Object-Oriented Programming: Polymorphism I

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.

1 Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies.
- Polymorphism enables us to "program in the *general*" rather than "program in the *specific*."
  - Enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.

1 Introduction

- 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.

1 Introduction

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

2 Introduction to Polymorphism: Polymorphic Video Game

Software Engineering Observation 12.1

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

Software Engineering Observation 12.2

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

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.

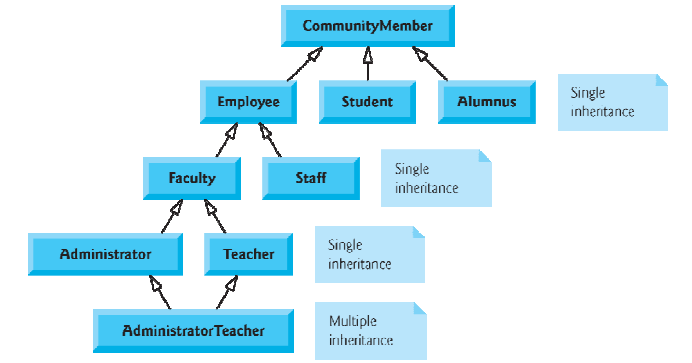


Fig. 11.2 | Inheritance hierarchy for university CommunityMembers.

3 Relationships Among Objects in an Inheritance Hierarchy

- 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.

3.1 Invoking Base-Class Functions from Derived-Class Objects

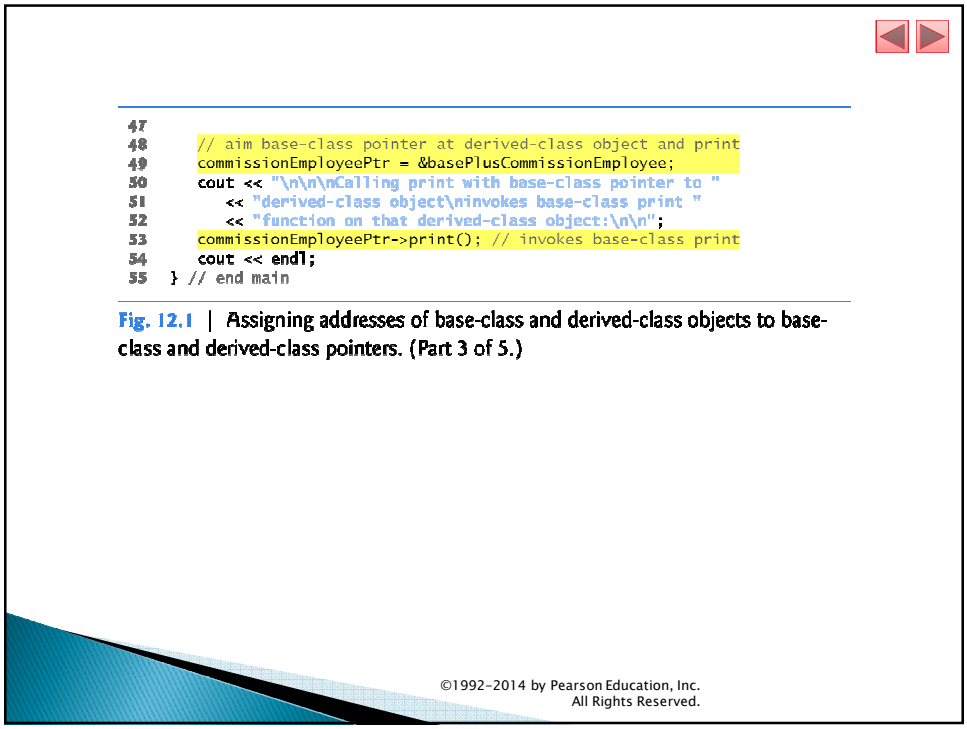
- The example in Fig. 12.1 reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee`
- The first two are natural and straightforward—we aim a base-class pointer at a base-class object and invoke base-class functionality, and we aim a derived-class pointer at a derived-class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

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

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

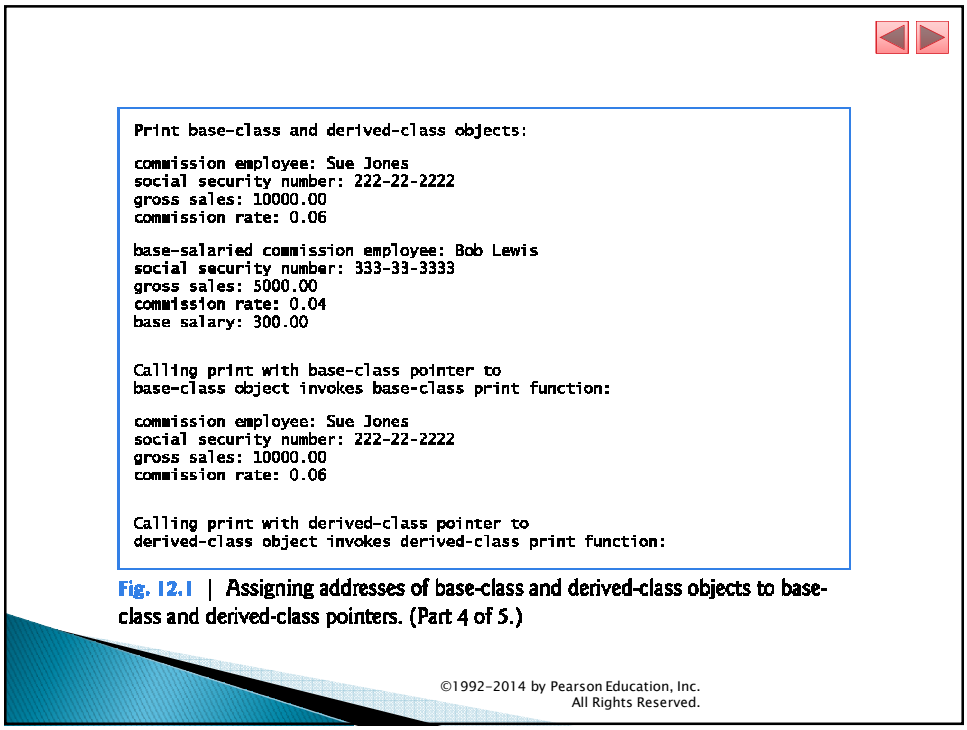
```
23    // create derived-class pointer
24    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = nullptr;
25
26    // set floating-point output formatting
27    cout << fixed << setprecision( 2 );
28
29    // output objects commissionEmployee and basePlusCommissionEmployee
30    cout << "Print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

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



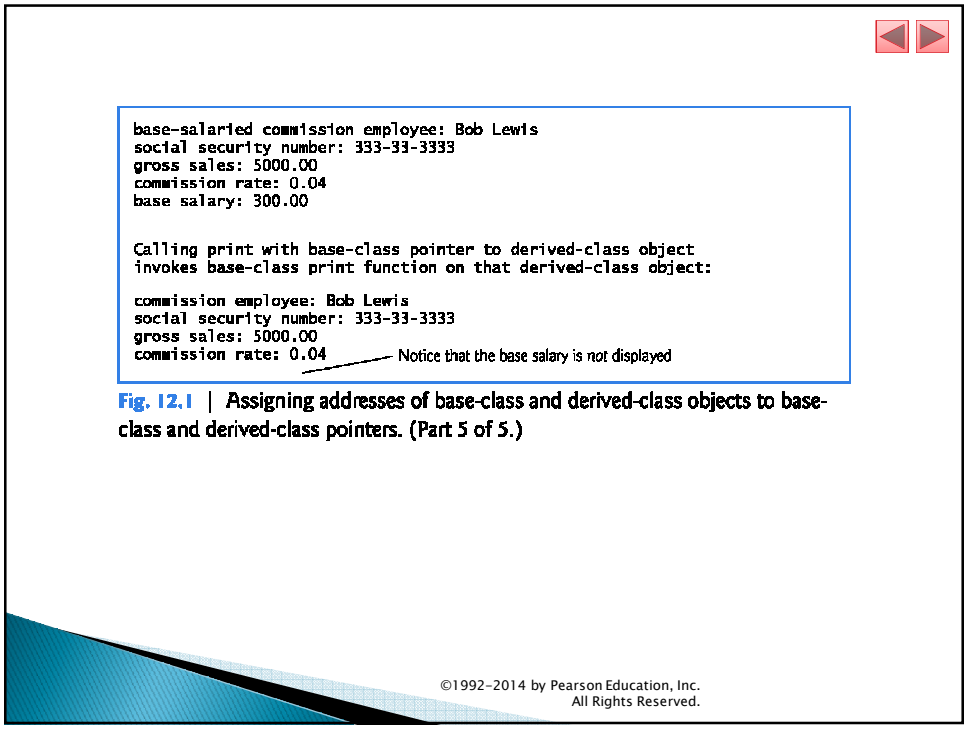
```
47
48     // aim base-class pointer at derived-class object and print
49     commissionEmployeePtr = &basePlusCommissionEmployee;
50     cout << "\n\n\nCalling print with base-class pointer to "
51        << "derived-class object\ninvokes base-class print "
52        << "function on that derived-class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

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



```
Print base-class and derived-class objects:

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

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


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

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


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

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



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

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

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

Notice that the base salary is not displayed

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

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

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

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

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

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

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

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

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

- Line 49 assigns the address of derived-class object `base-PlusCommissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 53 uses to invoke member function `print`.
  - This "crossover" is allowed because an object of a derived class *is an* object of its base class.
  - Note that despite the fact that the base class `CommissionEmployee` pointer points to a derived class `BasePlusCommissionEmployee` object, the base class `CommissionEmployee`'s `print` member function is invoked (rather than `BasePlusCommissionEmployee`'s `print` function).
- The output of each `print` member-function invocation in this program reveals that *the invoked functionality depends on the type of the pointer (or reference) used to invoke the function, not the type of the object for which the member function is called.*

3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 12.2, we aim a derived-class pointer at a base-class object.
- Line 14 attempts to assign the address of base-class object `commissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, but the C++ compiler generates an error.
- The compiler prevents this assignment, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`.


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

Microsoft Visual C++ compiler error message:

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

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

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

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

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

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

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

GNU C++ compiler error messages:

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

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

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

Downcasting

- The compiler will allow access to derived-class-only members from a base-class pointer that is aimed at a derived-class object *if* we explicitly cast the base-class pointer to a derived-class pointer—known as downcasting.
- Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer.
- After a downcast, the program *can* invoke derived-class functions that are not in the base class.
- Section 8 demonstrates how to *safely* use downcasting.

Software Engineering Observation 12.3

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

3.4 Virtual Functions and Virtual Destructors

Why virtual Functions Are Useful

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

Software Engineering Observation 12.4

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

3.4 Virtual Functions and Virtual Destructors

Declaring virtual Functions

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

Software Engineering Observation 12.5

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

Good Programming Practice 12.1

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

Software Engineering Observation 12.6

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

3.4 Virtual Functions and Virtual Destructors

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

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

3.4 Virtual Functions and Virtual Destructors

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

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

3.4 Virtual Functions and Virtual Destructors

`virtual` Functions in the `CommissionEmployee` Hierarchy

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

Error-Prevention Tip 12.1

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

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

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

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

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


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

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

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

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

3.4 Virtual Functions and Virtual Destructors

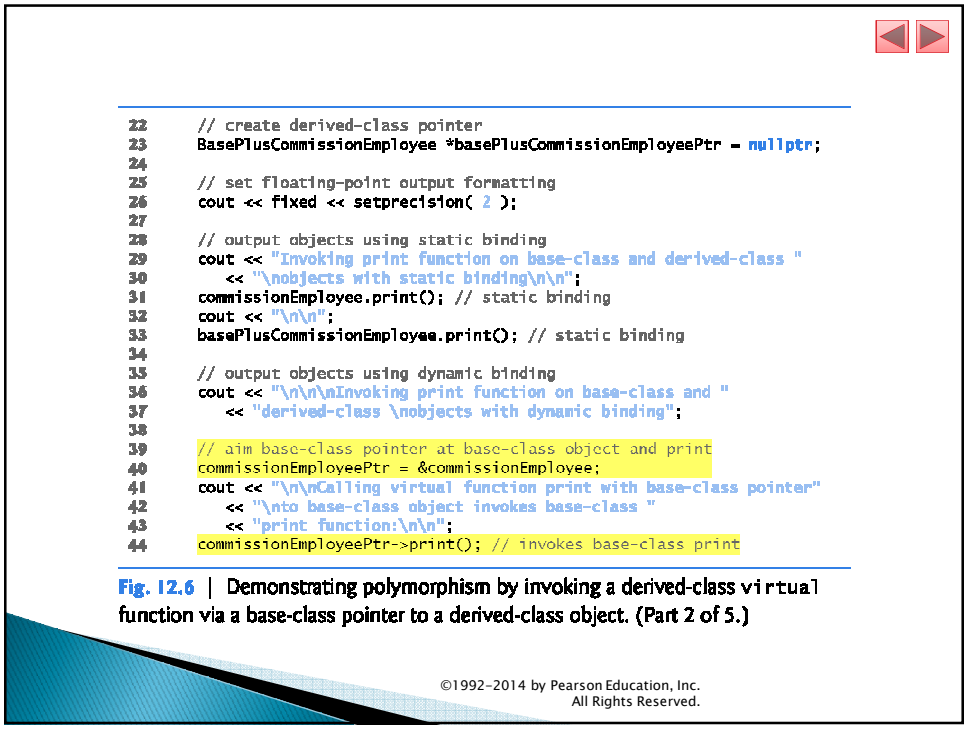
- We modified Fig. 12.1 to create the program of Fig. 12.6.
- Lines 40–51 of Fig. 12.6 demonstrate again that a `CommissionEmployee` pointer aimed at a `CommissionEmployee` object can be used to invoke `CommissionEmployee` functionality, and a `BasePlusCommissionEmployee` pointer aimed at a `BasePlusCommissionEmployee` object can be used to invoke `BasePlusCommissionEmployee` functionality.

3.4 Virtual Functions and Virtual Destructors

- Line 54 aims base-class pointer `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`.
- Note that when line 61 invokes member function `print` off the base-class pointer, the derived-class `BasePlusCommissionEmployee`'s `print` member function is invoked, so line 61 outputs different text than line 53 does in Fig. 12.1 (when member function `print` was not declared `virtual`).
- We see that declaring a member function `virtual` causes the program to dynamically determine which function to invoke *based on the type of object to which the handle points, rather than on the type of the handle*.

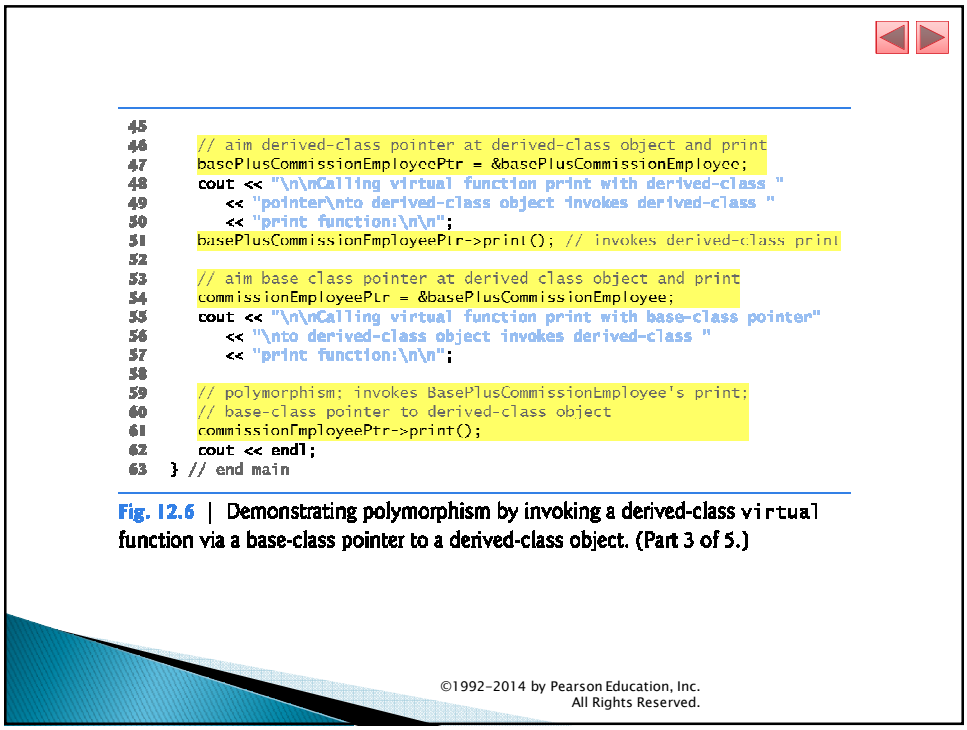
```
1  // Fig. 12.6: fig12_06.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    // create base-class object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

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



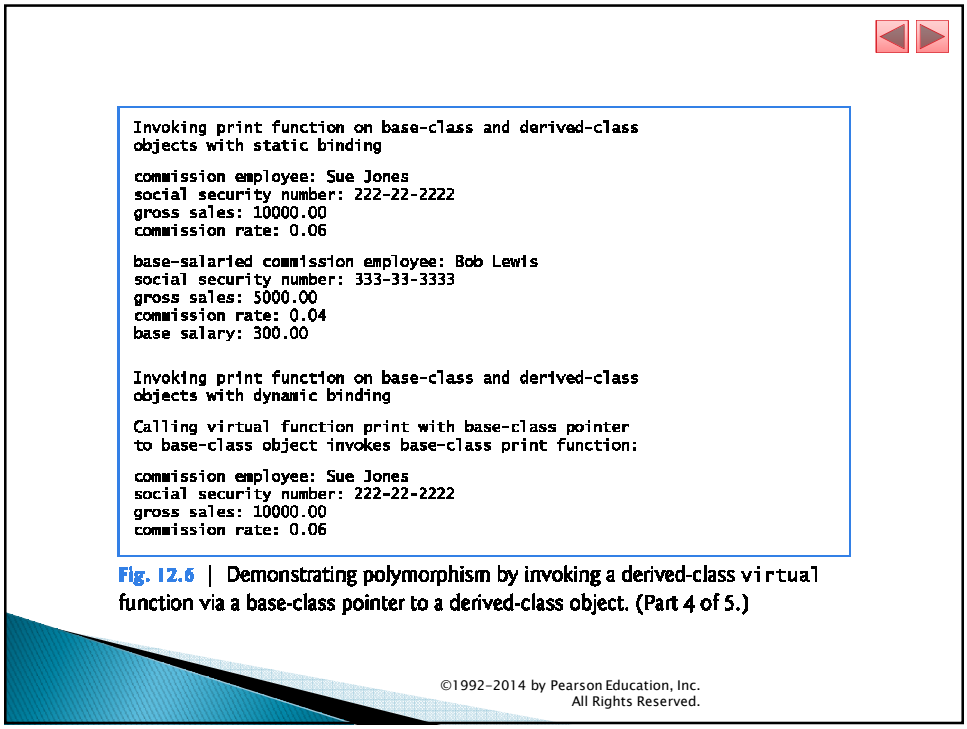
```
22     // create derived-class pointer
23     BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = nullptr;
24
25     // set floating-point output formatting
26     cout << fixed << setprecision( 2 );
27
28     // output objects using static binding
29     cout << "Invoking print function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

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



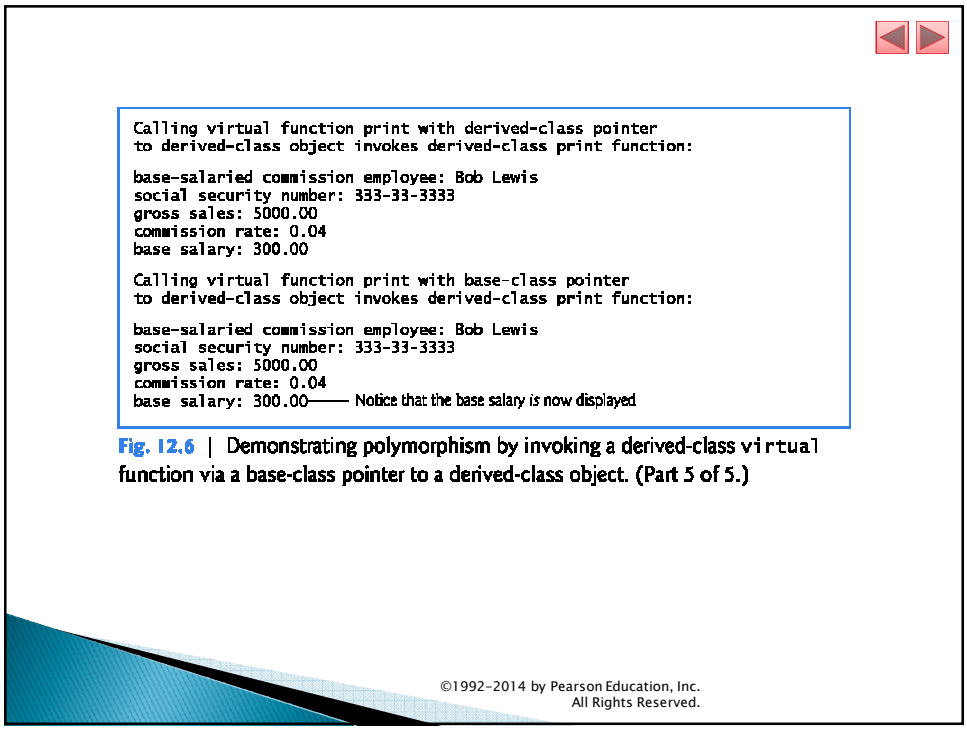
```
45
46     // aim derived-class pointer at derived-class object and print
47     basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
48     cout << "\n\nCalling virtual function print with derived-class "
49        << "pointer\nto derived-class object invokes derived-class "
50        <<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

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



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

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

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


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

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

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

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



```
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—— Notice that the base salary is now displayed
```

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

3.4 Virtual Functions and Virtual Destructors

`virtual` Destructors

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

3.4 Virtual Functions and Virtual Destructors

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

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

- Now, if an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a *base-class pointer*, the destructor for the *appropriate class* is called based on the object to which the base-class pointer points.

3.4 Virtual Functions and Virtual Destructors

- Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it's important for the destructors of both the derived and base classes to execute.
- The base-class destructor automatically executes after the derived-class destructor.

Error-Prevention Tip 12.2

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

Common Programming Error 12.1

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

3.4 Virtual Functions and Virtual Destructors

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

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

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

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

3.4 Virtual Functions and Virtual Destructors

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

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

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