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

- Now let's see how a typical `virtual` function call executes.
- Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 69 of Fig. 12.17).
- Assume that `baseClassPtr` contains `employees[ 1 ]` (i.e., the address of object `commissionEmployee` in `employees`).
- When the compiler compiles this statement, it determines that the call is indeed being made via a *base-class pointer* and that `print` is a `virtual` function.
- The compiler determines that `print` is the *second* entry in each of the *vtables.*
- To locate this entry, the compiler notes that it will need to skip the first entry.

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

- Thus, the compiler compiles an offset or displacement of four bytes (four bytes for each pointer on today’s popular 32-bit machines, and only one pointer needs to be skipped) into the table of machine-language object-code pointers to find the code that will execute the `virtual` function call.

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

- The compiler generates code that performs the following operations.
  1. Select the *i^th^* entry of `employees`, and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `commissionEmployee`.
  2. *Dereference* that pointer to get to the `commissionEmployee` object.
  3. *Dereference* `commissionEmployee`'s *vtable* pointer to get to the `CommissionEmployee` *vtable.*
  4. Skip the offset of four bytes to select the `print` function pointer.
  5. *Dereference* the `print` function pointer to form the "name" of the actual function to execute, and use the function call operator `()` to execute the appropriate `print` function.

Performance Tip 12.1

Polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient. You can use these capabilities with nominal impact on performance.

Performance Tip 12.2

Virtual functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that's nearly as efficient as hand-coded `switch`-based logic. Polymorphism's overhead is acceptable for most applications. In some situations—such as real-time applications with stringent performance requirements—polymorphism's overhead may be too high.

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

- Recall from the problem statement at the beginning of Section 12.6 that, for the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployee`s by adding 10 percent to their base salaries.
- When processing `Employee` objects polymorphically in Section 12.6.5, we did not need to worry about the "specifics."
- To adjust the base salaries of `BasePlusCommissionEmployee`s, we have to determine the specific type of each `Employee` object at execution time, then act appropriately.

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

- This section demonstrates the powerful capabilities of runtime type information (RTTI) and dynamic casting, which enable a program to determine an object's type at execution time and act on that object accordingly.
- Figure 12.19 uses the `Employee` hierarchy developed in Section 12.6 and increases by 10 percent the base salary of each `BasePlusCommissionEmployee`.

```cpp
// Fig. 12.19: fig12_19.cpp
// Demonstrating downcasting and runtime type information.
// NOTE: You may need to enable RTTI on your compiler
// before you can compile this application.
#include <iostream>
#include <iomanip>
#include <vector>
#include <typeinfo>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

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

   // create vector of three base-class pointers
   vector < Employee * > employees( 3 );

```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 1 of 4.)

```cpp
   // initialize vector with various kinds of Employees
   employees[ 0 ] = new SalariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   employees[ 1 ] = new CommissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   employees[ 2 ] = new BasePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   // polymorphically process each element in vector employees
   for ( Employee *employeePtr : employees )
   {
      employeePtr->print(); // output employee information
      cout << endl;

      // attempt to downcast pointer
      BasePlusCommissionEmployee *derivedPtr =
         dynamic_cast < BasePlusCommissionEmployee * >( employeePtr );

```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 2 of 4.)

```
        // determine whether element points to a BasePlusCommissionEmployee
        if ( derivedPtr != nullptr ) // true for "is a" relationship
        {
           double oldBaseSalary = derivedPtr->getBaseSalary();
           cout << "old base salary: $" << oldBaseSalary << endl;
           derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
           cout << "new base salary with 10% increase is: $"
              << derivedPtr->getBaseSalary() << endl;
        } // end if

        cout << "earned $" << employeePtr->earnings() << "\n\n";
     } // end for

     // release objects pointed to by vector's elements
     for ( const Employee *employeePtr : employees )
     {
        // output class name
        cout << "deleting object of "
           << typeid( *employeePtr ).name() << endl;

        delete employeePtr;
     } // end for
  } // end main
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 3 of 4.)

```
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
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Fig. 12.19 | Demonstrating downcasting and runtime type information. (Part 4 of 4.)

12.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- Since we process the Employees polymorphically, we cannot (with the techniques you've learned so far) be certain as to which type of Employee is being manipulated at any given time.
- BasePlusCommissionEmployee employees *must* be identified when we encounter them so they can receive the 10 percent salary increase.
- To accomplish this, we use operator dynamic_cast (line 39) to determine whether the current Employee's type is BasePlusCommissionEmployee.
- This is the *downcast* operation we referred to in Section 12.3.3.
- Lines 38–39 dynamically downcast employeePtr from type Employee * to type BasePlusCommissionEmployee *.

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

- If `employeePtr` element points to an object that *is a* `BasePlusCommissionEmployee` object, then that object's address is assigned to derived-class pointer `derivedPtr`; otherwise, `nullptr` is assigned to `derivedPtr`.
- Note that `dynamic_cast` rather than `static_cast` is *required* here to perform type checking on the underlying object—a `static_cast` would simply cast the `Employee *` to a `BasePlusCommissionEmployee *` regardless of the underlying object's type.

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

- With a `static_cast`, the program would attempt to increase every `Employee`'s base salary, resulting in undefined behavior for each object that is not a `BasePlusCommissionEmployee`.
- If the value returned by the `dynamic_cast` operator in lines 38–39 *is not* `nullptr`, the object *is* the correct type, and the `if` statement (lines 42–49) performs the special processing required for the `BasePlusCommissionEmployee` object.

12.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- Operator typeid (line 59) returns a *reference* to an object of class type_info that contains the information about the type of its operand, including the name of that type.
- When invoked, type_info member function name (line 59) returns a pointer-based string containing the typeid argument's type name (e.g., "class BasePlusCommissionEmployee").
- To use typeid, the program must include header <typeinfo> (line 8).

Portability Tip 12.1

The string returned by `type_info` member function `name` may vary by compiler.