Chapter 13 Object-Oriented Programming: Polymorphism

C++ How to Program, Late Objects Version, 7/e

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.

13.1 Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hier-archies.
- 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.

13.1 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 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 you add to the hierarchy.

13.2 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.
- If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object.
  - Any operation 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 `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
- Polymorphism occurs when a program invokes a `virtual` function through a base-class pointer or reference.
  - C++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated.

Software Engineering Observation 13.1

With `virtual` *functions and polymorphism, you can 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 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.

13.3 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.

13.3.1 Invoking Base-Class Functions from Derived-Class Objects

- The example in Figs. 13.1–13.5 demonstrates three ways to aim base and derived-class pointers at base and derived-class objects.
- The first two are 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).

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

- Class `CommissionEmployee` (Figs. 13.1–13.2), which we discussed in Chapter 12, is used to represent employees who are paid a percentage of their sales.
- Class `BasePlusCommissionEmployee` (Figs. 13.3–13.4), 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 30–33 of Fig. 13.4) redefines class `CommissionEmployee`'s `earnings` member function (lines 78–81 of Fig. 13.2) to include the object's base salary.
- Class `BasePlusCommissionEmployee`'s `print` member function (lines 36–44 of Fig. 13.4) redefines class `CommissionEmployee`'s `version` (lines 84–91 of Fig. 13.2) to display the same information plus the employee's base salary.

```
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 namespace std;
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
```

Fig. 13.1 | `CommissionEmployee` class header file. (Part 1 of 2.)

```
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
```

Fig. 13.1 | `CommissionEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  #include "CommissionEmployee.h" // CommissionEmployee class definition
5  using namespace std;
6
7  // constructor
8  CommissionEmployee::CommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate )
11    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
12 {
13    setGrossSales( sales ); // validate and store gross sales
14    setCommissionRate( rate ); // validate and store commission rate
15 } // end CommissionEmployee constructor
16
17 // set first name
18 void CommissionEmployee::setFirstName( const string &first )
19 {
20    firstName = first; // should validate
21 } // end function setFirstName
22
```

Fig. 13.2 | `CommissionEmployee` class implementation file. (Part 1 of 4.)

```
23  // return first name
24  string CommissionEmployee::getFirstName() const
25  {
26     return firstName;
27  } // end function getFirstName
28
29  // set last name
30  void CommissionEmployee::setLastName( const string &last )
31  {
32     lastName = last;  // should validate
33  } // end function setLastName
34
35  // return last name
36  string CommissionEmployee::getLastName() const
37  {
38     return lastName;
39  } // end function getLastName
40
41  // set social security number
42  void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
43  {
44     socialSecurityNumber = ssn; // should validate
45  } // end function setSocialSecurityNumber
46
```

Fig. 13.2 | `CommissionEmployee` class implementation file. (Part 2 of 4.)

```
47  // return social security number
48  string CommissionEmployee::getSocialSecurityNumber() const
49  {
50     return socialSecurityNumber;
51  } // end function getSocialSecurityNumber
52
53  // set gross sales amount
54  void CommissionEmployee::setGrossSales( double sales )
55  {
56     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
57  } // end function setGrossSales
58
59  // return gross sales amount
60  double CommissionEmployee::getGrossSales() const
61  {
62     return grossSales;
63  } // end function getGrossSales
64
65  // set commission rate
66  void CommissionEmployee::setCommissionRate( double rate )
67  {
68     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
69  } // end function setCommissionRate
70
```

Fig. 13.2 | `CommissionEmployee` class implementation file. (Part 3 of 4.)

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

Fig. 13.2 | `CommissionEmployee` class implementation file. (Part 4 of 4.)

```
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  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9  using namespace std;
10
11 class BasePlusCommissionEmployee : public CommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17    void setBaseSalary( double ); // set base salary
18    double getBaseSalary() const; // return base salary
19
20    double earnings() const; // calculate earnings
21    void print() const; // print BasePlusCommissionEmployee object
```

Fig. 13.3 | `BasePlusCommissionEmployee` class header file. (Part 1 of 2.)

```
22   private:
23      double baseSalary; // base salary
24   }; // end class BasePlusCommissionEmployee
25
26   #endif
```

Fig. 13.3 | `BasePlusCommissionEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.4: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  #include "BasePlusCommissionEmployee.h"
5  using namespace std;
6
7  // constructor
8  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate, double salary )
11    // explicitly call base-class constructor
12    : CommissionEmployee( first, last, ssn, sales, rate )
13 {
14    setBaseSalary( salary ); // validate and store base salary
15 } // end BasePlusCommissionEmployee constructor
16
17 // set base salary
18 void BasePlusCommissionEmployee::setBaseSalary( double salary )
19 {
20    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21 } // end function setBaseSalary
22
```

Fig. 13.4 | `BasePlusCommissionEmployee` class implementation file. (Part 1 of 2.)

```
23  // return base salary
24  double BasePlusCommissionEmployee::getBaseSalary() const
25  {
26     return baseSalary;
27  } // end function getBaseSalary
28
29  // calculate earnings
30  double BasePlusCommissionEmployee::earnings() const
31  {
32     return getBaseSalary() + CommissionEmployee::earnings();
33  } // end function earnings
34
35  // print BasePlusCommissionEmployee object
36  void BasePlusCommissionEmployee::print() const
37  {
38     cout << "base-salaried ";
39
40     // invoke CommissionEmployee's print function
41     CommissionEmployee::print();
42
43     cout << "\nbase salary: " << getBaseSalary();
44  } // end function print
```

Fig. 13.4 | `BasePlusCommissionEmployee` class implementation file. (Part 2 of 2.)

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

- 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`.
- 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`.

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

- 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 handle (i.e., the pointer or reference type) used to invoke the function, not the type of the object to which the handle points.I

```
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  #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 = 0;
18
19    // create derived-class object
20    BasePlusCommissionEmployee basePlusCommissionEmployee(
21       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
22
```

Fig. 13.5 | 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 = 0;
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";
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 5.)

```
46      basePlusCommissionEmployeePtr->print(); // invokes derived-class print
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
```

```
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
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 5.)

```
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
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 4 of 5.)

```
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
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 5 of 5.)

13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 13.6, 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, because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`.

```
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 } // end main
```

Fig. 13.6 | Aiming a derived-class pointer at a base-class object. (Part 1 of 2.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440: '=' :
cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee *'
        Cast from base to derived requires dynamic_cast or static_cast
```

GNU C++ compiler error messages:

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

Fig. 13.6 | Aiming a derived-class pointer at a base-class object. (Part 2 of 2.)

13.3.3 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 13.7 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer.

```
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 (allowed)
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
```

Fig. 13.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 3.)

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

Fig. 13.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 3.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp7_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp7_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp7_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)
```

Fig. 13.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 3 of 3.)

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

- The compiler allows 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 13.8 shows a concrete example of downcasting.

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's 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

- 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.
- 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`.
  - 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`.
  - 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.

13.3.4 Virtual Functions (cont.)

- 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 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 derived class overrides it.*

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 you browse a class hierarchy to locate a class to reuse, it's possible that a function in that class will exhibit `virtual` *function behavior even though it isn't 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.*

13.3.4 Virtual Functions (cont.)

- 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.
- 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 re-solved at compile time (this is called static binding) and the `virtual` function that is called is the one defined for (or inherited by) the class of that particular object—this is not polymorphic behavior.
- Dynamic binding with `virtual` functions occurs only off pointer (and, as we'll soon see, reference) handles.

13.3.4 Virtual Functions (cont.)

- Figures 13.8–13.9 are the header files for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively.
- 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 30–31 of Fig. 13.8 and lines 20–21 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.

```
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 namespace std;
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
```

Fig. 13.8 | `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 1 of 2.)

```
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
```

Fig. 13.8 | `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 2 of 2.)

```
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  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9  using namespace std;
10
11 class BasePlusCommissionEmployee : public CommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17    void setBaseSalary( double ); // set base salary
18    double getBaseSalary() const; // return base salary
19
20    virtual double earnings() const; // calculate earnings
21    virtual void print() const; // print BasePlusCommissionEmployee object
```

Fig. 13.9 | `BasePlusCommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 1 of 2.)

```
22   private:
23      double baseSalary; // base salary
24   }; // end class BasePlusCommissionEmployee
25
26   #endif
```

Fig. 13.9 | `BasePlusCommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 2 of 2.)

13.3.4 Virtual Functions (cont.)

- We modified Fig. 13.5 to create the program of Fig. 13.10.
- Lines 40–51 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. 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.

```
1  // Fig. 13.10: fig13_10.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 = 0;
17
18    // create derived-class object
19    BasePlusCommissionEmployee basePlusCommissionEmployee(
20       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
21
```

Fig. 13.10 | 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 = 0;
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";
```

Fig. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 5.)

```
44      commissionEmployeePtr->print(); // invokes base-class print
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. 13.10 | 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. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 4 of 5.)

```
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
```

Fig. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 5 of 5.)

13.3.5 Summary of the Allowed Assignments Between 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.
- Base-class objects cannot be treated as if they were derived-class objects—the derived class can have additional derived-class-only members.
- Aiming a derived-class pointer at a base-class object is not allowed without an explicit cast—such 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."

13.3.6 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers (cont.)

- We've discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:
  - Aiming a base-class pointer at a base-class object is straightforward—calls made off the base-class pointer simply invoke base-class functionality.
  - Aiming a derived-class pointer at a derived-class object is straightfor-ward—calls made off the derived-class pointer simply invoke derived-class functionality.
  - Aiming a base-class pointer at a derived-class object is safe, because the derived-class object *is an* object of its base class. This pointer can be used to invoke only base-class member functions.
  - 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.

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.

13.4 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 13.6

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

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 simpler sequential code. This simplification facilitates testing, debugging and program maintenance.

13.5 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 provides 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.

13.5 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 reasonable 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.

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

- A good example of this is the shape hierarchy in Fig. 12.3, which begins with abstract base class `Shape`.
- 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
  - ```
    // pure virtual function
    virtual void draw() const = 0;
    ```
- The "`= 0`" is a pure specifier.
- Pure `virtual` functions do not provide implementations.
    ```

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

- 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 you want all concrete derived classes to implement the function.

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.*

13.5 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.

13.6 Case Study: Payroll System Using Polymorphism

- This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section 12.4. We use an abstract class and polymorphism to perform payroll calculations based on the type of employee.

13.6 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 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.

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

- The UML class diagram in Fig. 13.11 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`.

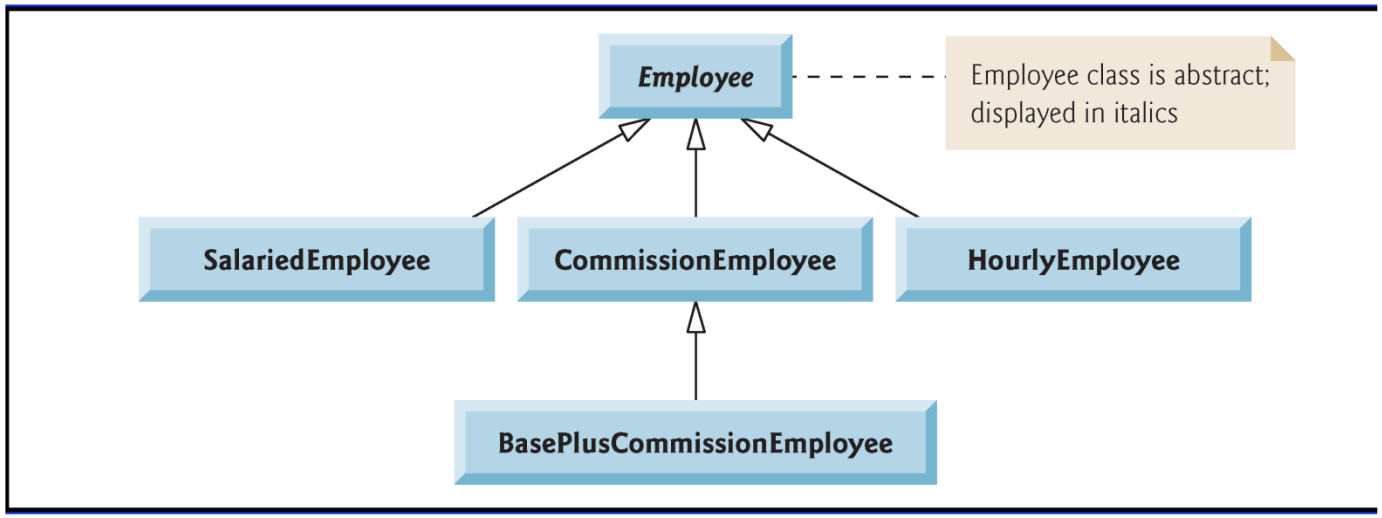

Fig. 13.11 | `Employee` hierarchy UML class diagram.

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 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).*

13.6.1 Creating Abstract Base Class `Employee`

- Class `Employee` (Figs. 13.13–13.14, 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 function—there is not enough information to determine what amount `earnings` should return.
- Each derived class overrides `earnings` with an appropriate implementation.

13.6.1 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 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.

13.6.1 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. 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.
- 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*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 13.12 | Polymorphic interface for the `Employee` hierarchy classes.

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

- Let's 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 13.14 contains the member-function implementations for class `Employee`.
- No implementation is provided for `virtual` function `earnings`.

```
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 namespace std;
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
```

Fig. 13.13 | `Employee` class header file. (Part 1 of 2.)

```
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
```

Fig. 13.13 | `Employee` class header file. (Part 2 of 2.)

```
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  #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. 13.14 | `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. 13.14 | `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. 13.14 | `Employee` class implementation file. (Part 3 of 3.)

13.6.2 Creating Concrete Derived Class `SalariedEmployee`

- Class `SalariedEmployee` (Figs. 13.15–13.16) derives from class `Employee` (line 8 of Fig. 13.15).

```
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
```

Fig. 13.15 | SalariedEmployee class header file.

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

- 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 10) to initialize the `private` data members that are inherited from the base class, but not accessible in the derived class.
- Function `earnings` (line 29–32) overrides pure `virtual` function `earnings` in `Employee` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary.
- If we did not implement `earnings`, class `SalariedEmployee` would be an abstract class.
- In class `SalariedEmployee`'s header file, 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. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  #include "SalariedEmployee.h" // SalariedEmployee class definition
5  using namespace std;
6
7  // constructor
8  SalariedEmployee::SalariedEmployee( const string &first,
9     const string &last, const string &ssn, double salary )
10    : Employee( first, last, ssn )
11 {
12    setWeeklySalary( salary );
13 } // end SalariedEmployee constructor
14
15 // set salary
16 void SalariedEmployee::setWeeklySalary( double salary )
17 {
18    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
19 } // end function setWeeklySalary
20
```

Fig. 13.16 | `SalariedEmployee` class implementation file. (Part 1 of 2.)

```
21  // return salary
22  double SalariedEmployee::getWeeklySalary() const
23  {
24     return weeklySalary;
25  } // end function getWeeklySalary
26
27  // calculate earnings;
28  // override pure virtual function earnings in Employee
29  double SalariedEmployee::earnings() const
30  {
31     return getWeeklySalary();
32  } // end function earnings
33
34  // print SalariedEmployee's information
35  void SalariedEmployee::print() const
36  {
37     cout << "salaried employee: ";
38     Employee::print(); // reuse abstract base-class print function
39     cout << "\nweekly salary: " << getWeeklySalary();
40  } // end function print
```

Fig. 13.16 | `SalariedEmployee` class implementation file. (Part 2 of 2.)

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

- Function `print` of class `SalariedEmployee` (lines 35–40 of Fig. 13.16) overrides `Employee` function `print`.
- If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`.

13.6.3 Creating Concrete Derived Class `HourlyEmployee`

- Class `HourlyEmployee` (Figs. 13.17–13.18) also derives from class `Employee` (line 8 of Fig. 13.17).

```
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    static const int hoursPerWeek = 168; // hours in one week
12
13    HourlyEmployee( const string &, const string &,
14       const string &, double = 0.0, double = 0.0 );
15
16    void setWage( double ); // set hourly wage
17    double getWage() const; // return hourly wage
18
19    void setHours( double ); // set hours worked
20    double getHours() const; // return hours worked
21
22    // keyword virtual signals intent to override
23    virtual double earnings() const; // calculate earnings
24    virtual void print() const; // print HourlyEmployee object
```

Fig. 13.17 | `HourlyEmployee` class header file. (Part 1 of 2.)

```
25   private:
26      double wage; // wage per hour
27      double hours; // hours worked for week
28   }; // end class HourlyEmployee
29
30   #endif // HOURLY_H
```

Fig. 13.17 | `HourlyEmployee` class header file. (Part 2 of 2.)

13.6.3 Creating Concrete Derived Class `HourlyEmployee` (cont.)

- Figure 13.18 contains the member-function implementations for class `HourlyEmployee`.
- The `HourlyEmployee` constructor, like the `SalariedEmployee` constructor, passes the first name, last name and social security number to the base class `Employee` constructor (line 10) to initialize the inherited `private` data members declared in the base class.
- In addition, `HourlyEmployee`'s `print` function calls base-class function `print` (line 55) to output the `Employee`-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  #include "HourlyEmployee.h" // HourlyEmployee class definition
5  using namespace std;
6
7  // constructor
8  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
9     const string &ssn, double hourlyWage, double hoursWorked )
10    : Employee( first, last, ssn )
11 {
12    setWage( hourlyWage ); // validate hourly wage
13    setHours( hoursWorked ); // validate hours worked
14 } // end HourlyEmployee constructor
15
16 // set wage
17 void HourlyEmployee::setWage( double hourlyWage )
18 {
19    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
20 } // end function setWage
21
```

Fig. 13.18 | `HourlyEmployee` class implementation file. (Part 1 of 3.)

```
22  // return wage
23  double HourlyEmployee::getWage() const
24  {
25     return wage;
26  } // end function getWage
27
28  // set hours worked
29  void HourlyEmployee::setHours( double hoursWorked )
30  {
31     hours = ( ( ( hoursWorked >= 0.0 ) &&
32        ( hoursWorked <= hoursPerWeek ) ) ? hoursWorked : 0.0 );
33  } // end function setHours
34
35  // return hours worked
36  double HourlyEmployee::getHours() const
37  {
38     return hours;
39  } // end function getHours
40
```

Fig. 13.18 | `HourlyEmployee` class implementation file. (Part 2 of 3.)

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

Fig. 13.18 | `HourlyEmployee` class implementation file. (Part 3 of 3.)

13.6.4 Creating Concrete Derived Class `CommissionEmployee`

- Class `CommissionEmployee` (Figs. 13.19–13.20) derives from `Employee` (Fig. 13.19, line 8).
- The constructor passes the first name, last name and social security number to the `Employee` constructor (line 10) to initialize `Employee`'s `private` data members.
- Function `print` calls base-class function `print` (line 50) to display the `Employee`-specific information.

```
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
```

Fig. 13.19 | `CommissionEmployee` class header file. (Part 1 of 2.)

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

Fig. 13.19 | `CommissionEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.20: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  #include "CommissionEmployee.h" // CommissionEmployee class definition
5  using namespace std;
6
7  // constructor
8  CommissionEmployee::CommissionEmployee( const string &first,
9     const string &last, const string &ssn, double sales, double rate )
10    : Employee( first, last, ssn )
11 {
12    setGrossSales( sales );
13    setCommissionRate( rate );
14 } // end CommissionEmployee constructor
15
16 // set commission rate
17 void CommissionEmployee::setCommissionRate( double rate )
18 {
19    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
20 } // end function setCommissionRate
21
```

Fig. 13.20 | `CommissionEmployee` class implementation file. (Part 1 of 3.)

```
22  // return commission rate
23  double CommissionEmployee::getCommissionRate() const
24  {
25      return commissionRate;
26  } // end function getCommissionRate
27
28  // set gross sales amount
29  void CommissionEmployee::setGrossSales( double sales )
30  {
31     grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
32  } // end function setGrossSales
33
34  // return gross sales amount
35  double CommissionEmployee::getGrossSales() const
36  {
37      return grossSales;
38  } // end function getGrossSales
39
40  // calculate earnings; override pure virtual function earnings in Employee
41  double CommissionEmployee::earnings() const
42  {
43     return getCommissionRate() * getGrossSales();
44  } // end function earnings
45
```

Fig. 13.20 | `CommissionEmployee` class implementation file. (Part 2 of 3.)

```
46  // print CommissionEmployee's information
47  void CommissionEmployee::print() const
48  {
49     cout << "commission employee: ";
50     Employee::print(); // code reuse
51     cout << "\ngross sales: " << getGrossSales()
52        << "; commission rate: " << getCommissionRate();
53  } // end function print
```

Fig. 13.20 | `CommissionEmployee` class implementation file. (Part 3 of 3.)

13.6.5 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

- Class `BasePlusCommissionEmployee` (Figs. 13.21–13.22) directly inherits from class `CommissionEmployee` (line 8 of Fig. 13.21) and therefore is an indirect derived class of class `Employee`.
- `BasePlusCommissionEmployee`'s `print` function (lines 36–41) 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. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from CommissionEmployee.
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
```

Fig. 13.21 | BasePlusCommissionEmployee class header file.

```
1  // Fig. 13.22: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  #include "BasePlusCommissionEmployee.h"
5  using namespace std;
6
7  // constructor
8  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate, double salary )
11    : CommissionEmployee( first, last, ssn, sales, rate )
12 {
13    setBaseSalary( salary ); // validate and store base salary
14 } // end BasePlusCommissionEmployee constructor
15
16 // set base salary
17 void BasePlusCommissionEmployee::setBaseSalary( double salary )
18 {
19    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
20 } // end function setBaseSalary
21
```

Fig. 13.22 | `BasePlusCommissionEmployee` class implementation file. (Part 1 of 2.)

```
22  // return base salary
23  double BasePlusCommissionEmployee::getBaseSalary() const
24  {
25      return baseSalary;
26  } // end function getBaseSalary
27
28  // calculate earnings;
29  // override virtual function earnings in CommissionEmployee
30  double BasePlusCommissionEmployee::earnings() const
31  {
32      return getBaseSalary() + CommissionEmployee::earnings();
33  } // end function earnings
34
35  // print BasePlusCommissionEmployee's information
36  void BasePlusCommissionEmployee::print() const
37  {
38     cout << "base-salaried ";
39     CommissionEmployee::print(); // code reuse
40     cout << "; base salary: " << getBaseSalary();
41  } // end function print
```

Fig. 13.22 | `BasePlusCommissionEmployee` class implementation file. (Part 2 of 2.)

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 23–30 create objects of each of the four concrete `Employee` derived classes.
- Lines 35–43 output each `Employee`'s information and earnings.
- Each member-function invocation in lines 35–43 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. 13.23: fig13_23.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 "HourlyEmployee.h"
10 #include "CommissionEmployee.h"
11 #include "BasePlusCommissionEmployee.h"
12 using namespace std;
13
14 void virtualViaPointer( const Employee * const ); // prototype
15 void virtualViaReference( const Employee & ); // prototype
16
17 int main()
18 {
19    // set floating-point output formatting
20    cout << fixed << setprecision( 2 );
21
```

Fig. 13.23 | Employee class hierarchy driver program. (Part 1 of 7.)

```
22      // create derived-class objects
23      SalariedEmployee salariedEmployee(
24         "John", "Smith", "111-11-1111", 800 );
25      HourlyEmployee hourlyEmployee(
26         "Karen", "Price", "222-22-2222", 16.75, 40 );
27      CommissionEmployee commissionEmployee(
28         "Sue", "Jones", "333-33-3333", 10000, .06 );
29      BasePlusCommissionEmployee basePlusCommissionEmployee(
30         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
31
32      cout << "Employees processed individually using static binding:\n\n";
33
34      // output each Employee's information and earnings using static binding
35      salariedEmployee.print();
36      cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
37      hourlyEmployee.print();
38      cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
39      commissionEmployee.print();
40      cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
41      basePlusCommissionEmployee.print();
42      cout << "\nearned $" << basePlusCommissionEmployee.earnings()
43         << "\n\n";
44
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 2 of 7.)

```
45      // create vector of four base-class pointers
46      vector < Employee * > employees( 4 );
47
48      // initialize vector with Employees
49      employees[ 0 ] = &salariedEmployee;
50      employees[ 1 ] = &hourlyEmployee;
51      employees[ 2 ] = &commissionEmployee;
52      employees[ 3 ] = &basePlusCommissionEmployee;
53
54      cout << "Employees processed polymorphically via dynamic binding:\n\n";
55
56      // call virtualViaPointer to print each Employee's information
57      // and earnings using dynamic binding
58      cout << "Virtual function calls made off base-class pointers:\n\n";
59
60      for ( size_t i = 0; i < employees.size(); i++ )
61         virtualViaPointer( employees[ i ] );
62
63      // call virtualViaReference to print each Employee's information
64      // and earnings using dynamic binding
65      cout << "Virtual function calls made off base-class references:\n\n";
66
67      for ( size_t i = 0; i < employees.size(); i++ )
68         virtualViaReference( *employees[ i ] ); // note dereferencing
69   } // end main
```

Fig. 13.23 | Employee class hierarchy driver program. (Part 3 of 7.)

```
70
71 // call Employee virtual functions print and earnings off a
72 // base-class pointer using dynamic binding
73 void virtualViaPointer( const Employee * const baseClassPtr )
74 {
75    baseClassPtr->print();
76    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
77 } // end function virtualViaPointer
78
79 // call Employee virtual functions print and earnings off a
80 // base-class reference using dynamic binding
81 void virtualViaReference( const Employee &baseClassRef )
82 {
83    baseClassRef.print();
84    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
85 } // end function virtualViaReference
```

Fig. 13.23 | `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

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
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 5 of 7.)

```
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
```

Fig. 13.23 | `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

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
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 7 of 7.)

13.6.6 Demonstrating Polymorphic Processing (cont.)

- Line 46 allocates `vector employees`, which contains four `Employee` pointers.
- Line 49 aims `employees[ 0 ]` at object `salariedEmployee`.
- Line 50 aims `employees[ 1 ]` at object `hourlyEmployee`.
- Line 51 aims `employees[ 2 ]` at object `commissionEmployee`.
- Line 52 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`.

13.6.6 Demonstrating Polymorphic Processing (cont.)

- The loop in lines 60–61 traverses `vector employees` and invokes function `virtualViaPointer` (lines 73–77) 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 75) and `earnings` (line 76).
- Note that function `virtualViaPointer` does not contain any `SalariedEmployee`, `HourlyEmployee`, `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.

13.6.6 Demonstrating Polymorphic Processing (cont.)

- The `for` statement (lines 67–68) traverses `employees` and invokes function `virtualViaReference` (lines 81–85) for each element in the `vector`.
- 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 68).
- Each call to `virtualViaReference` invokes `virtual` functions `print` (line 83) and `earnings` (line 84) 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.

13.6 (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.

13.6 (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.
- 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`.
- 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.

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

- 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.
- 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.

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

- 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.
- 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`.

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

- The `earnings` function pointer in the *vtable* for class `CommissionEmployee` points to `CommissionEmployee`'s `earnings` function 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 `HourlyEmployee`, both functions override the functions in class `Employee`.

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

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

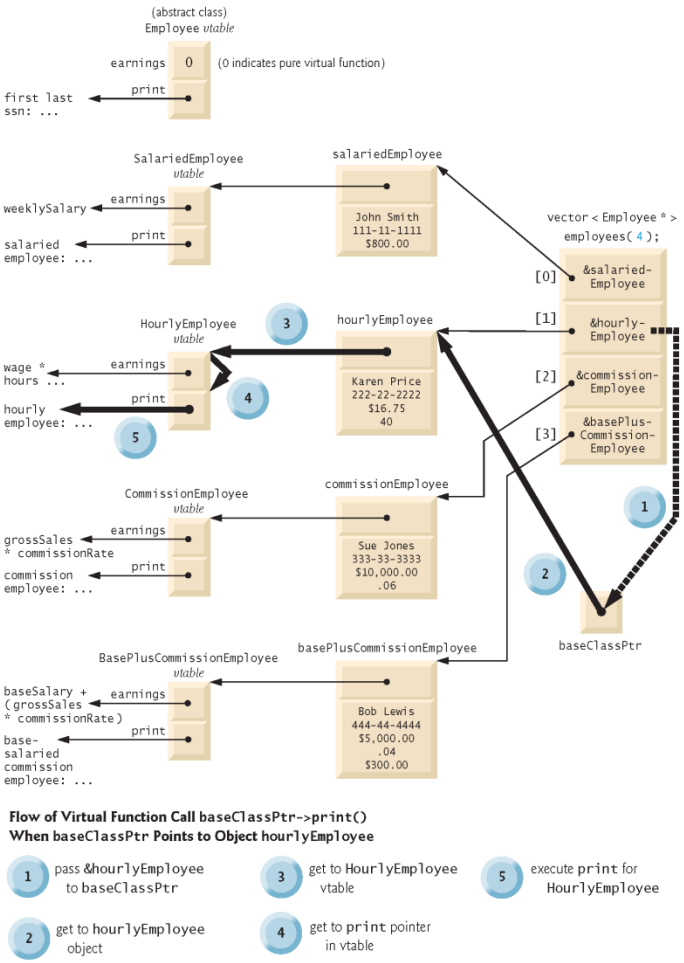

Fig. 13.24 | How `virtual` function calls work.

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

- Polymorphism is accomplished through an elegant data structure involving three levels of pointers.
- We've discussed one level—the function pointers in the *vtable*.
- These point to the actual functions that execute when a `virtual` function is invoked.
- Now we consider the second level of pointers.
- 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.
- In Fig. 13.24, these pointers are associated with the objects created in Fig. 13.23.
- Notice that the diagram displays each of the object's data member values.

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

- 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.

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

- Fig. 13.24 depicts the `vector` `employees` that contains `Employee` pointers.
- Now let's see how a typical `virtual` function call executes.
- Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 75 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.
- 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.

13.6 (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.

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

- The compiler generates code that performs the following operations.
  - Select the *i*th entry of `employees`, and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `hourlyEmployee`.
  - Dereference that pointer to get to the `hourlyEmployee` object.
  - Dereference `hourlyEmployee`'s *vtable* pointer to get to the `HourlyEmployee` *vtable*.
  - Skip the offset of four bytes to select the `print` function pointer.
  - 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 13.1

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 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. Polymorphism's overhead is acceptable for most applications. But in some situations—such as real-time applications with stringent performance requirements—polymorphism's overhead 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 files—no 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).*

13.7 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 13.6 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 13.6.6, 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.
- This section demonstrates the powerful capabilities of runtime 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.

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

- Some compilers require that RTTI be enabled before it can be used in a program.
  - In Visual C++ 2008, this option is enabled by default.
- Figure 13.25 uses the `Employee` hierarchy developed in Section 13.6 and increases by 10 percent the base salary of each `BasePlusCommissionEmployee`.

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and runtime type information.
3  // NOTE: You may need to enable RTTI on your compiler
4  // before you can execute this application.
5  #include <iostream>
6  #include <iomanip>
7  #include <vector>
8  #include <typeinfo>
9  #include "Employee.h"
10 #include "SalariedEmployee.h"
11 #include "HourlyEmployee.h"
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14 using namespace std;
15
16 int main()
17 {
18    // set floating-point output formatting
19    cout << fixed << setprecision( 2 );
20
21    // create vector of four base-class pointers
22    vector < Employee * > employees( 4 );
23
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 1 of 4.)

```
24      // initialize vector with various kinds of Employees
25      employees[ 0 ] = new SalariedEmployee(
26         "John", "Smith", "111-11-1111", 800 );
27      employees[ 1 ] = new HourlyEmployee(
28         "Karen", "Price", "222-22-2222", 16.75, 40 );
29      employees[ 2 ] = new CommissionEmployee(
30         "Sue", "Jones", "333-33-3333", 10000, .06 );
31      employees[ 3 ] = new BasePlusCommissionEmployee(
32         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
33
34      // polymorphically process each element in vector employees
35      for ( size_t i = 0; i < employees.size(); i++ )
36      {
37         employees[ i ]->print(); // output employee information
38         cout << endl;
39
40         // downcast pointer
41         BasePlusCommissionEmployee *derivedPtr =
42            dynamic_cast < BasePlusCommissionEmployee * >
43               ( employees[ i ] );
44
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 2 of 4.)

```
45          // determine whether element points to base-salaried
46          // commission employee
47          if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
48          {
49             double oldBaseSalary = derivedPtr->getBaseSalary();
50             cout << "old base salary: $" << oldBaseSalary << endl;
51             derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
52             cout << "new base salary with 10% increase is: $"
53                << derivedPtr->getBaseSalary() << endl;
54          } // end if
55
56          cout << "earned $" << employees[ i ]->earnings() << "\n\n";
57       } // end for
58
59       // release objects pointed to by vector's elements
60       for ( size_t j = 0; j < employees.size(); j++ )
61       {
62          // output class name
63          cout << "deleting object of "
64             << typeid( *employees[ j ] ).name() << endl;
65
66          delete employees[ j ];
67       } // end for
68    } // end main
```

Fig. 13.25 | 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

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
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 4 of 4.)

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

- Since we process the employees polymorphically, we cannot (with the techniques we've learned) 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 42) to determine whether the type of each object is `BasePlusCommissionEmployee`.
- This is the downcast operation we referred to in Section 13.3.3.
- Lines 41–43 dynamically downcast `employees[i]` from type `Employee *` to type `BasePlusCommissionEmployee *`.

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

- 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 41–43 is not `0`, the object is the correct type, and the `if` statement (lines 47–54) performs the special processing required for the `BasePlusCommissionEmployee` object.

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

- Operator `typeid` (line 64) 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 64) returns a pointer-based string that contains the type name (e.g., `"class BasePlusCommissionEmployee"`) of the argument passed to `typeid`.
- To use `typeid`, the program must include header file `<typeinfo>` (line 8).

Portability Tip 13.1

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

13.9 Virtual Destructors

- A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy.
- So far you've 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.

Error-Prevention Tip 13.2

If a class has `virtual` *functions, 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 13.5

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