Chapter 12 Object-Oriented Programming: Polymorphism

C++ How to Program, 9/e, GE

OBJECTIVES

In this chapter you'll learn:

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

12.1 Introduction

- 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*."
  - Enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.
- Polymorphism works off base-class pointer handles and base-class *reference handles*, but *not* off name handles.
- Relying on each object to know how to "do the right thing" in response to the same function call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

12.1 Introduction (cont.)

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

12.2 Introduction to Polymorphism: Polymorphic Video Game

Software Engineering Observation 12.1

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

Software Engineering Observation 12.2

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

12.3 Relationships Among Objects in an Inheritance Hierarchy

- 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.
- 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.
- 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.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects

- The example in Fig. 12.1 reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Section 11.3.5.
- The first two are natural and straightforward—we aim a base-class pointer at a base-class object and invoke base-class functionality, and we aim a derived-class pointer at a derived-class object and invoke derived-class functionality.
- 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.

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

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 5.)

```
23      // create derived-class pointer
24      BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = nullptr;
25
26      // set floating-point output formatting
27      cout << fixed << setprecision( 2 );
28
29      // output objects commissionEmployee and basePlusCommissionEmployee
30      cout << "Print base-class and derived-class objects:\n\n";
31      commissionEmployee.print(); // invokes base-class print
32      cout << "\n\n";
33      basePlusCommissionEmployee.print(); // invokes derived-class print
34
35      // aim base-class pointer at base-class object and print
36      commissionEmployeePtr = &commissionEmployee; // perfectly natural
37      cout << "\n\n\nCalling print with base-class pointer to "
38         << "\nbase-class object invokes base-class print function:\n\n";
39      commissionEmployeePtr->print(); // invokes base-class print
40
41      // aim derived-class pointer at derived-class object and print
42      basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
43      cout << "\n\n\nCalling print with derived-class pointer to "
44         << "\nderived-class object invokes derived-class "
45         << "print function:\n\n";
46      basePlusCommissionEmployeePtr->print(); // invokes derived-class print
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 5.)

```
47
48      // aim base-class pointer at derived-class object and print
49      commissionEmployeePtr = &basePlusCommissionEmployee;
50      cout << "\n\n\nCalling print with base-class pointer to "
51         << "derived-class object\ninvokes base-class print "
52         << "function on that derived-class object:\n\n";
53      commissionEmployeePtr->print(); // invokes base-class print
54      cout << endl;
55   } // end main
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 5.)

```
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:
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 4 of 5.)

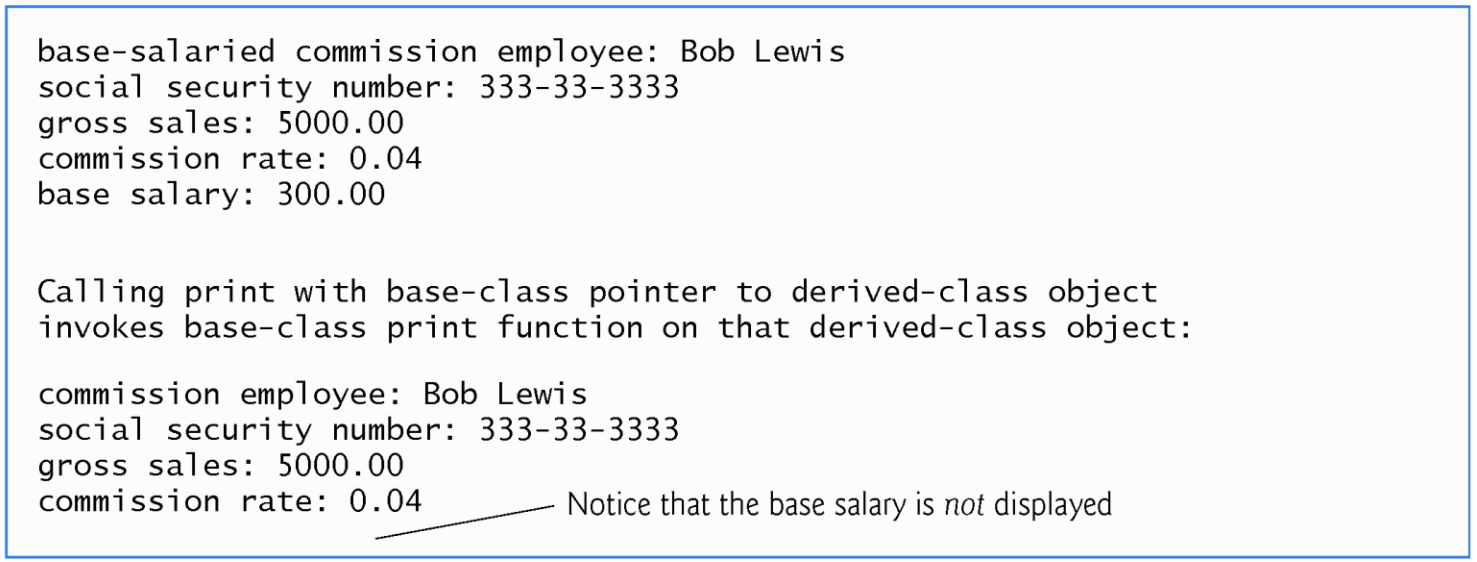

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 5 of 5.)

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

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

- Line 36 assigns the address of base-class object `commissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 39 uses to invoke member function `print` on that `CommissionEmployee` object.
  - This invokes the version of `print` defined in base class `CommissionEmployee`.

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

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

- Line 42 assigns the address of derived-class object `basePlusCommissionEmployee` to derived-class pointer `basePlusCommissionEmployee-Ptr`, which line 46 uses to invoke member function `print` on that `BasePlusCommissionEmployee` object.
  - This invokes the version of `print` defined in derived class `BasePlusCommissionEmployee`.

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

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

- Line 49 assigns the address of derived-class object `BasePlusCommissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 53 uses to invoke member function `print`.
  - This "crossover" is allowed 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 pointer (or reference) used to invoke the function, not the type of the object for which the member function is called.*

12.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 12.2, we aim a derived-class pointer at a base-class object.
- 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`.

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

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch12\Fig12_02\fig12_02.cpp(14): error C2440: '=' :
   cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee *'
          Cast from base to derived requires dynamic_cast or static_cast
```

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

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

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

```
1  // Fig. 12.3: fig12_03.cpp
2  // Attempting to invoke derived-class-only member functions
3  // via a base-class pointer.
4  #include <string>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    CommissionEmployee *commissionEmployeePtr = nullptr; // base class ptr
12    BasePlusCommissionEmployee basePlusCommissionEmployee(
13       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
14
15    // aim base-class pointer at derived-class object (allowed)
16    commissionEmployeePtr = &basePlusCommissionEmployee;
17
```

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

```
18      // invoke base-class member functions on derived-class
19      // object through base-class pointer (allowed)
20      string firstName = commissionEmployeePtr->getFirstName();
21      string lastName = commissionEmployeePtr->getLastName();
22      string ssn = commissionEmployeePtr->getSocialSecurityNumber();
23      double grossSales = commissionEmployeePtr->getGrossSales();
24      double commissionRate = commissionEmployeePtr->getCommissionRate();
25
26      // attempt to invoke derived-class-only member functions
27      // on derived-class object through base-class pointer (disallowed)
28      double baseSalary = commissionEmployeePtr->getBaseSalary();
29      commissionEmployeePtr->setBaseSalary( 500 );
30  } // end main
```

GNU C++ compiler error messages:

```
fig12_03.cpp:28:47: error: 'class CommissionEmployee' has no member named
   'getBaseSalary'
fig12_03.cpp:29:27: error: 'class CommissionEmployee' has no member named
   'setBaseSalary'
```

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

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

Downcasting

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

Software Engineering Observation 12.3

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

12.3.4 Virtual Functions and Virtual Destructors

Why virtual Functions Are Useful

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

Software Engineering Observation 12.4

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

12.3.4 Virtual Functions and Virtual Destructors (cont.)

Declaring virtual Functions

- To enable this behavior, we declare `draw` in the base class as a virtual function, and we override `draw` in each of the derived classes to draw the appropriate shape.
- From an implementation perspective, *overriding* a function is no different than *redefining* one.
  - An overridden function in a derived class has the *same signature and return type* (i.e., *prototype*) as the function it overrides in its base class.
- If we declare the base-class function as `virtual`, we can *override* that function to enable *polymorphic behavior*.
- We declare a `virtual` function by preceding the function's prototype with the key-word `virtual` in the base class.

Software Engineering Observation 12.5

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

Good Programming Practice 12.1

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 class hierarchy to promote program clarity.

Software Engineering Observation 12.6

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.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

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

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

- When a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator (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 object—this is *not* polymorphic behavior.
- Dynamic binding with `virtual` functions occurs only off pointers (and, as we'll soon see, references).

12.3.4 Virtual Functions and Virtual Destructors (cont.)

***virtual* Functions in the *CommissionEmployee* Hierarchy**

- Figures 12.4–12.5 are the headers for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively.
- We modified these to declare each class's `earnings` and `print` member functions as `virtual` (lines 29–30 of Fig. 12.4 and lines 19–20 of Fig. 12.5).
- Because functions `earnings` and `print` are `virtual` in class `CommissionEmployee`, class `BasePlusCommissionEmployee`'s `earnings` and `print` functions *override* class `CommissionEmployee`'s.
- In addition, class `BasePlusCommissionEmployee`'s `earnings` and `print` functions are declare `override`.

Error-Prevention Tip 12.1

Apply C++11's `override` keyword to every overridden function in a derived-class. This forces the compiler to check whether the base class has a member function with the same name and parameter list (i.e., the same signature). If not, the compiler generates an error.

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class header declares earnings and print as virtual.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7
8  class CommissionEmployee
9  {
10 public:
11    CommissionEmployee( const std::string &, const std::string &,
12       const std::string &, double = 0.0, double = 0.0 );
13
14    void setFirstName( const std::string & ); // set first name
15    std::string getFirstName() const; // return first name
16
17    void setLastName( const std::string & ); // set last name
18    std::string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const std::string & ); // set SSN
21    std::string getSocialSecurityNumber() const; // return SSN
22
```

Fig. 12.4 | CommissionEmployee class header declares earnings and print as virtual.

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

Fig. 12.4 | CommissionEmployee class header declares `earnings` and `print` as `virtual`.

```
1  // Fig. 12.5: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9
10 class BasePlusCommissionEmployee : public CommissionEmployee
11 {
12 public:
13    BasePlusCommissionEmployee( const std::string &, const std::string &,
14       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
15
16    void setBaseSalary( double ); // set base salary
17    double getBaseSalary() const; // return base salary
18
```

Fig. 12.5 | `BasePlusCommissionEmployee` class header declares `earnings` and `print` functions as `virtual` and `override`. (Part 1 of 2.)

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

Fig. 12.5 | `BasePlusCommissionEmployee` class header declares `earnings` and `print` functions as `virtual` and `override`. (Part 2 of 2.)

12.3.4 Virtual Functions and Virtual Destructors (cont.)

- We modified Fig. 12.1 to create the program of Fig. 12.6.
- Lines 40–51 of Fig. 12.6 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 54 aims base-class pointer `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`.
- Note that when line 61 invokes member function `print` off the base-class pointer, the derived-class `BasePlusCommissionEmployee`'s `print` member function is invoked, so line 61 outputs different text than line 53 does in Fig. 12.1 (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*.

```
1  // Fig. 12.6: fig12_06.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    // create base-class object
12    CommissionEmployee commissionEmployee(
13       "Sue", "Jones", "222-22-2222", 10000, .06 );
14
15    // create base-class pointer
16    CommissionEmployee *commissionEmployeePtr = nullptr;
17
18    // create derived-class object
19    BasePlusCommissionEmployee basePlusCommissionEmployee(
20       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
21
```

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

```
22      // create derived-class pointer
23      BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = nullptr;
24
25      // set floating-point output formatting
26      cout << fixed << setprecision( 2 );
27
28      // output objects using static binding
29      cout << "Invoking print function on base-class and derived-class "
30         << "\nobjects with static binding\n\n";
31      commissionEmployee.print(); // static binding
32      cout << "\n\n";
33      basePlusCommissionEmployee.print(); // static binding
34
35      // output objects using dynamic binding
36      cout << "\n\n\nInvoking print function on base-class and "
37         << "derived-class \nobjects with dynamic binding";
38
39      // aim base-class pointer at base-class object and print
40      commissionEmployeePtr = &commissionEmployee;
41      cout << "\n\nCalling virtual function print with base-class pointer"
42         << "\nto base-class object invokes base-class "
43         << "print function:\n\n";
44      commissionEmployeePtr->print(); // invokes base-class print
```

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

```
45
46      // aim derived-class pointer at derived-class object and print
47      basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
48      cout << "\n\nCalling virtual function print with derived-class "
49         << "pointer\nto derived-class object invokes derived-class "
50         << "print function:\n\n";
51      basePlusCommissionEmployeePtr->print(); // invokes derived-class print
52
53      // aim base-class pointer at derived-class object and print
54      commissionEmployeePtr = &basePlusCommissionEmployee;
55      cout << "\n\nCalling virtual function print with base-class pointer"
56         << "\nto derived-class object invokes derived-class "
57         << "print function:\n\n";
58
59      // polymorphism; invokes BasePlusCommissionEmployee's print;
60      // base-class pointer to derived-class object
61      commissionEmployeePtr->print();
62      cout << endl;
63   } // end main
```

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

```
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
```

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

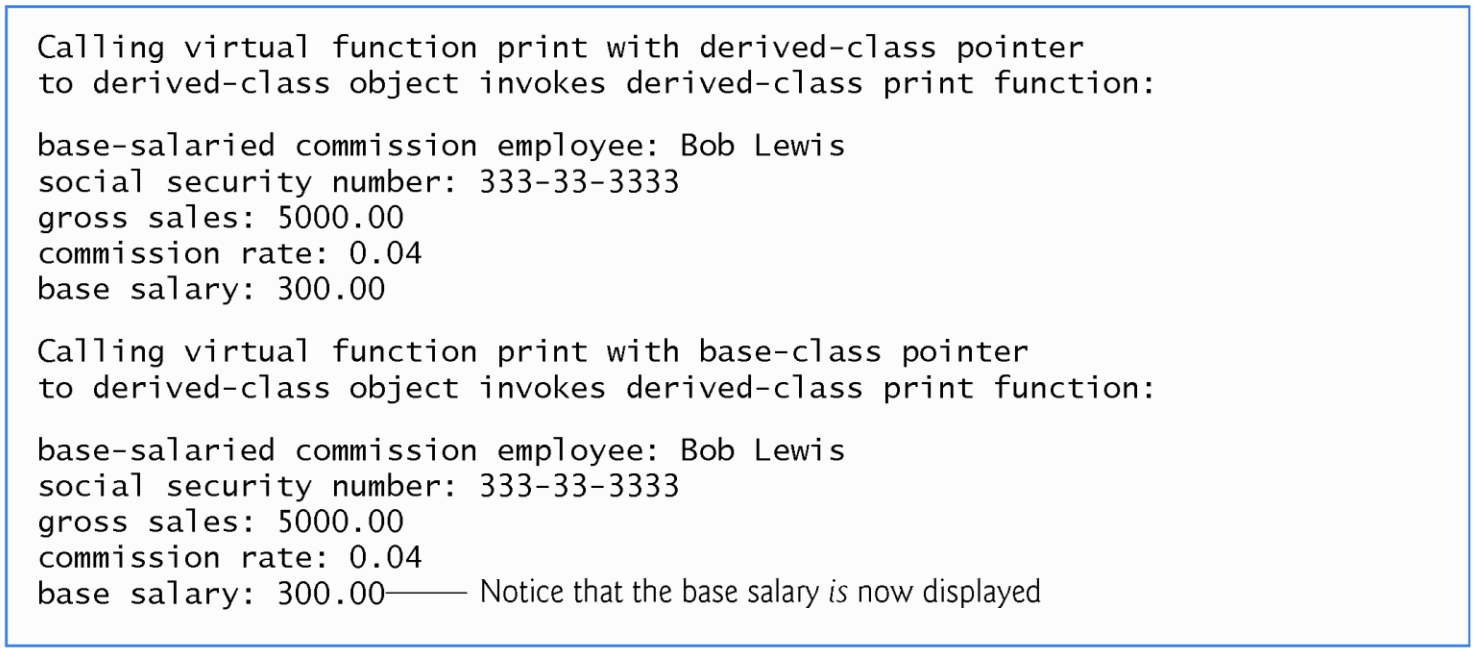

Fig. 12.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 5 of 5.)

12.3.4 Virtual Functions and Virtual Destructors (cont.)

virtual Destructors

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

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

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

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

Error-Prevention Tip 12.2

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

Common Programming Error 12.1

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

12.3.4 Virtual Functions and Virtual Destructors (cont.)

C++11: `final` Member Functions and Classes

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

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

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

12.3.4 Virtual Functions and Virtual Destructors (cont.)

- As of C++11, you can declare a class as final to prevent it from being used as a base class, as in

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

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

12.4 Type Fields and `switch` Statements

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

Software Engineering Observation 12.7

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

Software Engineering Observation 12.8

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

12.5 Abstract Classes and Pure `virtual` Functions

- There are cases in which it's useful to define *classes from which you never intend to instantiate any objects*.
- Such classes are called abstract classes.
- Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as abstract base classes.
- These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*—derived classes must define the "missing pieces."
- An abstract class is a base class from which other classes can inherit.
- Classes that can be used to instantiate objects are called concrete classes.
- Such classes define *every* member function they declare.

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

- Abstract base classes are *too generic* to define real objects; we need to be *more specific* before we can think of instantiating objects.
- For example, if someone tells you to "draw the two-dimensional shape," what shape would you draw?
- Concrete classes provide the *specifics* that make it possible to instantiate objects.
- An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes.
- In some cases, abstract classes constitute the top few levels of the hierarchy.
- A good example of this is the shape hierarchy in Fig. 12.3, which begins with abstract base class `Shape`.

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

Pure Virtual Functions

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

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

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

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

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

Software Engineering Observation 12.9

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

Common Programming Error 12.2

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

Software Engineering Observation 12.10

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

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

- 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.

12.6 Case Study: Payroll System Using Polymorphism

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

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

- We create an enhanced employee hierarchy to solve the following problem:
  - A company pays its employees weekly. The employees are of three types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, commission employees are paid a percentage of their sales 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.

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

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

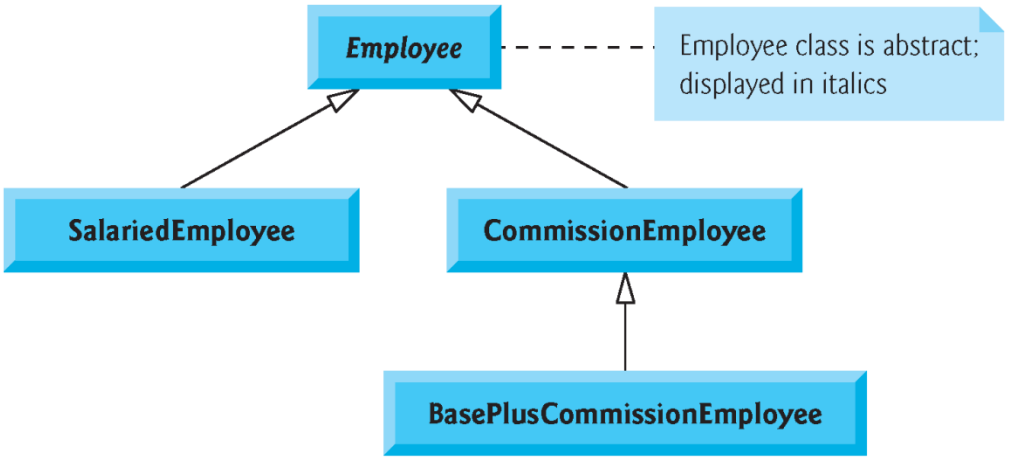

Fig. 12.7 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 12.11

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

12.6.1 Creating Abstract Base Class `Employee`

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

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

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

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

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

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

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

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

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

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

`Employee` Class Header

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

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

`Employee` Class Member-Function Definitions

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

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

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

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

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

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

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

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

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

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

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

12.6.2 Creating Concrete Derived Class `SalariedEmployee`

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

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

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

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

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

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

`SalariedEmployee` Class Member-Function Definitions

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

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

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

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

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

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

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

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

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

12.6.3 Creating Concrete Derived Class `CommissionEmployee`

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

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

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

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

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

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

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

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

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

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

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

12.6.4 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

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

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

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

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

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

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

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

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

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

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

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

12.6.5 Demonstrating Polymorphic Processing

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

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

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

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

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

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

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

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

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

```
Employees processed individually using static binding:

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

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

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

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

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

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

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

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

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

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

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

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

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

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

12.6.5 Demonstrating Polymorphic Processing (cont.)

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

12.6.5 Demonstrating Polymorphic Processing (cont.)

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

12.6.5 Demonstrating Polymorphic Processing (cont.)

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

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

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

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

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

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

***Employee Class* vtable**

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

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

SalariedEmployee Class vtable

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

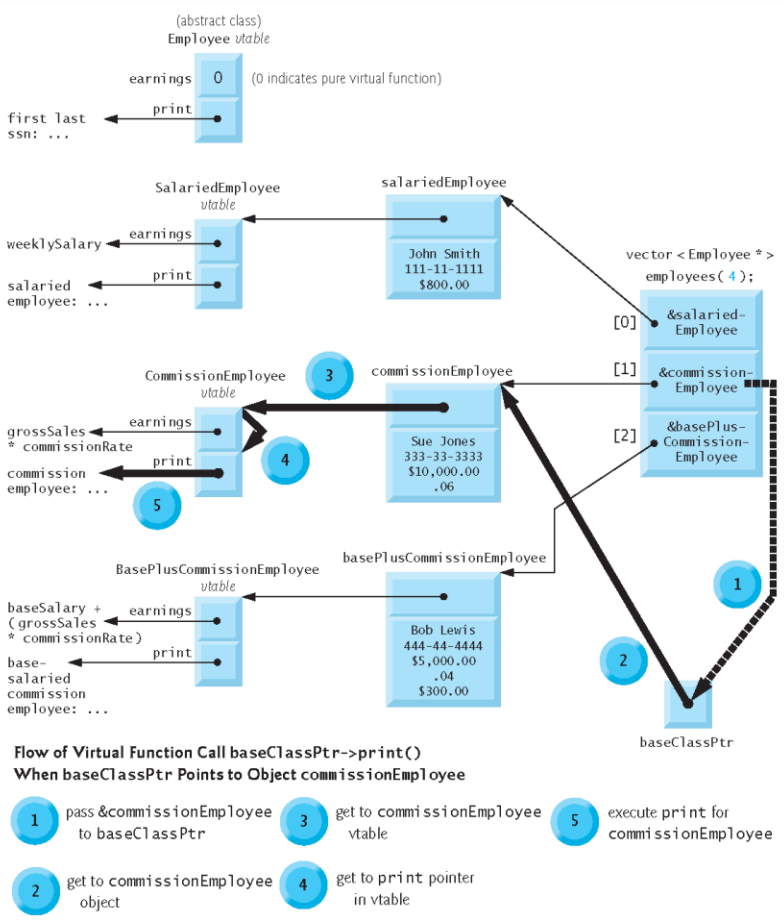

Fig. 12.18 | How `virtual` function calls work.

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

CommissionEmployee Class vtable

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

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

***BasePlusCommissionEmployee* Class vtable**

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

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

Three Levels of Pointers to Implement Polymorphism

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

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

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

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

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

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

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

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

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

Performance Tip 12.1

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

Performance Tip 12.2

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

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

- Recall from the problem statement at the beginning of Section 12.6 that, for the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployee`s by adding 10 percent to their base salaries.
- When processing `Employee` objects polymorphically in Section 12.6.5, we did not need to worry about the "specifics."
- To adjust the base salaries of `BasePlusCommissionEmployee`s, we have to determine the specific type of each `Employee` object at execution time, then act appropriately.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Portability Tip 12.1

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