```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

`virtual` Destructors

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

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

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 12.2

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 12.1

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

- In C++11, a base-class virtual function that's declared final in its prototype, as in
  `virtual` *someFunction*`(` *parameters* `) final;`
- *cannot* be overridden in any derived class—this guarantees that the base class's `final` member function definition will be used by all base-class objects and by all objects of the base class's direct and indirect derived classes.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

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

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

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.4 Type Fields and `switch` Statements

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.7

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.8

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

12.5 Abstract Classes and Pure `virtual` Functions

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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

Pure Virtual Functions

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

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

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 12.9

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.