Chapter 12
Object-Oriented Programming: Polymorphism

C++ How to Program, 9/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.

12.1 Introduction
12.2 Introduction to Polymorphism: Polymorphic Video Game
12.3 Relationships Among Objects in an Inheritance Hierarchy
12.3.1 Invoking Base-Class Functions from Derived-Class Objects
12.3.2 Aiming Derived-Class Pointers at Base-Class Objects
12.3.3 Derived-Class Member-Function Calls via Base-Class Pointers
12.3.4 Virtual Functions and Virtual Destructors
12.4 Type Fields and `switch` Statements
12.5 Abstract Classes and Pure `virtual` Functions
12.6 Case Study: Payroll System Using Polymorphism
12.6.1 Creating Abstract Base Class `Employee`
12.6.2 Creating Concrete Derived Class `SalariedEmployee`
12.6.3 Creating Concrete Derived Class `CommissionEmployee`
12.6.4 Creating Indirect Concrete Derived Class `BasePlusCommission-Employee`
12.6.5 Demonstrating Polymorphic Processing

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

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

12.9 Wrap-Up

12.1 Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies.
- Polymorphism enables us to “program in the *general*” rather than “program in the *specific*.”
  - Enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy’s base class.
- Polymorphism works off base-class pointer handles and base-class *reference handles*, but *not* off name handles.
- Relying on each object to know how to “do the right thing” in response to the same function call is the key concept of polymorphism.
- The same message sent to a variety of objects has “many forms” of results—hence the term polymorphism.

12.1 Introduction (cont.)

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

12.2 Introduction to Polymorphism: Polymorphic Video Game

Software Engineering Observation 12.1

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

Software Engineering Observation 12.2

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

12.3 Relationships Among Objects in an Inheritance Hierarchy

- The next several sections present a series of examples that demonstrate how base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects.
- A key concept in these examples is to demonstrate that an object of a derived class can be treated as an object of its base class.
- Despite the fact that the derived-class objects are of different types, the compiler allows this because each derived-class object *is an* object of its base class.
- However, we cannot treat a base-class object as an object of any of its derived classes.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

12.3.1 Invoking Base-Class Functions from Derived-Class Objects

- The example in Fig. 12.1 reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Section 11.3.5.
- The first two are natural and straightforward—we aim a base-class pointer at a base-class object and invoke base-class functionality, and we aim a derived-class pointer at a derived-class object and invoke derived-class functionality.
- Then, we demonstrate the relationship between derived classes and base classes (i.e., the *is-a* relationship of inheritance) by aiming a base-class pointer at a derived-class object and showing that the base-class functionality is indeed available in the derived-class object.

```
// Fig. 12.1: fig12_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 = nullptr;

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

```

Fig. 12.1 | 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 = nullptr;

   // 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

   // 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
```

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

```
    // 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. 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.)