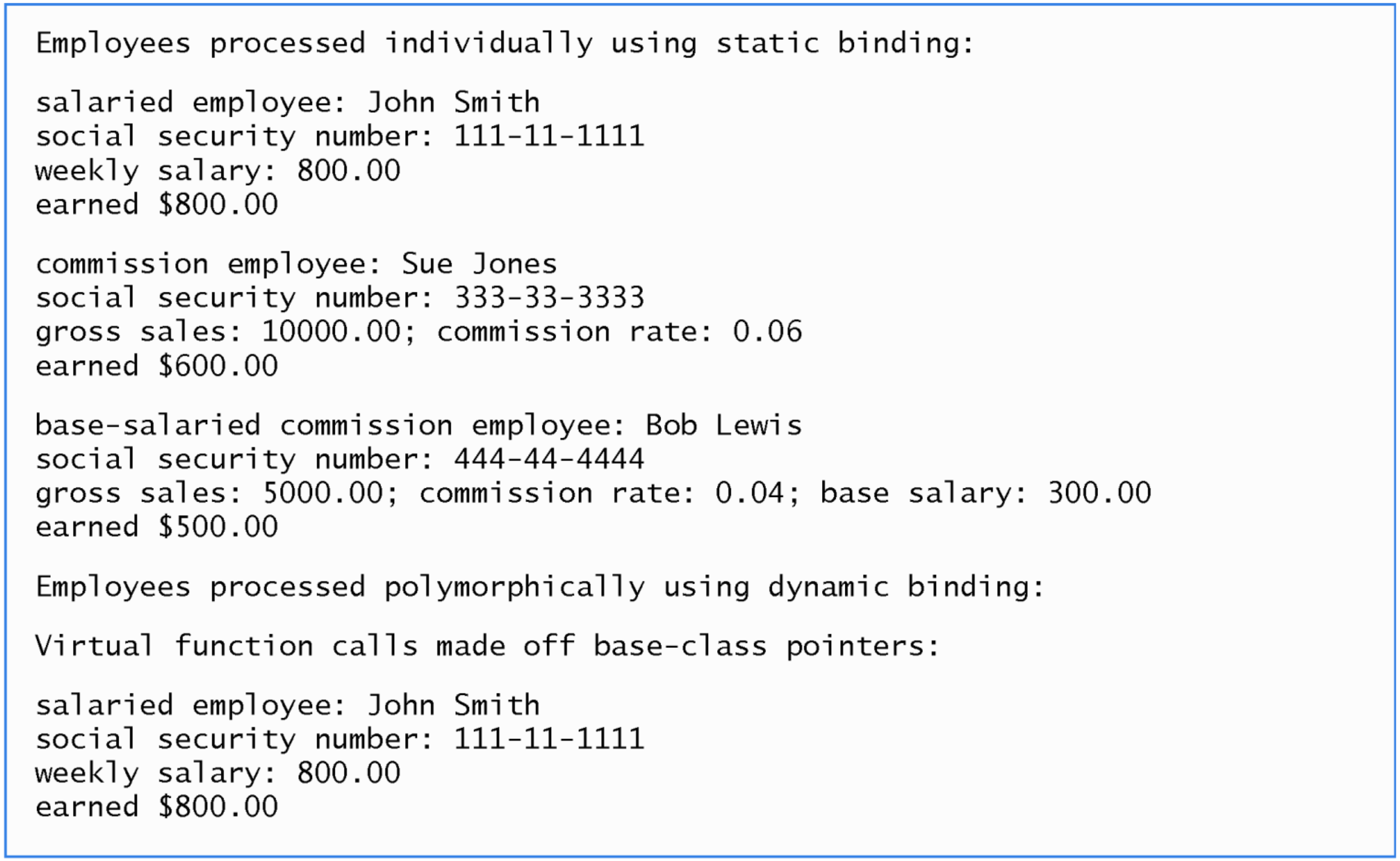

```
Employees processed individually using static binding:

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

Employees processed polymorphically using 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
```

**Fig. 12.17** | Employee class hierarchy driver program. (Part 5 of 7.)

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

```
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** | `Employee` class hierarchy driver program. (Part 6 of 7.)

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

```
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** | `Employee` class hierarchy driver program. (Part 7 of 7.)

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

# 12.6.5 Demonstrating Polymorphic Processing (cont.)

- Line 41 creates the `vector employees`, which contains three `Employee` pointers.
- Line 44 aims `employees[ 0 ]` at object `salariedEmployee`.
- Line 45 aims `employees[ 1 ]` at object `commissionEmployee`.
- Line 46 aims `employees[ 2 ]` at object `basePlusCommissionEmployee`.
- The compiler allows these assignments, because a `SalariedEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`*.*

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

# 12.6.5 Demonstrating Polymorphic Processing (cont.)

- Lines 54–55 traverse `vector` `employees` and invoke function `virtualViaPointer` (lines 67–71) 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 `print` (line 69) and `earnings` (line 70).
- Note that 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.

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

# 12.6.5 Demonstrating Polymorphic Processing (cont.)

- Lines 61–62 traverse `employees` and invoke function `virtualViaReference` (lines 75–79) for each `vector` element.
- Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a reference to the object obtained by dereferencing the pointer stored in each `employees` element (line 62).
- Each call to `virtualViaReference` invokes `virtual` functions `print` (line 77) and `earnings` (line 78) 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.
- This is another example of *dynamic binding*.
- The output produced using base-class references is identical to the output produced using base-class pointers.

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

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

- This section discusses how C++ can implement polymorphism, `virtual` functions and dynamic binding internally.
- This will give you a solid understanding of how these capabilities really work.
- More importantly, it will help you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time.
- You'll see that polymorphism is accomplished through three levels of pointers (i.e., "triple indirection").
- Then we'll show how an executing program uses these data structures to execute `virtual` functions and achieve the dynamic binding associated with polymorphism.
- Our discussion explains one possible implementation; this is not a language requirement.

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

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

- When C++ compiles a class that has one or more `virtual` functions, it builds a virtual function table (*vtable*) for that class.
- The *vtable* contains pointers to the class's `virtual` functions.
- Just as the name of a built-in array contains the address in memory of the array's first element, a pointer to a function contains the starting address in memory of the code that performs the function's task.
- An executing program uses the *vtable* to select the proper function implementation each time a `virtual` function of that class is called.
- The leftmost column of Fig. 12.18 illustrates the *vtables* for classes `Employee`, `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.

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

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

***Employee Class* vtable**

- In the `Employee` class *vtable*, the first function pointer is set to 0 (i.e., the `nullptr`), because function `earnings` is a *pure* `virtual` function and therefore lacks an implementation.
- The second function pointer points to function `print`, which displays the employee's full name and social security number.
- Any class that has one or more null pointers in its *vtable* is an *abstract* class.
- Classes without any null *vtable* pointers are concrete classes.

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

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

***SalariedEmployee* Class vtable**

- Class SalariedEmployee overrides function earnings to return the employee's weekly salary, so the function pointer points to the earnings function of class SalariedEmployee.
- SalariedEmployee also overrides print, so the corresponding function pointer points to the SalariedEmployee member function that prints "salaried employee: " followed by the employee's name, social security number and weekly salary.

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

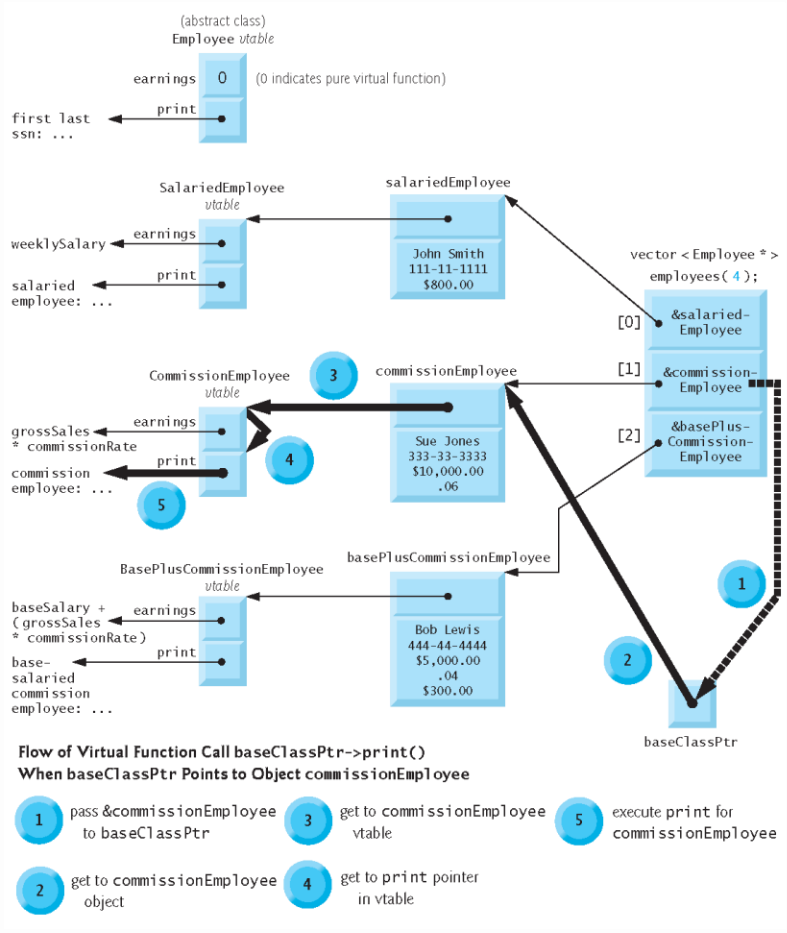

Fig. 12.18 | How virtual function calls work.

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

# 12.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” (cont.)

***CommissionEmployee* *Class* vtable**

- The earnings function pointer in the *vtable* for class CommissionEmployee points to CommissionEmployee’s earnings function that returns the employee’s gross sales multiplied by the commission rate.
- The print function pointer points to the CommissionEmployee version of the function, which prints the employee’s type, name, social security number, commission rate and gross sales.
- As in class SalariedEmployee, both functions override the functions in class Employee.

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

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

## *BasePlusCommissionEmployee Class* vtable

- The `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee` points to the `BasePlusCommissionEmployee`'s `earnings` function, which returns the employee's base salary plus gross sales multiplied by commission rate.
- The `print` function pointer points to the `BasePlusCommissionEmployee` version of the function, which prints the employee's base salary plus the type, name, social security number, commission rate and gross sales.
- Both functions override the functions in class `CommissionEmployee`.

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

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

***Three Levels of Pointers to Implement Polymorphism***

- Polymorphism is accomplished through an elegant data structure involving three levels of pointers.
- We've discussed one level—the function pointers in the *vtable.*
- These point to the actual functions that execute when a `virtual` function is invoked.
- Now we consider the second level of pointers.
- *Whenever an object of a class with one or more `virtual` functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class.*
- This pointer is normally at the front of the object, but it isn't required to be implemented that way.

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

# 12.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” (cont.)

- In Fig. 12.18, these pointers are associated with the objects created in Fig. 12.17.
- Notice that the diagram displays each of the object’s data member values.
- The third level of pointers simply contains the handles to the objects that receive the `virtual` function calls.
- The handles in this level may also be references.
- Fig. 12.18 depicts the `vector` `employees` that contains `Employee` pointers.

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