Chapter 13 Object-Oriented Programming: Polymorphism

C++ How to Program, 8/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

- 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.

Software Engineering Observation 13.1

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 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

- A series of examples will 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 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.

```cpp
// Fig. 13.1: fig13_01.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 5.)

```
   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

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

   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "Print base-class and derived-class objects:\n\n";
   commissionEmployee.print(); // invokes base-class print
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // invokes derived-class print

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 5.)

```
   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

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

Fig. 13.1 | 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
```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 4 of 5.)

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

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


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

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

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 5 of 5.)

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.

13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 13.2, 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`.

```cpp
// Fig. 13.2: fig13_02.cpp
// Aiming a derived-class pointer at a base-class object.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // aim derived-class pointer at base-class object
   // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
   basePlusCommissionEmployeePtr = &commissionEmployee;
} // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch13\Fig13_02\fig13_02.cpp(14) : error C2440: '=' :
   cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee
*'
        Cast from base to derived requires dynamic_cast or static_cast
```

Fig. 13.2 | Aiming a derived-class pointer at a base-class object.

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

```
// Fig. 13.3: fig13_03  .cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

```

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 3.)

```
int main()
{
   CommissionEmployee *commissionEmployeePtr = 0; // base class
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;

   // invoke base-class member functions on derived-class
   // object through base-class pointer (allowed)
   string firstName = commissionEmployeePtr->getFirstName();
   string lastName = commissionEmployeePtr->getLastName();
   string ssn = commissionEmployeePtr->getSocialSecurityNumber();
   double grossSales = commissionEmployeePtr->getGrossSales();
   double commissionRate = commissionEmployeePtr->getCommissionRate();

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

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 3.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_03.cpp:26: error: 'getBaseSalary' undeclared (first use this function)
fig13_03.cpp:27: error: 'setBaseSalary' undeclared (first use this function)
```

Fig. 13.3 | 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 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.

13.3.4 Virtual Functions

- 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.

Software Engineering Observation 13.4

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.

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 13.5

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 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 class 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 13.6

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.

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

13.3.4 Virtual Functions (cont.)

- Figures 13.4–13.5 are the headers for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively.
- The only new feature in these files is that we specify each class's `earnings` and `print` member functions as `virtual` (lines 30–31 of Fig. 13.4 and lines 20–21 of Fig. 13.5).
- 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.

```cpp
// Fig. 13.4: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
      double = 0.0, double = 0.0 );

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

```

Fig. 13.4 | CommissionEmployee class header declares earnings and print as virtual. (Part 1 of 2.)

```
   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Fig. 13.4 | CommissionEmployee class header declares earnings and print as virtual. (Part 2 of 2.)

```
// Fig. 13.5: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

```

Fig. 13.5 | BasePlusCommissionEmployee class header declares earnings and print functions as virtual. (Part 1 of 2.)

```
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Fig. 13.5 | BasePlusCommissionEmployee class header declares earnings and print functions as virtual. (Part 2 of 2.)

13.3.4 Virtual Functions (cont.)

- We modified Fig. 13.1 to create the program of Fig. 13.6.
- 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. 13.1 (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.

```
// Fig. 13.6: fig13_06.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 1 of 6.)

```
    // create derived-class pointer
    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

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

    // output objects using static binding
    cout << "Invoking print function on base-class and derived-class "
       << "\nobjects with static binding\n\n";
    commissionEmployee.print(); // static binding
    cout << "\n\n";
    basePlusCommissionEmployee.print(); // static binding

    // output objects using dynamic binding
    cout << "\n\n\nInvoking print function on base-class and "
       << "derived-class \nobjects with dynamic binding";

```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 6.)

```
   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto base-class object invokes base-class "
      << "print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto derived-class object invokes derived-class "
      << "print function:\n\n";

```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 3 of 6.)

```
    // polymorphism; invokes BasePlusCommissionEmployee's print;
    // base-class pointer to derived-class object
    commissionEmployeePtr->print();
    cout << endl;
} // end main
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 4 of 6.)

```
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:
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 5 of 6.)

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

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

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

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

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

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 6 of 6.)

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 13.7

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

Software Engineering Observation 13.8

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

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

- 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 13.9

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

Common Programming Error 13.1

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

Software Engineering Observation 13.10

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

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.