Topic 6
Object-oriented Programming
Lecture 6b (chapter 12: polymorphism)

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

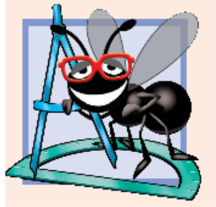
- Intro to Polymorphism
- Object relationships
- Virtual functions and destructors
- Abstract classes and pure virtual functions

12.1 Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies.
- “program in the general” rather than “program in the specific.”
  - 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.)

- 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.



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

- 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.
  - 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.
- **We cannot treat a base-class object as an 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

- 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.
- 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 derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300};

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

```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 4.)

```
   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:\n"
      << commissionEmployee.toString() // base-class toString
      << "\n\n"
      << basePlusCommissionEmployee.toString(); // derived-class toString

   // natural: aim base-class pointer at base-class object
   CommissionEmployee* commissionEmployeePtr{&commissionEmployee};
   cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
      << "\nBASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:\n"
      << commissionEmployeePtr->toString(); // base version

   // natural: aim derived-class pointer at derived-class object
   BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
      &basePlusCommissionEmployee}; // natural
   cout << "\n\nCALLING TOSTRING WITH DERIVED-CLASS POINTER TO "
      << "\nDERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
      << "TOSTRING FUNCTION:\n"
      << basePlusCommissionEmployeePtr->toString(); // derived version

```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 4.)

```
    // aim base-class pointer at derived-class object
    commissionEmployeePtr = &basePlusCommissionEmployee;
    cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
       << "DERIVED-CLASS OBJECT\nINVOKES BASE-CLASS TOSTRING "
       << "FUNCTION ON THAT DERIVED-CLASS OBJECT:\n"
       << commissionEmployeePtr->toString() // base version
       << endl;
}
```

```
DISPLAY 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
```

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 4.)

```
CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO
DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING 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 TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS OBJECT
INVOKES BASE-CLASS TOSTRING 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. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 4 of 4.)

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

- Aiming a Base-Class Pointer at a Base-Class Object
- Line 28 creates base-class pointer commissionEmployeePtr and initializes it with the address of base-class object commissionEmployee
- Line 31 uses this pointer to invoke member function toString on that CommissionEmployee object.
    - Invokes the version of toString defined in base class CommissionEmployee.

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

- Aiming a Derived-Class Pointer at a Derived-Class Object
- Lines 34-35 create basePlusCommissionEmployeePtr and initialize it with the address of derived-class object basePlusCommissionEmployee
- Line 39 uses this pointer to invoke member function toString on that BasePlusCommissionEmployee.
  - This invokes the version of toString defined in derived class BasePlusCommissionEmployee.

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

- Aiming a Base-Class Pointer at a Derived-Class Object
- Line 42 assigns the address of derived-class object basePlusCommissionEmployee to base-class pointer commissionEmployeePtr, which line 46 uses to invoke toString.
  - “crossover” allowed because of the is an relationship
  - base class CommissionEmployee’s toString member function is invoked (rather than BasePlusCommissionEmployee’s toString function).
- 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.

12.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 12.2, we attempt to aim a derived-class pointer at a base-class object.
- C++ compiler generates an error.
- The compiler prevents this assignment, because a CommissionEmployee is not a BasePlusCommissionEmployee.

```cpp
// Fig. 12.2: fig12_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};

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

Microsoft Visual C++ compiler error message:

```
c:\examples\ch12\fig12_02\fig12_02.cpp(13):
   error C2440: 'initializing': cannot convert from 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *'
```

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

12.3.3 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-only member function off a base-class pointer.

```cpp
// Fig. 12.3: fig12_03.cpp
// Attempting to invoke derived-class-only member functions
// via a base-class pointer.
#include <string>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main() {
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300};

   // aim base-class pointer at derived-class object (allowed)
   CommissionEmployee* 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()};
```

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

```
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    }
```

GNU C++ compiler error messages:

```
fig12_03.cpp:26:45: error: ‘class CommissionEmployee’ has no member named
‘getBaseSalary’
    double baseSalary{commissionEmployeePtr->getBaseSalary()};
                                             ^
fig12_03.cpp:27:27: error: ‘class CommissionEmployee’ has no member named
‘setBaseSalary’
    commissionEmployeePtr->setBaseSalary(500);
```

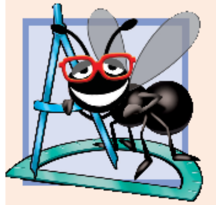
Fig. 12.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 2.)

12.3.3 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 12.8 demonstrates how to safely use downcasting.

12.4.1 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 12.3

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

12.4.2 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.
- 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 12.4

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 12.1

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

Software Engineering Observation 12.5

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.*

12.4.3 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 or a base-class reference to a derived-class, 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.

12.4.4 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, 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).

12.4.4 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 toString member functions as virtual.
- Because functions earnings and toString are virtual in class CommissionEmployee, class BasePlusCommissionEmployee's earnings and toString functions override class CommissionEmployee's.
- In addition, class BasePlusCommissionEmployee's earnings and toString functions are declare override.

Error-Prevention Tip 12.1

To help prevent errors, apply C++11's `override` *keyword to the prototype of every derived-class function that overrides a base-class* `virtual` *function. This enables the compiler to check whether the base class has a* `virtual` *member function with the same signature. If not, the compiler generates an error. Not only does this ensure that you override the base-class function with the appropriate signature, it also prevents you from accidentally hiding a base-class function that has the same name and a different signature.*

```cpp
// Fig. 12.4: CommissionEmployee.h
// CommissionEmployee class with virtual earnings and toString functions.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class

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

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

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

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

Fig. 12.4 | `CommissionEmployee` class header declares `earnings` and `toString` as `virtual`. (Part 1 of 2.)

```cpp

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

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

   virtual double earnings() const; // calculate earnings
   virtual std::string toString() const; // string representation
private:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};

#endif
```

Fig. 12.4 | `CommissionEmployee` class header declares `earnings` and `toString` as `virtual`. (Part 2 of 2.)

```cpp
// Fig. 12.5: 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

class BasePlusCommissionEmployee : public CommissionEmployee {
public:
   BasePlusCommissionEmployee(const std::string&, const std::string&,
      const std::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 override; // calculate earnings
   virtual std::string toString() const override; // string representation
private:
   double baseSalary; // base salary
};

#endif
```

Fig. 12.5 | BasePlusCommissionEmployee class header declares earnings and toString functions as virtual and override.

12.4.4 virtual Functions in the CommissionEmployee Hierarchy

- We modified Fig. 12.1 to create the program of Fig. 12.6.
- Line 47 aims base-class pointer commissionEmployeePtr at derived-class object basePlusCommissionEmployee.
- When line 54 invokes member function toString off the base-class pointer, the derived-class BasePlusCommissionEmployee’s toString member function is invoked.
- 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.

```cpp
// Fig. 12.6: fig12_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 derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee{
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300};

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

```

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.)

```
   // output objects using static binding
   cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND DERIVED-CLASS "
      << "\nOBJECTS WITH STATIC BINDING\n"
      << commissionEmployee.toString() // static binding
      << "\n\n"
      << basePlusCommissionEmployee.toString(); // static binding

   // output objects using dynamic binding
   cout << "\n\nINVOKING TOSTRING FUNCTION ON BASE-CLASS AND "
      << "\nDERIVED-CLASS OBJECTS WITH DYNAMIC BINDING";

   // natural: aim base-class pointer at base-class object
   const CommissionEmployee* commissionEmployeePtr{&commissionEmployee};
   cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
      << "\nTO BASE-CLASS OBJECT INVOKES BASE-CLASS "
      << "TOSTRING FUNCTION:\n"
      << commissionEmployeePtr->toString(); // base version

```

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.)

```
      // natural: aim derived-class pointer at derived-class object
      const BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
         &basePlusCommissionEmployee}; // natural
      cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS "
         << "POINTER\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
         << "TOSTRING FUNCTION:\n"
         << basePlusCommissionEmployeePtr->toString(); // derived version

      // aim base-class pointer at derived-class object
      commissionEmployeePtr = &basePlusCommissionEmployee;
      cout << "\n\nCALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER"
         << "\nTO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
         << "TOSTRING FUNCTION:\n";

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

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.)

12.4.6 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.

12.4.6 virtual Destructors

- 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.

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.*

12.4.6 virtual Destructors

- Destructor definition also may be written as follows:
  - virtual ~CommissionEmployee() = default;
- You can tell the compiler to explicitly generate the default version of a default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator or destructor by following the special member function's prototype with = default.
  - Useful, for example, when you explicitly define a constructor for a class and still want the compiler to generate a default constructor as well—in that case, add the following declaration to your class definition:
    - ClassName() = default;

12.4.7 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
- 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.

12.4.7 C++11: final Member Functions and Classes

- 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.

Software Engineering Observation 12.7

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

12.6 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.
- Cannot be used to instantiate objects, because they 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 concrete classes.
- Such classes define every member function they declare.

12.6 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.

12.6.1 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.

12.6.1 Pure virtual Functions

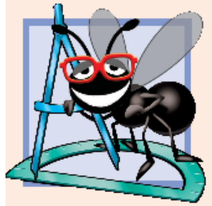
- Each concrete derived class must override all base-class pure virtual functions with concrete implementations of those functions; otherwise the derived class is also abstract.
- The difference between a virtual function and a pure virtual function is that a virtual function has an implementation and 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.

Software Engineering Observation 12.8

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

Common Programming Error 12.2

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



Software Engineering Observation 12.9

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.*

12.6.1 Pure virtual Functions

- 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.

12.7 Case Study: Payroll System Using Polymorphism

- This section reexamines the CommissionEmployee-BasePlusCommissionEmployee hierarchy that we explored throughout Section 11.3. We use an abstract class and polymorphism to perform payroll calculations based on the type of employee.

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

- Enhanced employee hierarchy to solve the following problem:
  - A company pays its employees weekly. The employees are of three types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, 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.

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

- The UML class diagram in Fig. 12.7 shows the inheritance hierarchy for our polymorphic employee payroll application.
- Abstract class name Employee is italicized, per UML convention.
- Abstract base class Employee declares the “interface” to the hierarchy—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.

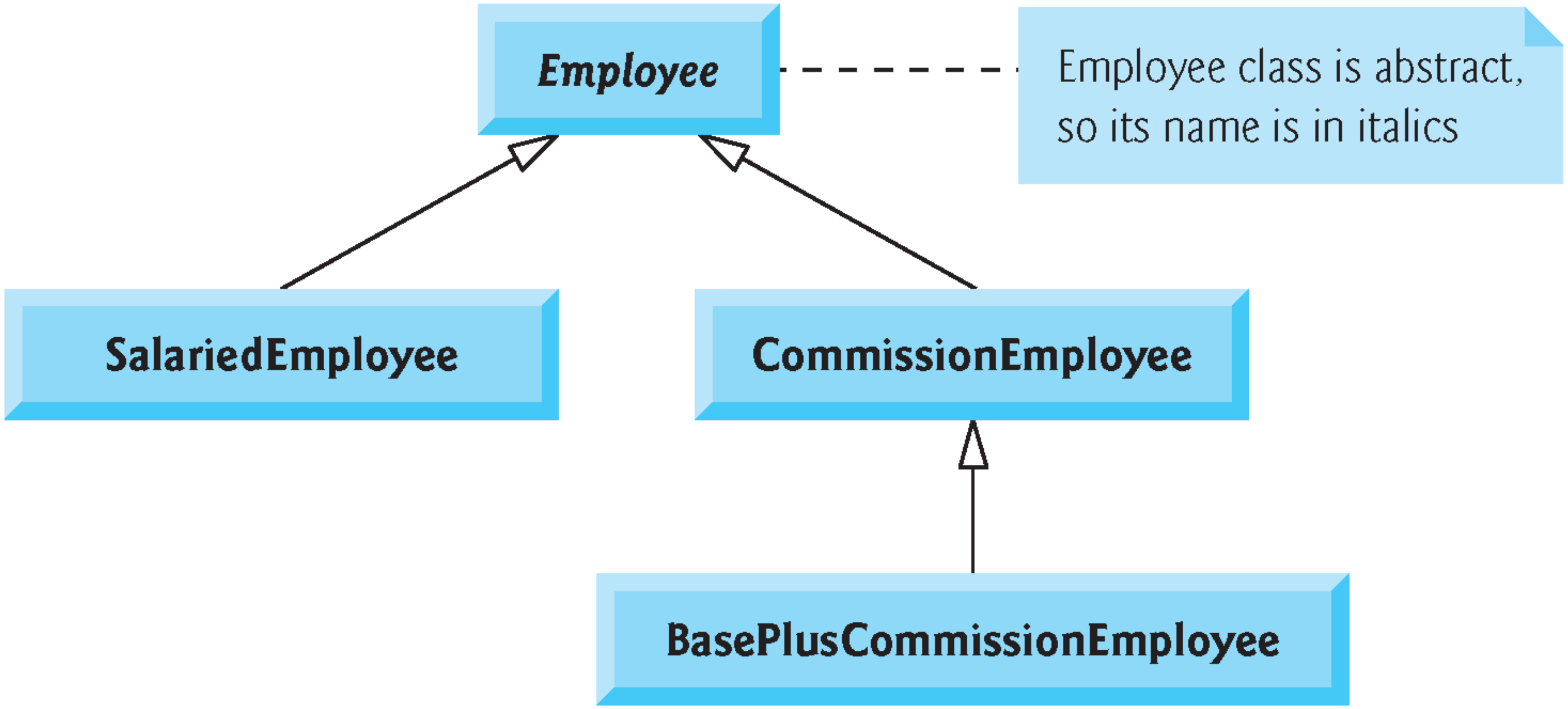


Fig. 12.7 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 12.10

A derived class can inherit interface and/or implementation from a base class. Hierarchies designed for ***implementation inheritance*** *tend to have their functionality high in the hierarchy—each derived class inherits one or more member functions from 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 by every derived class, but the individual derived classes provide their own implementations of the function(s).*

12.7.1 Creating Abstract Base Class Employee

- Class Employee (Figs. 12.9–12.10, discussed in further detail shortly) provides functions earnings and toString, in addition to various get and set functions that manipulate Employee's data members.
- An earnings function certainly applies generally 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.

12.7.1 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 object that is an Employee.
- Iterate through the vector and call earnings for each Employee object.
- C++ processes these function calls polymorphically.
- Including earnings as a pure virtual function in Employee forces every direct concrete derived class of Employee to override earnings.
- Enables the hierarchy designer to demand that each derived class provide an appropriate pay calculation, if indeed that derived class is to be concrete.

12.7.1 Creating Abstract Base Class Employee (cont.)

- Function toString in class Employee returns the first name, last name and social security number of the employee.
- Function toString could also call earnings, even though toString is a pure-virtual function in class Employee, because each concrete class is guaranteed to have an implementation of earnings.
- The diagram in Fig. 12.8 shows each of the five classes in the hierarchy down the left side and functions earnings and toString across the top.

12.7.1 Creating Abstract Base Class Employee (cont.)

- For each class, the diagram shows the desired results of each function.
- Italic text represents where the values from a particular object are used in the earnings and toString functions.
- 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`	`toString`
`Employee`	`= 0`	*firstName lastName* `social security number:` *SSN*
`Salaried-Employee`	*weeklySalary*	`salaried employee:` *firstName lastName* `social security number:` *SSN* `weekly salary:` *weeklySalary*
`Commission-Employee`	*commissionRate* `*` *grossSales*	`commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*`;` `commission rate:` *commissionRate*

`BasePlus-Commission-Employee`	`(`*commissionRate* `*` *grossSales*`) +` *baseSalary*	`base-salaried commission employee:` *firstName lastName*
		`social security number:` *SSN*
		`gross sales:` *grossSales*`;`
		`commission rate:` *commissionRate*`;`
		`base salary:` *baseSalary*

Fig. 12.8 | Polymorphic interface for the `Employee` hierarchy classes.

12.7.1 Creating Abstract Base Class Employee (cont.)

- Class Employee's header (Fig. 12.9).
- The public member functions include
  - a constructor that takes the first name, last name and social security number as arguments
  - a default virtual destructor
  - set functions for first name, last name and social security number
  - get functions for first name, last name and social security number
  - pure virtual function earnings
  - virtual function toString

```
// Fig. 12.9: Employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string> // C++ standard string class

class Employee {
public:
   Employee(const std::string&, const std::string&, const std::string &);
   virtual ~Employee() = default; // compiler generates virtual destructor

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

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

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

Fig. 12.9 | Employee abstract base class. (Part 1 of 2.)

```

   // pure virtual function makes Employee an abstract base class
   virtual double earnings() const = 0; // pure virtual
   virtual std::string toString() const; // virtual
private:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
};

#endif // EMPLOYEE_H
```

Fig. 12.9 | Employee abstract base class. (Part 2 of 2.)

12.7.1 Creating Abstract Base Class Employee (cont.)

- Figure 12.10 contains the member-function definitions for class Employee.
- No implementation is provided for virtual function earnings.

```cpp
// Fig. 12.10: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <sstream>
#include "Employee.h" // Employee class definition
using namespace std;

// constructor
Employee::Employee(const string& first, const string& last,
   const string& ssn)
   : firstName(first), lastName(last), socialSecurityNumber(ssn) {}

// set first name
void Employee::setFirstName(const string& first) {firstName = first;}

// return first name
string Employee::getFirstName() const {return firstName;}

// set last name
void Employee::setLastName(const string& last) {lastName = last;}

// return last name
string Employee::getLastName() const {return lastName;}
```

Fig. 12.10 | Employee class implementation file. (Part 1 of 2.)

```
// set social security number
void Employee::setSocialSecurityNumber(const string& ssn) {
   socialSecurityNumber = ssn; // should validate
}

// return social security number
string Employee::getSocialSecurityNumber() const {
   return socialSecurityNumber;
}

// toString Employee's information (virtual, but not pure virtual)
string Employee::toString() const {
   return getFirstName() + " "s + getLastName() +
      "\nsocial security number: "s + getSocialSecurityNumber();
}
```

Fig. 12.10 | Employee class implementation file. (Part 2 of 2.)

12.7.2 Creating Concrete Derived Class SalariedEmployee

- Class SalariedEmployee (Figs. 12.11–12.12) derives from class Employee

```
// Fig. 12.11: SalariedEmployee.h
// SalariedEmployee class derived from Employee.
#ifndef SALARIED_H
#define SALARIED_H

#include <string> // C++ standard string class
#include "Employee.h" // Employee class definition

class SalariedEmployee : public Employee {
public:
   SalariedEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0);
   virtual ~SalariedEmployee() = default; // virtual destructor

   void setWeeklySalary(double); // set weekly salary
   double getWeeklySalary() const; // return weekly salary
```

Fig. 12.11 | SalariedEmployee class header. (Part 1 of 2.)

```

      // keyword virtual signals intent to override
      virtual double earnings() const override; // calculate earnings
      virtual std::string toString() const override; // string representation
   private:
      double weeklySalary; // salary per week
   };

   #endif // SALARIED_H
```

Fig. 12.11 | SalariedEmployee class header. (Part 2 of 2.)

12.7.2 Creating Concrete Derived Class SalariedEmployee (cont.)

- Figure 12.12 contains the member-function definitions for SalariedEmployee.
- The constructor passes the first name, last name and social security number to the Employee constructor to initialize the private data members that are inherited from the base class, but not accessible in the derived class.
- earnings overrides pure virtual function earnings in Employee to provide a concrete implementation that returns the SalariedEmployee's weekly salary.

12.7.2 Creating Concrete Derived Class SalariedEmployee (cont.)

- If we did not define earnings, class SalariedEmployee would be an abstract class.
- In class SalariedEmployee's header, we declared member functions earnings and toString as virtual
  - This is redundant.
  - We defined them as virtual in Employee, so they remain virtual functions throughout the class hierarchy.

```cpp
// Fig. 12.12: SalariedEmployee.cpp
// SalariedEmployee class member-function definitions.
#include <iomanip>
#include <stdexcept>
#include <sstream>
#include "SalariedEmployee.h" // SalariedEmployee class definition
using namespace std;

// constructor
SalariedEmployee::SalariedEmployee(const string& first,
   const string& last, const string& ssn, double salary)
   : Employee(first, last, ssn) {
   setWeeklySalary(salary);
}

// set salary
void SalariedEmployee::setWeeklySalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Weekly salary must be >= 0.0");
   }

   weeklySalary = salary;
}
```

Fig. 12.12 | SalariedEmployee class implementation file. (Part 1 of 2.)

```cpp

// return salary
double SalariedEmployee::getWeeklySalary() const {return weeklySalary;}

// calculate earnings;
// override pure virtual function earnings in Employee
double SalariedEmployee::earnings() const {return getWeeklySalary();}

// return a string representation of SalariedEmployee's information
string SalariedEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "salaried employee: "
      << Employee::toString() // reuse abstract base-class function
      << "\nweekly salary: " << getWeeklySalary();
   return output.str();
}
```

Fig. 12.12 | SalariedEmployee class implementation file. (Part 2 of 2.)

12.7.2 Creating Concrete Derived Class SalariedEmployee (cont.)

- Function toString of class SalariedEmployee overrides Employee function toString.
- If class SalariedEmployee did not override toString, SalariedEmployee would inherit the Employee version of toString.

12.7.3 Creating Concrete Derived Class CommissionEmployee

- Class CommissionEmployee (Figs. 12.13–12.14) derives from Employee
- The constructor passes the first name, last name and social security number to the Employee constructor to initialize Employee’s private data members.
- Function toString calls base-class function toString to display the Employee-specific information.

```cpp
// Fig. 12.13: CommissionEmployee.h
// CommissionEmployee class derived from Employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
#include "Employee.h" // Employee class definition

class CommissionEmployee : public Employee {
public:
   CommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0);
   virtual ~CommissionEmployee() = default; // virtual destructor

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

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

   // keyword virtual signals intent to override
   virtual double earnings() const override; // calculate earnings
   virtual std::string toString() const override; // string representation
```

Fig. 12.13 | CommissionEmployee class header. (Part 1 of 2.)

```
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};

#endif // COMMISSION_H
```

Fig. 12.13 | `CommissionEmployee` class header. (Part 2 of 2.)

```cpp
// Fig. 12.14: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iomanip>
#include <stdexcept>
#include <sstream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(const string &first,
   const string &last, const string &ssn, double sales, double rate)
   : Employee(first, last, ssn) {
   setGrossSales(sales);
   setCommissionRate(rate);
}

// set gross sales amount
void CommissionEmployee::setGrossSales(double sales) {
   if (sales < 0.0) {
      throw invalid_argument("Gross sales must be >= 0.0");
   }

   grossSales = sales;
}
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 1 of 3.)

```
// return gross sales amount
double CommissionEmployee::getGrossSales() const {return grossSales;}

// set commission rate
void CommissionEmployee::setCommissionRate(double rate) {
   if (rate <= 0.0 || rate > 1.0) {
      throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
   }

   commissionRate = rate;
}

// return commission rate
double CommissionEmployee::getCommissionRate() const {
   return commissionRate;
}

// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const {
   return getCommissionRate() * getGrossSales();
}
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 2 of 3.)

```cpp
// return a string representation of CommissionEmployee's information
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "commission employee: " << Employee::toString()
      << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
   return output.str();
}
```

Fig. 12.14 | CommissionEmployee class implementation file. (Part 3 of 3.)

12.7.4 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

- Class BasePlusCommissionEmployee (Figs. 12.15–12.16) directly inherits from class CommissionEmployee and therefore is an indirect derived class of class Employee.
- BasePlusCommissionEmployee’s toString function outputs "base-salaried", followed by the output of base-class CommissionEmployee’s toString function (another example of code reuse), then the base salary.

```cpp
// Fig. 12.15: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class definition

class BasePlusCommissionEmployee : public CommissionEmployee {
public:
   BasePlusCommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0, double = 0.0);
   virtual ~BasePlusCommissionEmployee() = default; // virtual destructor

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

Fig. 12.15 | `BasePlusCommissionEmployee` class header. (Part 1 of 2.)

```
    // keyword virtual signals intent to override
    virtual double earnings() const override; // calculate earnings
    virtual std::string toString() const override; // string representation
private:
    double baseSalary; // base salary per week
};

#endif // BASEPLUS_H
```

Fig. 12.15 | BasePlusCommissionEmployee class header. (Part 2 of 2.)

```cpp
// Fig. 12.16: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iomanip>
#include <stdexcept>
#include <sstream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string& first, const string& last, const string& ssn,
   double sales, double rate, double salary)
   : CommissionEmployee(first, last, ssn, sales, rate) {
   setBaseSalary(salary); // validate and store base salary
}

// set base salary
void BasePlusCommissionEmployee::setBaseSalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Salary must be >= 0.0");
   }

   baseSalary = salary;
}
```

Fig. 12.16 | `BasePlusCommissionEmployee` class implementation file. (Part 1 of 2.)

```cpp

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const {
    return baseSalary;
}

// calculate earnings;
// override virtual function earnings in CommissionEmployee
double BasePlusCommissionEmployee::earnings() const {
    return getBaseSalary() + CommissionEmployee::earnings();
}

// return a string representation of a BasePlusCommissionEmployee
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2);
   output << "base-salaried " << CommissionEmployee::toString()
      << "; base salary: " << getBaseSalary();
   return output.str();
}
```

Fig. 12.16 | BasePlusCommissionEmployee class implementation file. (Part 2 of 2.)

12.7.5 Demonstrating Polymorphic Processing

- Fig. 12.17 creates an object of each of the four concrete classes SalariedEmployee, CommissionEmployee and BasePlusCommissionEmployee.
- Manipulates these objects, first with static binding, then polymorphically, using a vector of Employee pointers.
- Lines 20–25 create objects of each of the four concrete Employee derived classes.
- Lines 28–34 output each Employee’s information and earnings.
  - Each member-function invocation is an example of static binding—at compile time, because we are using name, the compiler can identify each object’s type to determine which toString and earnings functions are called.

```cpp
// Fig. 12.17: fig12_17.cpp
// Processing Employee derived-class objects with static binding
// then polymorphically using dynamic binding.
#include <iostream>
#include <iomanip>
#include <vector>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

void virtualViaPointer(const Employee* const); // prototype
void virtualViaReference(const Employee&); // prototype

int main() {
   cout << fixed << setprecision(2); // set floating-point formatting

```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 1 of 7.)

```
    // create derived-class objects
    SalariedEmployee salariedEmployee{
       "John", "Smith", "111-11-1111", 800};
    CommissionEmployee commissionEmployee{
       "Sue", "Jones", "333-33-3333", 10000, .06};
    BasePlusCommissionEmployee basePlusCommissionEmployee{
       "Bob", "Lewis", "444-44-4444", 5000, .04, 300};

    // output each Employee's information and earnings using static binding
    cout << "EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING\n"
       << salariedEmployee.toString()
       << "\nearned $" << salariedEmployee.earnings() << "\n\n"
       << commissionEmployee.toString()
       << "\nearned $" << commissionEmployee.earnings() << "\n\n"
       << basePlusCommissionEmployee.toString()
       << "\nearned $" << basePlusCommissionEmployee.earnings() << "\n\n";

```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 2 of 7.)

```
   // create and initialize vector of three base-class pointers
   vector<Employee *> employees{&salariedEmployee, &commissionEmployee,
      &basePlusCommissionEmployee};

   cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING\n\n";

   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS POINTERS\n";

   for (const Employee* employeePtr : employees) {
      virtualViaPointer(employeePtr);
   }

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS REFERENCES\n";

   for (const Employee* employeePtr : employees) {
      virtualViaReference(*employeePtr); // note dereferencing
   }
}
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 3 of 7.)

```cpp

// call Employee virtual functions toString and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer(const Employee* const baseClassPtr) {
    cout << baseClassPtr->toString()
       << "\nearned $" << baseClassPtr->earnings() << "\n\n";
}

// call Employee virtual functions toString and earnings off a
// base-class reference using dynamic binding
void virtualViaReference(const Employee& baseClassRef) {
    cout << baseClassRef.toString()
       << "\nearned $" << baseClassRef.earnings() << "\n\n";
}
```

Fig. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (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

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. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 5 of 7.)

```
EMPLOYEES PROCESSED POLYMORPHICALLY VIA 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

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. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (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

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. 12.17 | Processing `Employee` derived-class objects with static binding then polymorphically using dynamic binding. (Part 7 of 7.)

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 37-38 create and initialize the vector employees, which contains three Employee pointers.
- The compiler allows these assignments, because a SalariedEmployee is an Employee, a CommissionEmployee is an Employee and a BasePlusCommissionEmployee is an Employee.

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 46–48 traverse vector employees and invoke function virtualViaPointer 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 toString and earnings
- Function virtualViaPointer does not contain any SalariedEmployee, 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.

12.7.5 Demonstrating Polymorphic Processing (cont.)

- Lines 54–59 traverse employees and invoke function virtualViaReference for each vector element.
- Function virtualViaReference receives in its parameter baseClassRef (of type const Employee &) a reference to the object obtained by dereferencing an employees element pointer
- Each call to virtualViaReference invokes virtual functions toString and earnings via 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.