy developed in Section 13.6 and increases by 10 percent the base salary of each `BasePlusCommissionEmployee`. Line 31 declares four-element `vector employees` that stores pointers to `Employee` objects. Lines 3441 populate the `vector` with the addresses of dynamically allocated objects of classes `SalariedEmployee` (Figs. 13.1513.16), `HourlyEmployee` (Figs. 13.1713.18), `CommissionEmployee` (Figs. 13.1913.20) and `BasePlusCommissionEmployee` (Figs. 13.2113.22).

[Page 734]

Figure 13.25. Demonstrating downcasting and run-time type information.
(This item is displayed on pages 732 - 734 in the print version)

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and run-time type information.
3  // NOTE: For this example to run in Visual C++ .NET,
4  // you need to enable RTTI (Run-Time Type Info) for the project.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
```

```
19  #include "Employee.h"
20  #include "SalariedEmployee.h"
21  #include "HourlyEmployee.h"
22  #include "CommissionEmployee.h"
23  #include "BasePlusCommissionEmployee.h"
24
25  int main()
26  {
27     // set floating-point output formatting
28     cout << fixed << setprecision( 2 );
29
30     // create vector of four base-class pointers
31     vector < Employee * > employees( 4 );
32
33     // initialize vector with various kinds of Employees
34     employees[ 0 ] = new SalariedEmployee(
35        "John", "Smith", "111-11-1111", 800 );
36     employees[ 1 ] = new HourlyEmployee(
37        "Karen", "Price", "222-22-2222", 16.75, 40 );
38     employees[ 2 ] = new CommissionEmployee(
39        "Sue", "Jones", "333-33-3333", 10000, .06 );
40     employees[ 3 ] = new BasePlusCommissionEmployee(
41        "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43     // polymorphically process each element in vector employees
44     for ( size_t i = 0; i < employees.size(); i++ )
45     {
46        employees[ i ]->print(); // output employee information
47        cout << endl;
48
49        // downcast pointer
50        BasePlusCommissionEmployee *derivedPtr =
51           dynamic_cast < BasePlusCommissionEmployee * >
52              ( employees[ i ] );
53
54        // determine whether element points to base-salaried
55        // commission employee
56        if ( derivedPtr !=0 ) // 0 if not a BasePlusCommissionEmployee
57        {
58           double oldBaseSalary = derivedPtr->getBaseSalary();
59           cout << "old base salary: $" << oldBaseSalary << endl;
60           derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
61           cout << "new base salary with 10% increase is: $"
62              << derivedPtr->getBaseSalary() << endl;
63        } // end if
64
65        cout << "earned $" << employees[ i ]->earnings() << "\n\n";
66     } // end for
67
68     // release objects pointed to by vector's elements
69     for ( size_t j = 0; j < employees.size(); j++ )
70     {
71        // output class name
72        cout << "deleting object of "
73           << typeid( *employees[ j ] ).name() << endl;
74
75        delete employees[ j ];
76     } // end for
77
78     return 0;
79  } // end main
```

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

```
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

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 HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

The `for` statement at lines 4466 iterates through the `employees vector` and displays each `Employee`'s information by invoking member function `print` (line 46). Recall that because `print` is declared `virtual` in base class `Employee`, the system invokes the appropriate derived-class object's `print` function.

In this example, as we encounter `BasePlusCommissionEmployee` objects, we wish to increase their base salary by 10 percent. Since we process the employees generically (i.e., polymorphically), we cannot (with the techniques we've learned) be certain as to which type of `Employee` is being manipulated at any given time. This creates a problem, because `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 51) to determine whether the type of each object is `BasePlusCommissionEmployee`. This is the downcast operation we referred to in Section 13.3.3. Lines 5052 dynamically downcast `employees[i]` from type `Employee *` to type `BasePlusCommissionEmployee *`. If the `vector` element points to an object that is a `BasePlusCommissionEmployee` object, then that object's address is assigned to `commissionPtr`; otherwise, `0` is assigned to derived-class pointer `derivedPtr`.

If the value returned by the `dynamic_cast` operator in lines 5052 is not `0`, the object is the correct type and the `if` statement (lines 5663) performs the special processing required for the `BasePlusCommissionEmployee` object. Lines 58, 60 and 62 invoke `BasePlusCommissionEmployee` functions `getBaseSalary` and `setBaseSalary` to retrieve and update the employee's salary.

[Page 735]

Line 65 invokes member function `earnings` on the object to which `employees[i]` points. Recall that `earnings` is declared `virtual` in the base class, so the program invokes the derived-class object's `earnings` functionanother example of dynamic binding.

The `for` loop at lines 6976 displays each employee's object type and uses the `delete` operator to deallocate the dynamic memory to which each `vector` element points. Operator **`typeid`** (line 73) 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 73) returns a pointer-based string that contains the type name (e.g., `"class BasePlusCommissionEmployee"`) of the argument passed to `typeid`.

[Note: The exact contents of the string returned by `type_info` member function `name` may vary by compiler.] To use `typeid`, the program must include header file **`<typeinfo>`** (line 16).

Note that we avoid several compilation errors in this example by downcasting an `Employee` pointer to a `BasePlusCommissionEmployee` pointer (lines 5052). If we remove the `dynamic_cast` from line 51 and attempt to assign the current `Employee` pointer directly to `BasePlusCommissionEmployee` pointer `commissionPtr`, we will receive a compilation error. C++ does not allow a program to assign a base-class pointer to a derived-class pointer because the is-a relationship does not applya `CommissionEmployee` is not a `BasePlusCommissionEmployee`. The is-a relationship applies only between the derived class and its base classes, not vice versa.

Similarly, if lines 58, 60 and 62 used the current base-class pointer from `employees`, rather than derived-class pointer `commissionPtr`, to invoke derived-class-only functions `getBaseSalary` and `setBaseSalary`, we would receive a compilation error at each of these lines. As you learned in Section 13.3.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed. Although lines 58, 60 and 62 execute only if `commissionPtr` is not `0` (i.e., if the cast can be performed), we cannot attempt to invoke derived class `BasePlusCommissionEmployee` functions `getBaseSalary` and `setBaseSalary` on the base class `Employee` pointer. Recall that, using a base class `Employee` pointer, we can invoke only functions found in base class `Employeeearnings`, `print` and `Employee`'s get and set functions.

[Page 735 (continued)]

13.9. Virtual Destructors

A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy. So far you have seen **nonvirtual destructors**destructors that are not declared with keyword `virtual`. If a derived-class object with a nonvirtual destructor is destroyed explicitly 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 **`virtual destructor`** (i.e., a destructor that is declared with keyword `virtual`) in the base class. This makes all derived-class destructors `virtual` even though they do not have the same name as the base-class destructor. 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 is important for the destructors of both the derived class and base class to execute. The base-class destructor automatically executes after the derived-class destructor.

[Page 736]

Good Programming Practice 13.2

If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class. Classes derived from this class may contain destructors that must be called properly.

Common Programming Error 13.5

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

[Page 736 (continued)]

13.10. (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for commonality among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more efficient and elegant manner that enables us to process objects of these classes polymorphically. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into C++ header files.

In Section 3.11, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes`BalanceInquiry`, `Withdrawal` and `Deposit`to represent the transactions that the ATM system can perform. Figure 13.26 shows the attributes and operations of these classes. Note that they have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent types of transactions. Figure 13.26 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing these classes. We place the common functionality in base class `TRansaction` and derive classes `BalanceInquiry`, `Withdrawal` and `Deposit` from `transaction` (Fig. 13.27).

Figure 13.26. Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

[View full size image]

Figure 13.27. Class diagram modeling generalization relationship between base class `TRansaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.
(This item is displayed on page 737 in the print version)

[View full size image]

[Page 737]

The UML specifies a relationship called a generalization to model inheritance. Figure 13.27 is the class diagram that models the inheritance relationship between base class `transaction` and its three derived classes. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` are derived from class `transaction`. Class `transaction` is said to be a generalization of its derived classes. The derived classes are said to be specializations of class `TRansaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in base class `TRansaction`. We no longer list `accountNumber` in the second compartment of each derived class, because the three derived classes inherit this attribute from `transaction`. Recall, however, that derived classes cannot access `private` attributes of a base class. We therefore include `public` member function `getAccountNumber` in class `TRansaction`. Each derived class inherits this member function, enabling the derived class to access its `accountNumber` as needed to execute a transaction.

According to Fig. 13.26, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so base class `transaction` should contain `public` member function `execute`. However, it does not make sense to implement `execute` in class `transaction`, because the functionality that this member function provides depends on the specific type of the actual transaction. We therefore declare member function `execute` as a pure `virtual` function in base class `transaction`. This makes `transaction` an abstract class and forces any class derived from `transaction` that must be a concrete class (i.e., `BalanceInquiry`, `Withdrawal` and `Deposit`) to implement pure `virtual` member function `execute` to make the derived class concrete. The UML requires that we place abstract class names (and pure `virtual` functionsabstract operations in the UML) in italics, so `transaction` and its member function `execute` appear in italics in Fig. 13.27. Note that operation `execute` is not italicized in derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Each derived class overrides base class `transaction`'s `execute` member function with an appropriate implementation. Note that Fig. 13.27 includes operation execute in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden member function.

[Page 738]

As you learned in this chapter, a derived class can inherit interface or implementation from a base class. Compared to a hierarchy designed for implementation inheritance, one designed for interface inheritance tends to have its functionality lower in the hierarchya base class signifies one or more functions that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the function(s). The inheritance hierarchy designed for the `ATM` system takes advantage of this type of inheritance, which provides the `ATM` with an elegant way to execute all transactions "in the general." Each class derived from `transaction` inherits some implementation details (e.g., data member `accountNumber`), but the primary benefit of incorporating inheritance into our system is that the derived classes share a common interface (e.g., pure `virtual` member function `execute`). The `ATM` can aim a

transaction pointer at any transaction, and when the ATM invokes execute tHRough this pointer, the version of execute appropriate to that transaction (i.e., implemented in that derived class's .cpp file) runs automatically. For example, suppose a user chooses to perform a balance inquiry. The ATM aims a transaction pointer at a new object of class BalanceInquiry, which the C++ compiler allows because a BalanceInquiry is a transaction. When the ATM uses this pointer to invoke execute, BalanceInquiry's version of execute is called.

This polymorphic approach also makes the system easily extensible. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional TRansaction derived class that overrides the execute member function with a version appropriate for the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type from the main menu and for the ATM to instantiate and execute objects of the new derived class. The ATM could execute transactions of the new type using the current code, because it executes all transactions identically.

As you learned earlier in the chapter, an abstract class like TRansaction is one for which the programmer never intends to instantiate objects. An abstract class simply declares common attributes and behaviors for its derived classes in an inheritance hierarchy. Class TRansaction defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include pure virtual member function execute in class TRansaction if execute lacks a concrete implementation. Conceptually, we include this member function because it is the defining behavior of all transactionsexecuting. Technically, we must include member function execute in base class TRansaction so that the ATM (or any other class) can polymorphically invoke each derived class's overridden version of this function through a TRansaction pointer or reference.

Derived classes BalanceInquiry, Withdrawal and Deposit inherit attribute accountNumber from base class transaction, but classes Withdrawal and Deposit contain the additional attribute amount that distinguishes them from class BalanceInquiry. Classes Withdrawal and Deposit require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class BalanceInquiry has no need for such an attribute and requires only an account number to execute. Even though two of the three transaction derived classes share this attribute, we do not place it in base class transactionwe place only features common to all the derived classes in the base class, so derived classes do not inherit unnecessary attributes (and operations).

Figure 13.28 presents an updated class diagram of our model that incorporates inheritance and introduces class TRansaction. We model an association between class ATM and class transaction to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type TRansaction exist in the system at a time). Because a Withdrawal is a type of TRansaction, we no longer draw an association line directly between class ATM and class Withdrawalderived class Withdrawal inherits base class transaction's association with class ATM. Derived classes BalanceInquiry and Deposit also inherit this association, which replaces the previously omitted associations between classes BalanceInquiry and Deposit and class ATM. Note again the use of triangular hollow arrowheads to indicate the specializations of class TRansaction, as indicated in Fig. 13.27.

[Page 739]

Figure 13.28. Class diagram of the ATM system (incorporating inheritance). Note that abstract class name transaction appears in italics.

[View full size image]

We also add an association between class `TRansaction` and the `BankDatabase` (Fig. 13.28). All `TRansactions` require a reference to the `BankDatabase` so they can access and modify account information. Each `TRansaction` derived class inherits this reference, so we no longer model the association between class `Withdrawal` and the `BankDatabase`. Note that the association between class `transaction` and the `BankDatabase` replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and the `BankDatabase`.

We include an association between class `transaction` and the `Screen` because all `transactions` display output to the user via the `Screen`. Each derived class inherits this association. Therefore, we no longer include the association previously modeled between `Withdrawal` and the `Screen`. Class `Withdrawal` still participates in associations with the `CashDispenser` and the `Keypad`, howeverthese associations apply to derived class `Withdrawal` but not to derived classes `BalanceInquiry` and `Deposit`, so we do not move these associations to base class `transaction`.

[Page 740]

Our class diagram incorporating inheritance (Fig. 13.28) also models `Deposit` and `BalanceInquiry`. We show associations between `Deposit` and both the `DepositSlot` and the `Keypad`. Note that class `BalanceInquiry` takes part in no associations other than those inherited from class `transaction`a `BalanceInquiry` interacts only with the `BankDatabase` and the `Screen`.

The class diagram of Fig. 9.20 showed attributes and operations with visibility markers. Now we present a modified class diagram in Fig. 13.29 that includes abstract base class `transaction`. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 13.28), but instead shows the attributes and operations after we have employed inheritance in our system. Note that abstract class name `transaction` and abstract operation name `execute` in class `transaction` appear in italics. To save space, as we did in Fig. 4.24, we do not include those attributes shown by associations in Fig. 13.28we do, however, include them in the C++ implementation in Appendix G. We also omit all operation parameters, as we did in Fig. 9.20incorporating inheritance does not affect the parameters already modeled in Figs. 6.226.25.

[Page 741]

Figure 13.29. Class diagram after incorporating inheritance into the system.
(This item is displayed on page 740 in the print version)

[View full size image]

Software Engineering Observation 13.12

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Fig. 13.28 and Fig. 13.29), a good practice that promotes readability is to divide this information between two class diagramsone focusing on associations and the other on attributes and operations. However, when examining classes modeled in this fashion, it is crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 13.28 to observe the inheritance relationship between `transaction` and its derived classes that is omitted from Fig. 13.29.

Implementing the ATM System Design Incorporating Inheritance

In Section 9.12, we began implementing the ATM system design in C++ code. We now modify our implementation to incorporate inheritance, using class `Withdrawal` as an example.

1. If a class `A` is a generalization of class `B`, then class `B` is derived from (and is a specialization of) class `A`. For example, abstract base class `TRansaction` is a generalization of class `Withdrawal`. Thus, class `Withdrawal` is derived from (and is a specialization of) class `transaction`. Figure 13.30 contains a portion of class `Withdrawal`'s header file, in which the class definition indicates the inheritance relationship between `Withdrawal` and `transaction` (line 9).

Figure 13.30. `Withdrawal` **class definition that derives from** `TRansaction`**.**
(This item is displayed on page 742 in the print version)

```
1  // Fig. 13.30: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  // class Withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

2. If class `A` is an abstract class and class `B` is derived from class `A`, then class `B` must implement the pure `virtual` functions of class `A` if class `B` is to be a concrete class. For example, class `transaction` contains pure `virtual` function `execute`, so class `Withdrawal` must implement this member function if we want to instantiate a `Withdrawal` object. Figure 13.31 contains the C++ header file for class `Withdrawal` from Fig. 13.28 and Fig. 13.29. Class `Withdrawal` inherits data member `accountNumber` from base class `transaction`, so `Withdrawal` does not declare this data member. Class `Withdrawal` also inherits references to the `Screen` and the `BankDatabase` from its base class `transaction`, so we do not include these references in our code. Figure 13.29 specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 19 of Fig. 13.31 declares a data member for attribute `amount`. Line 16 contains the function prototype for operation `execute`. Recall that, to be a concrete class, derived class `Withdrawal` must provide a concrete implementation of the pure `virtual` function `execute` in base class `transaction`. The prototype in line 16 signals your intent to override the base class pure `virtual` function. You must provide this prototype if you will provide an implementation in the `.cpp` file. We present this implementation in Appendix G. The `keypad` and `cashDispenser` references (lines 2021) are data members derived from `Withdrawal`'s associations in Fig. 13.28. In the implementation of this class in Appendix G, a constructor initializes these references to actual objects. Once again, to be able to compile the declarations of the references in lines 2021, we include the forward declarations in lines 89.

[Page 742]

Figure 13.31. `Withdrawal` **class header file based on Fig. 13.28 and Fig. 13.29.**

```
1  // Fig. 13.31: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 // class Withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    // member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
```

```
20     Keypad &keypad; // reference to ATM's keypad
21     CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22  }; // end class Withdrawal
23
24  #endif // WITHDRAWAL_H
```

ATM Case Study Wrap-Up

This concludes our object-oriented design of the ATM system. A complete C++ implementation of the ATM system in 877 lines of code appears in Appendix G. This working implementation uses key programming notions, including classes, objects, encapsulation, visibility, composition, inheritance and polymorphism. The code is abundantly commented and conforms to the coding practices you've learned. Mastering this code is a wonderful capstone experience for you after studying Chapters 1213.

[Page 743]

Software Engineering Case Study Self-Review Exercises

13.1 The UML uses an arrow with a _____ to indicate a generalization relationship.

- **a.** solid filled arrowhead
- **b.** triangular hollow arrowhead
- **c.** diamond-shaped hollow arrowhead
- **d.** stick arrowhead

13.2 State whether the following statement is true or false, and if false, explain why: The UML requires that we underline abstract class names and operation names.

13.3 Write a C++ header file to begin implementing the design for class `transaction` specified in Fig. 13.28 and Fig. 13.29. Be sure to include `private` references based on class `TRansaction`'s associations. Also be sure to include `public` get functions for any of the `private` data members that the derived classes must access to perform their tasks.

Answers to Software Engineering Case Study Self-Review Exercises

13.1 b.

13.2 False. The UML requires that we italicize abstract class names and operation names.

13.3 The design for class `transaction` yields the header file in Fig. 13.32. In the implementation in Appendix G, a constructor initializes `private` reference attributes `screen` and `bankDatabase` to actual objects, and member functions `getScreen` and `getBankDatabase` access these attributes. These member functions allow classes derived from `TRansaction` to access the ATM's screen and interact with the bank's database.

Figure 13.32. `transaction` class header file based on Fig. 13.28 and Fig. 13.29.

```
1  // Fig. 13.32: Transaction.h
```

```
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   int getAccountNumber(); // return account number
   Screen &getScreen(); // return reference to screen
   BankDatabase &getBankDatabase(); // return reference to bank database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```


[Page 744]

13.11. Wrap-Up

In this chapter we discussed polymorphism, which enables us to "program in the general" rather than "program in the specific," and we showed how this makes programs more extensible. We began with an example of how polymorphism would allow a screen manager to display several "space" objects. We then demonstrated how base-class and derived-class pointers can be aimed at base-class and derived-class objects. We said that aiming base-class pointers at base-class objects is natural, as is aiming derived-class pointers at derived-class objects. Aiming base-class pointers at derived-class objects is also natural because a derived-class object is an object of its base class. You learned why aiming derived-class pointers at base-class objects is dangerous and why the compiler disallows such assignments. We introduced `virtual` functions, which enable the proper functions to be called when objects at various levels of an inheritance hierarchy are referenced (at execution time) via base-class pointers. This is known as dynamic or late binding. We then discussed pure `virtual` functions (`virtual` functions that do not provide an implementation) and abstract classes (classes with one or more pure `virtual` functions). You learned that abstract classes cannot be used to instantiate objects, while concrete classes can. We then demonstrated using abstract classes in an inheritance hierarchy. You learned how polymorphism works "under the hood" with vtables that are created by the compiler. We discussed downcasting base-class pointers to derived-class pointers to enable a program to call derived-class-only member functions. The chapter concluded with a discussion of `virtual` destructors, and how they ensure that all appropriate destructors in an inheritance hierarchy run on a derived-class object when that object is deleted via a base-class pointer.

In the next chapter, we discuss templates, a sophisticated feature of C++ that enables programmers to define a family of related classes or functions with a single code segment.


[Page 744 (continued)]

Summary

- With `virtual` functions and polymorphism, it becomes possible to design and implement systems that are more easily extensible. Programs can be written to process objects of types that may not exist when the program is under development.
- Polymorphic programming with `virtual` functions can eliminate the need for `switch` logic. The programmer can use the `virtual` function mechanism to perform the equivalent logic automatically, thus avoiding the kinds of errors typically associated with `switch` logic.
- Derived classes can provide their own implementations of a base-class `virtual` function if necessary, but if they do not, the base class's implementation is used.
- If a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator, the reference is resolved at compile time (this is called static binding); the `virtual` function that is called is the one defined for the class of that particular object.
- In many situations it is useful to define abstract classes for which the programmer never intends to create objects. Because these are used only as base classes, we refer to them as abstract base classes. No objects of an abstract class may be instantiated.
- Classes from which objects can be instantiated are called concrete classes.
- A class is made abstract by declaring one or more of its `virtual` functions to be pure. A pure `virtual` function is one with a pure specifier (`= 0`) in its declaration.

[Page 745]

- If a class is derived from a class with a pure `virtual` function and that derived class does not supply a definition for that pure `virtual` function, then that `virtual` function remains pure in the derived class. Consequently, the derived class is also an abstract class.
- C++ enables polymorphismthe ability for objects of different classes related by inheritance to respond differently to the same member-function call.
- Polymorphism is implemented via `virtual` functions and dynamic binding.
- When a request is made through a base-class pointer or reference to use a `virtual` function, C++ chooses the correct overridden function in the appropriate derived class associated with the object.
- Through the use of `virtual` functions and polymorphism, a member-function call can cause different actions, depending on the type of the object receiving the call.
- Although we cannot instantiate objects of abstract base classes, we can declare pointers and references to objects of abstract base classes. Such pointers and references can be used to enable polymorphic manipulations of derived-class objects instantiated from concrete derived classes.
- Dynamic binding requires that at runtime, the call to a virtual member function be routed to the `virtual` function version appropriate for the class. A `virtual` function table called the vtable is implemented as an array containing function pointers. Each class with `virtual` functions has a vtable. For each `virtual` function in the class, the vtable has an entry containing a function pointer to the version of the `virtual` function to use for an object of that class. The `virtual` function to use for a particular class could be the function defined in that class, or it could be a function inherited either directly or indirectly from a base class higher in the hierarchy.
- When a base class provides a `virtual` member function, derived classes can override the `virtual` function, but they do not have to override it. Thus, a derived class can use a base class's version of a

`virtual` function.

- Each object of a class with `virtual` functions contains a pointer to the vtable for that class. When a function call is made from a base-class pointer to a derived-class object, the appropriate function pointer in the vtable is obtained and dereferenced to complete the call at execution time. This vtable lookup and pointer dereferencing require nominal runtime overhead.
- Any class that has one or more `0` pointers in its vtable is an abstract class. Classes without any `0` vtable pointers are concrete classes.
- New kinds of classes are regularly added to systems. New classes are accommodated by dynamic binding (also called late binding). The type of an object need not be known at compile time for a virtual-function call to be compiled. At runtime, the appropriate member function will be called for the object to which the pointer points.
- Operator `dynamic_cast` checks the type of the object to which the pointer points, then determines whether this type has an is-a relationship with the type to which the pointer is being converted. If there is an is-a relationship, `dynamic_cast` returns the object's address. If not, `dynamic_cast` returns `0`.
- Operator `typeid` returns a reference to an object of class `type_info` that contains information about the type of its operand, including the name of the type. To use `typeid`, the program must include header file `<typeinfo>`.
- When invoked, `type_info` member function `name` returns a pointer-based string that contains the name of the type that the `type_info` object represents.
- Operators `dynamic_cast` and `typeid` are part of C++'s run-time type information (RTTI) feature, which allows a program to determine an object's type at runtime.
- Declare the base-class destructor `virtual` if the class contains `virtual` functions. This makes all derived-class destructors virtual, even though they do not have the same name as the base-class destructor. If an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called. After a derived-class destructor runs, the destructors for all of that class's base classes run all the way up the hierarchythe root class's destructor runs last.

[Page 746]


[Page 746 (continued)]

Terminology

abstract base class

abstract class

base-class pointer to a base-class object

base-class pointer to a derived-class object

concrete class

dangerous pointer manipulation

derived-class pointer to a base-class object

derived-class pointer to a derived-class object

displacement

downcasting

dynamic binding

dynamic casting

`dynamic_cast`

dynamically determine function to execute

flow of control of a `virtual` function call

implementation inheritance

interface inheritance

iterator class

late binding

`name` function of class `type_info`

nonvirtual destructor

object's vtable pointer

offset into a vtable

override a function

polymorphic programming

polymorphism

polymorphism as an alternative to `switch` logic

programming in the general

programming in the specific

pure specifier

pure `virtual` function

RTTI (run-time type information)

static binding

`switch` logic

`type_info` class

`typeid` operator

`<typeinfo>` header file

`virtual` destructor

`virtual` function

`virtual` function table (vtable)

`virtual` keyword

vtable

vtable pointer

[Page 746 (continued)]

Self-Review Exercises

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

a. Treating a base-class object as a(n) _____ can cause errors.

b. Polymorphism helps eliminate _____ logic.

c. If a class contains at least one pure `virtual` function, it is a(n) _____ class.

d. Classes from which objects can be instantiated are called _____ classes.

e. Operator _____ can be used to downcast base-class pointers safely.

f. Operator `typeid` returns a reference to a(n) _____ object.

g. _____ involves using a base-class pointer or reference to invoke `virtual` functions on base-class and derived-class objects.

h. Overridable functions are declared using keyword _____ .

i. Casting a base-class pointer to a derived-class pointer is called _____ .

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

a. All `virtual` functions in an abstract base class must be declared as pure `virtual` functions.

b. Referring to a derived-class object with a base-class handle is dangerous.

c. A class is made abstract by declaring that class `virtual`.

d. If a base class declares a pure `virtual` function, a derived class must implement that function to become a concrete class.

e. Polymorphic programming can eliminate the need for `switch` logic.

[Page 747]

Answers to Self-Review Exercises

13.1 a) derived-class object. b) switch. c) abstract. d) concrete. e) `dynamic_cast`. f) `type_info`. g) Polymorphism. h) `virtual`. i) downcasting.

13.2 a) False. An abstract base class can include virtual functions with implementations. b) False. Referring to a base-class object with a derived-class handle is dangerous. c) False. Classes are never declared `virtual`. Rather, a class is made abstract by including at least one pure virtual function in the class. d) True. e) True.

[Page 747 (continued)]

Exercises

13.3 How is it that polymorphism enables you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

13.4 Discuss the problems of programming with `switch` logic. Explain why polymorphism can be an effective alternative to using `switch` logic.

13.5 Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

13.6 What are `virtual` functions? Describe a circumstance in which `virtual` functions would be appropriate.

13.7 Distinguish between static binding and dynamic binding. Explain the use of `virtual` functions and the vtable in dynamic binding.

13.8 Distinguish between `virtual` functions and pure `virtual` functions.

13.9 Suggest one or more levels of abstract base classes for the `Shape` hierarchy discussed in this chapter and shown in Fig. 12.3. (The first level is `Shape`, and the second level consists of the classes `TwoDimensionalShape` and `THReeDimensionalShape`.)

13.10 How does polymorphism promote extensibility?

13.11 You have been asked to develop a flight simulator that will have elaborate graphical outputs.

Explain why polymorphic programming would be especially effective for a problem of this nature.

13.12 (Payroll System Modification) Modify the payroll system of Figs. 13.1313.23 to include `private` data member `birthDate` in class `Employee`. Use class `Date` from Figs. 11.1211.13 to represent an employee's birthday. Assume that payroll is processed once per month. Create a `vector` of `Employee` references to store the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

13.13 (`Shape` Hierarchy) Implement the `Shape` hierarchy designed in Exercise 12.7 (which is based on the hierarchy in Fig. 12.3). Each `TwoDimensionalShape` should contain function `getArea` to calculate the area of the two-dimensional shape. Each `ThreeDimensionalShape` should have member functions `getArea` and `getVolume` to calculate the surface area and volume of the three-dimensional shape, respectively. Create a program that uses a `vector` of `Shape` pointers to objects of each concrete class in the hierarchy. The program should print the object to which each vector element points. Also, in the loop that processes all the shapes in the `vector`, determine whether each shape is a `TwoDimensionalShape` or a `ThreeDimensionalShape`. If a shape is a `TwoDimensionalShape`, display its area. If a shape is a `ThreeDimensionalShape`, display its area and volume.

13.14 (Polymorphic Screen Manager Using `Shape` Hierarchy) Develop a basic graphics package. Use the `Shape` hierarchy implemented in Exercise 13.13. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill characters to be used in drawing each shape. The user can specify more than one of the same shape. As you create each shape, place a `Shape *` pointer to each new `Shape` object into an array. Each `Shape` class should now have its own `draw` member function. Write a polymorphic screen manager that walks through the array, sending `draw` messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape.

[Page 748]

13.15 (`Package` Inheritance Hierarchy) Use the `Package` inheritance hierarchy created in Exercise 12.9 to create a program that displays the address information and calculates the shipping costs for several `Packages`. The program should contain a `vector` of `Package` pointers to objects of classes `TwoDayPackage` and `OvernightPackage`. Loop through the `vector` to process the `Packages` polymorphically. For each `Package`, invoke get functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each `Package`'s `calculateCost` member function and print the result. Keep track of the total shipping cost for all `Packages` in the `vector`, and display this total when the loop terminates.

13.16 (Polymorphic Banking Program Using `Account` Hierarchy) Develop a polymorphic banking program using the `Account` hierarchy created in Exercise 12.10. Create a `vector` of `Account` pointers to `SavingsAccount` and `CheckingAccount` objects. For each `Account` in the `vector`, allow the user to specify an amount of money to withdraw from the `Account` using member function `debit` and an amount of money to deposit into the `Account` using member function `credit`. As you process each `Account`, determine its type. If an `Account` is a `SavingsAccount`, calculate the amount of interest owed to the `Account` using member function `calculateInterest`, then add the interest to the account balance using member function `credit`. After processing an `Account`, print the updated account balance obtained by invoking base class member function `getBalance`.