Chapter 13 Object-Oriented Programming: Polymorphism

Yu-Shuen Wang, CS, NCTU

13.1 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."
- 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.

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 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.

13.2 Polymorphism Examples

- With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked.
- If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object.
- Polymorphism occurs when a program invokes a `virtual` function through a base-class pointer or reference.

Software Engineering Observation 13.1

With `virtual` *functions and polymorphism, you can 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.*

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
```

Fig. 13.1 | `CommissionEmployee` class header file. (Part 1 of 2.)

```
24      void setGrossSales( double ); // set gross sales amount
25      double getGrossSales() const; // return gross sales amount
26
27      void setCommissionRate( double ); // set commission rate
28      double getCommissionRate() const; // return commission rate
29
30      double earnings() const; // calculate earnings
31      void print() const; // print CommissionEmployee object
32   private:
33      string firstName;
34      string lastName;
35      string socialSecurityNumber;
36      double grossSales; // gross weekly sales
37      double commissionRate; // commission percentage
38   }; // end class CommissionEmployee
39
40   #endif
```

Fig. 13.1 | `CommissionEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.3: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9  using namespace std;
10
11 class BasePlusCommissionEmployee : public CommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17    void setBaseSalary( double ); // set base salary
18    double getBaseSalary() const; // return base salary
19
20    double earnings() const; // calculate earnings
21    void print() const; // print BasePlusCommissionEmployee object
```

Fig. 13.3 | `BasePlusCommissionEmployee` class header file. (Part 1 of 2.)

```
22   private:
23      double baseSalary; // base salary
24   }; // end class BasePlusCommissionEmployee
25
26   #endif
```

Fig. 13.3 | `BasePlusCommissionEmployee` class header file. (Part 2 of 2.)

```
23   // return base salary
24   double BasePlusCommissionEmployee::getBaseSalary() const
25   {
26      return baseSalary;
27   } // end function getBaseSalary
28
29   // calculate earnings
30   double BasePlusCommissionEmployee::earnings() const
31   {
32      return getBaseSalary() + CommissionEmployee::earnings();
33   } // end function earnings
34
35   // print BasePlusCommissionEmployee object
36   void BasePlusCommissionEmployee::print() const
37   {
38      cout << "base-salaried ";
39
40      // invoke CommissionEmployee's print function
41      CommissionEmployee::print();
42
43      cout << "\nbase salary: " << getBaseSalary();
44   } // end function print
```

Fig. 13.4 | `BasePlusCommissionEmployee` class implementation file. (Part 2 of 2.)

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  #include <iomanip>
6  #include "CommissionEmployee.h"
7  #include "BasePlusCommissionEmployee.h"
8  using namespace std;
9
10 int main()
11 {
12    // create base-class object
13    CommissionEmployee commissionEmployee(
14       "Sue", "Jones", "222-22-2222", 10000, .06 );
15
16    // create base-class pointer
17    CommissionEmployee *commissionEmployeePtr = 0;
18
19    // create derived-class object
20    BasePlusCommissionEmployee basePlusCommissionEmployee(
21       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
22
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 5.)

```
23      // create derived-class pointer
24      BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
25
26      // set floating-point output formatting
27      cout << fixed << setprecision( 2 );
28
29      // output objects commissionEmployee and basePlusCommissionEmployee
30      cout << "Print base-class and derived-class objects:\n\n";
31      commissionEmployee.print(); // invokes base-class print
32      cout << "\n\n";
33      basePlusCommissionEmployee.print(); // invokes derived-class print
34
35      // aim base-class pointer at base-class object and print
36      commissionEmployeePtr = &commissionEmployee; // perfectly natural
37      cout << "\n\n\nCalling print with base-class pointer to "
38         << "\nbase-class object invokes base-class print function:\n\n";
39      commissionEmployeePtr->print(); // invokes base-class print
40
41      // aim derived-class pointer at derived-class object and print
42      basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
43      cout << "\n\n\nCalling print with derived-class pointer to "
44         << "\nderived-class object invokes derived-class "
45         << "print function:\n\n";
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 5.)

```
46      basePlusCommissionEmployeePtr->print(); // invokes derived-class print
47
48      // aim base-class pointer at derived-class object and print
49      commissionEmployeePtr = &basePlusCommissionEmployee;
50      cout << "\n\n\nCalling print with base-class pointer to "
51         << "derived-class object\ninvokes base-class print "
52         << "function on that derived-class object:\n\n";
53      commissionEmployeePtr->print(); // invokes base-class print
54      cout << endl;
55   } // end main
```

```
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
```

Fig. 13.5 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 5.)

```
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:

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

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

Failure example

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15 } // end main
```

Fig. 13.6 | Aiming a derived-class pointer at a base-class object. (Part 1 of 2.)

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.

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer (allowed)
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
```

Fig. 13.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 3.)

```
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   } // end main
```

Fig. 13.7 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 3.)

13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers (cont.)

- The compiler allows 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.

Software Engineering Observation 13.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it's acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to send that derived-class object messages that do not appear in the base class.

13.3.4 Virtual Functions (cont.)

- If we declare the base-class function as `virtual`, we can override that function to enable polymorphic behavior.
- For example, 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.

Software Engineering Observation 13.4

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 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 hierarchy to promote program clarity.*

Software Engineering Observation 13.5

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.*

13.3.4 Virtual Functions (cont.)

- If a program invokes a `virtual` function through a base-class pointer to a derived-class, the program will choose the correct derived-class function dynamically (i.e., at execution time) based on the object type.
  - Known as dynamic binding or late binding.
- When a `virtual` function is called by referencing a specific object, 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.

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
```

Fig. 13.8 | `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 1 of 2.)

```
23
24      void setGrossSales( double ); // set gross sales amount
25      double getGrossSales() const; // return gross sales amount
26
27      void setCommissionRate( double ); // set commission rate
28      double getCommissionRate() const; // return commission rate
29
30      virtual double earnings() const; // calculate earnings
31      virtual void print() const; // print CommissionEmployee object
32   private:
33      string firstName;
34      string lastName;
35      string socialSecurityNumber;
36      double grossSales; // gross weekly sales
37      double commissionRate; // commission percentage
38   }; // end class CommissionEmployee
39
40   #endif
```

Fig. 13.8 | `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 2 of 2.)

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9  using namespace std;
10
11 class BasePlusCommissionEmployee : public CommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17    void setBaseSalary( double ); // set base salary
18    double getBaseSalary() const; // return base salary
19
20    virtual double earnings() const; // calculate earnings
21    virtual void print() const; // print BasePlusCommissionEmployee object
```

Fig. 13.9 | `BasePlusCommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 1 of 2.)

```
22   private:
23      double baseSalary; // base salary
24   }; // end class BasePlusCommissionEmployee
25
26   #endif
```

Fig. 13.9 | `BasePlusCommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`. (Part 2 of 2.)

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    // create base-class object
12    CommissionEmployee commissionEmployee(
13       "Sue", "Jones", "222-22-2222", 10000, .06 );
14
15    // create base-class pointer
16    CommissionEmployee *commissionEmployeePtr = 0;
17
18    // create derived-class object
19    BasePlusCommissionEmployee basePlusCommissionEmployee(
20       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
21
```

Fig. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 1 of 5.)

```
22      // create derived-class pointer
23      BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
24
25      // set floating-point output formatting
26      cout << fixed << setprecision( 2 );
27
28      // output objects using static binding
29      cout << "Invoking print function on base-class and derived-class "
30         << "\nobjects with static binding\n\n";
31      commissionEmployee.print(); // static binding
32      cout << "\n\n";
33      basePlusCommissionEmployee.print(); // static binding
34
35      // output objects using dynamic binding
36      cout << "\n\n\nInvoking print function on base-class and "
37         << "derived-class \nobjects with dynamic binding";
38
39      // aim base-class pointer at base-class object and print
40      commissionEmployeePtr = &commissionEmployee;
41      cout << "\n\nCalling virtual function print with base-class pointer"
42         << "\nto base-class object invokes base-class "
43         << "print function:\n\n";
```

Fig. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 5.)

```
44      commissionEmployeePtr->print(); // invokes base-class print
45
46      // aim derived-class pointer at derived-class object and print
47      basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
48      cout << "\n\nCalling virtual function print with derived-class "
49         << "pointer\nto derived-class object invokes derived-class "
50         << "print function:\n\n";
51      basePlusCommissionEmployeePtr->print(); // invokes derived-class print
52
53      // aim base-class pointer at derived-class object and print
54      commissionEmployeePtr = &basePlusCommissionEmployee;
55      cout << "\n\nCalling virtual function print with base-class pointer"
56         << "\nto derived-class object invokes derived-class "
57         << "print function:\n\n";
58
59      // polymorphism; invokes BasePlusCommissionEmployee's print;
60      // base-class pointer to derived-class object
61      commissionEmployeePtr->print();
62      cout << endl;
63   } // end main
```

Fig. 13.10 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 3 of 5.)

```
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:

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
```

13.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- Although a derived-class object also *is a* base-class object, the two objects are nevertheless different.
- Base-class objects cannot be treated as if they were derived-class objects—the derived class can have additional derived-class-only members.
- Aiming a derived-class pointer at a base-class object is not allowed without an explicit cast
- The cast relieves the compiler of the responsibility of issuing an error message.

13.3.6 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers (cont.)

- We've discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:
  - Aiming a base-class pointer at a base-class
  - Aiming a derived-class pointer at a derived-class object
  - Aiming a base-class pointer at a derived-class object
  - Aiming a derived-class pointer at a base-class object

Common Programming Error 13.1

After aiming a base-class pointer at a derived-class object, attempting to reference derived-class-only members with the base-class pointer is a compilation error.

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."

13.5 Abstract Classes and Pure `virtual` Functions

- 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, “Drawing the two-dimensional shape.”
- 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.

13.5 Abstract Classes and Pure `virtual` Functions (cont.)

- A good example of this is the shape, 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
  - ```
    // pure virtual function
    virtual void draw() const = 0;
    ```
- 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.

13.5 Abstract Classes and Pure `virtual` Functions (cont.)

- 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 13.8

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 13.3

Attempting to instantiate an object of an abstract class causes a compilation error.

Software Engineering Observation 13.9

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.*

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.

13.6 Case Study: Payroll System Using Polymorphism

- We create an enhanced employee hierarchy to solve the following problem:
  - A company pays its employees weekly. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, 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.

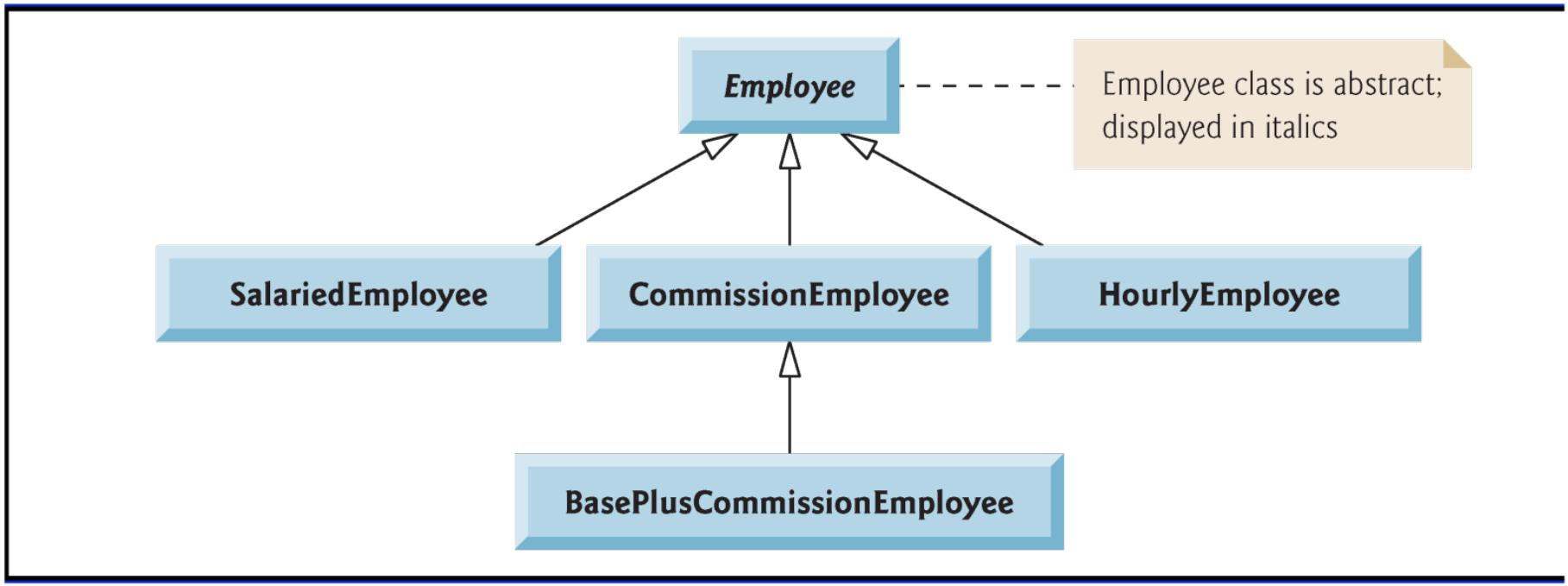


Fig. 13.11 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 13.10

A derived class can inherit interface or implementation from a base class. Hierarchies designed for ***implementation inheritance*** *tend to have their functionality high in the hierarchy—each new derived class inherits one or more member functions that were defined in 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 for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).*

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 13.12 | Polymorphic interface for the `Employee` hierarchy classes.

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
22
```

Fig. 13.13 | `Employee` class header file. (Part 1 of 2.)

```
23      // pure virtual function makes Employee abstract base class
24      virtual double earnings() const = 0; // pure virtual
25      virtual void print() const; // virtual
26   private:
27      string firstName;
28      string lastName;
29      string socialSecurityNumber;
30   }; // end class Employee
31
32   #endif // EMPLOYEE_H
```

Fig. 13.13 | `Employee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

Fig. 13.15 | SalariedEmployee class header file.

```
21   // return salary
22   double SalariedEmployee::getWeeklySalary() const
23   {
24      return weeklySalary;
25   } // end function getWeeklySalary
26
27   // calculate earnings;
28   // override pure virtual function earnings in Employee
29   double SalariedEmployee::earnings() const
30   {
31      return getWeeklySalary();
32   } // end function earnings
33
34   // print SalariedEmployee's information
35   void SalariedEmployee::print() const
36   {
37      cout << "salaried employee: ";
38      Employee::print(); // reuse abstract base-class print function
39      cout << "\nweekly salary: " << getWeeklySalary();
40   } // end function print
```

Fig. 13.16 | `SalariedEmployee` class implementation file. (Part 2 of 2.)

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    static const int hoursPerWeek = 168; // hours in one week
12
13    HourlyEmployee( const string &, const string &,
14       const string &, double = 0.0, double = 0.0 );
15
16    void setWage( double ); // set hourly wage
17    double getWage() const; // return hourly wage
18
19    void setHours( double ); // set hours worked
20    double getHours() const; // return hours worked
21
22    // keyword virtual signals intent to override
23    virtual double earnings() const; // calculate earnings
24    virtual void print() const; // print HourlyEmployee object
```

Fig. 13.17 | HourlyEmployee class header file. (Part 1 of 2.)

```
25   private:
26      double wage; // wage per hour
27      double hours; // hours worked for week
28   }; // end class HourlyEmployee
29
30   #endif // HOURLY_H
```

Fig. 13.17 | `HourlyEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
```

Fig. 13.19 | CommissionEmployee class header file. (Part 1 of 2.)

```
23   private:
24      double grossSales; // gross weekly sales
25      double commissionRate; // commission percentage
26   }; // end class CommissionEmployee
27
28   #endif // COMMISSION_H
```

Fig. 13.19 | `CommissionEmployee` class header file. (Part 2 of 2.)

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from CommissionEmployee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

Fig. 13.21 | BasePlusCommissionEmployee class header file.

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>
7  #include "Employee.h"
8  #include "SalariedEmployee.h"
9  #include "HourlyEmployee.h"
10 #include "CommissionEmployee.h"
11 #include "BasePlusCommissionEmployee.h"
12 using namespace std;
13
14 void virtualViaPointer( const Employee * const ); // prototype
15 void virtualViaReference( const Employee & ); // prototype
16
17 int main()
18 {
19    // set floating-point output formatting
20    cout << fixed << setprecision( 2 );
21
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 1 of 7.)

```
22      // create derived-class objects
23      SalariedEmployee salariedEmployee(
24         "John", "Smith", "111-11-1111", 800 );
25      HourlyEmployee hourlyEmployee(
26         "Karen", "Price", "222-22-2222", 16.75, 40 );
27      CommissionEmployee commissionEmployee(
28         "Sue", "Jones", "333-33-3333", 10000, .06 );
29      BasePlusCommissionEmployee basePlusCommissionEmployee(
30         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
31
32      cout << "Employees processed individually using static binding:\n\n";
33
34      // output each Employee's information and earnings using static binding
35      salariedEmployee.print();
36      cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
37      hourlyEmployee.print();
38      cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
39      commissionEmployee.print();
40      cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
41      basePlusCommissionEmployee.print();
42      cout << "\nearned $" << basePlusCommissionEmployee.earnings()
43         << "\n\n";
44
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 2 of 7.)

```
45      // create vector of four base-class pointers
46      vector < Employee * > employees( 4 );
47
48      // initialize vector with Employees
49      employees[ 0 ] = &salariedEmployee;
50      employees[ 1 ] = &hourlyEmployee;
51      employees[ 2 ] = &commissionEmployee;
52      employees[ 3 ] = &basePlusCommissionEmployee;
53
54      cout << "Employees processed polymorphically via dynamic binding:\n\n";
55
56      // call virtualViaPointer to print each Employee's information
57      // and earnings using dynamic binding
58      cout << "Virtual function calls made off base-class pointers:\n\n";
59
60      for ( size_t i = 0; i < employees.size(); i++ )
61         virtualViaPointer( employees[ i ] );
62
63      // call virtualViaReference to print each Employee's information
64      // and earnings using dynamic binding
65      cout << "Virtual function calls made off base-class references:\n\n";
66
67      for ( size_t i = 0; i < employees.size(); i++ )
68         virtualViaReference( *employees[ i ] ); // note dereferencing
69   } // end main
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 3 of 7.)

```
70
71 // call Employee virtual functions print and earnings off a
72 // base-class pointer using dynamic binding
73 void virtualViaPointer( const Employee * const baseClassPtr )
74 {
75    baseClassPtr->print();
76    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
77 } // end function virtualViaPointer
78
79 // call Employee virtual functions print and earnings off a
80 // base-class reference using dynamic binding
81 void virtualViaReference( const Employee &baseClassRef )
82 {
83    baseClassRef.print();
84    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
85 } // end function virtualViaReference
```

Fig. 13.23 | `Employee` class hierarchy driver program. (Part 4 of 7.)

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 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.
  - You might forget to test all possible cases in a `switch` statement.
  - When adding new types, you might forget to insert the new cases in all relevant `switch` statements.

Software Engineering Observation 13.6

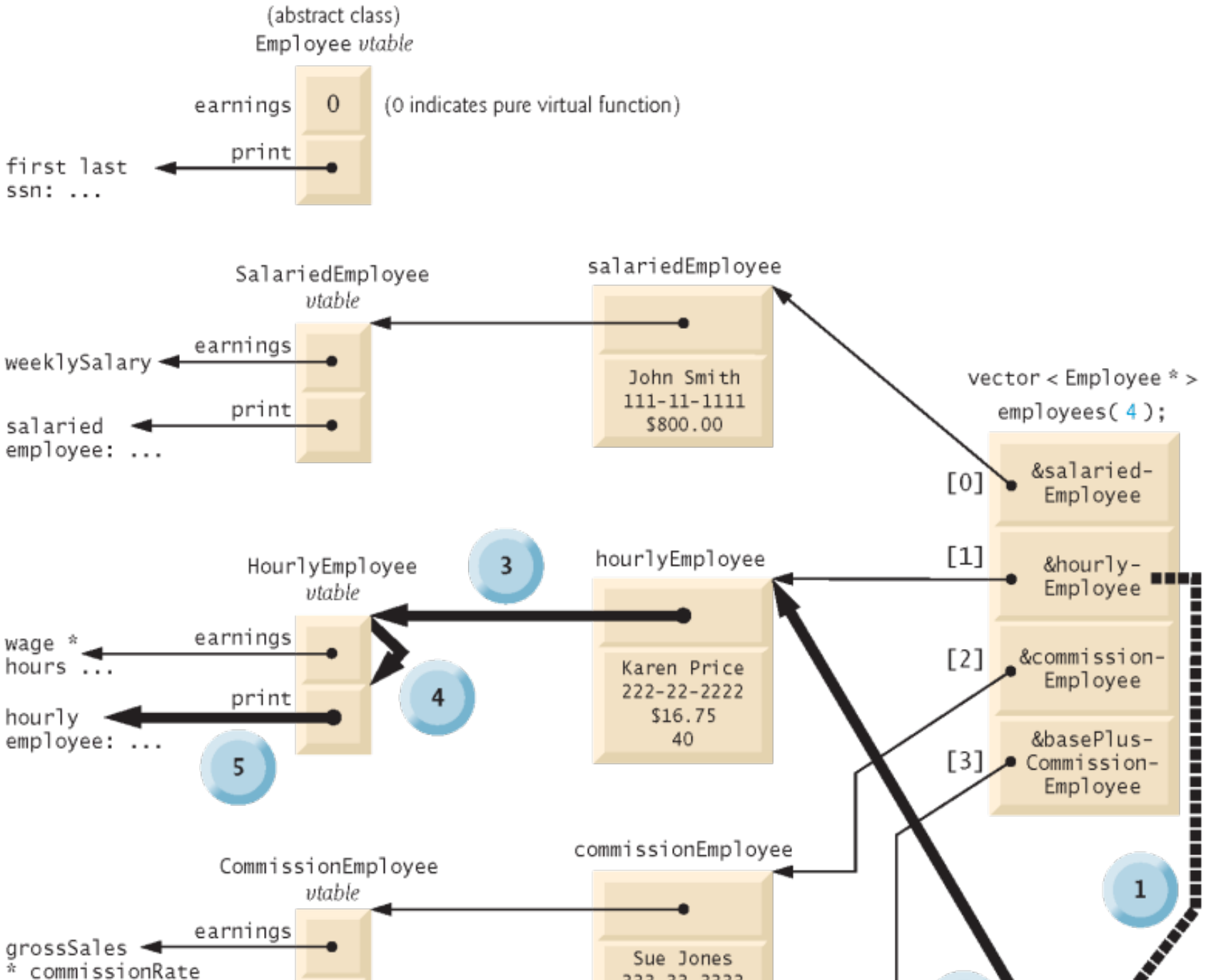
Polymorphic programming can eliminate the need for `switch` *logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with* `switch` *logic.*

Software Engineering Observation 13.7

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.

13.6 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- When C++ compiles a class that has one or more `virtual` functions, it builds a virtual function table (vtable) for that class.
- An executing program uses the *vtable* to select the proper function implementation each time a `virtual` function of that class is called.
- In the vtable for class `Employee`, the first function pointer is set to 0 (i.e., the null pointer) because function `earnings` is a pure `virtual` function and therefore lacks an implementation.


(abstract class)
Employee vtable
earnings
0
(0 indicates pure virtual function)
print
first last
ssn: ...
SalariedEmployee
vtable
earnings
weeklySalary
print
salaried
employee: ...
salariedEmployee
John Smith
111-11-1111
$800.00
vector < Employee * >
employees(4);
[0]
&salaried-
Employee
[1]
&hourly-
Employee
[2]
&commission-
Employee
[3]
&basePlus-
Commission-
Employee
HourlyEmployee
vtable
3
hourlyEmployee
earnings
wage *
hours ...
print
hourly
employee: ...
4
5
Karen Price
222-22-2222
$16.75
40
commissionEmployee
CommissionEmployee
vtable
earnings
grossSales
* commissionRate
Sue Jones
1

13.6 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Polymorphism is accomplished through an elegant data structure involving three levels 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.

Performance Tip 13.1

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

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and runtime type information.
3  // NOTE: You may need to enable RTTI on your compiler
4  // before you can execute this application.
5  #include <iostream>
6  #include <iomanip>
7  #include <vector>
8  #include <typeinfo>
9  #include "Employee.h"
10 #include "SalariedEmployee.h"
11 #include "HourlyEmployee.h"
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14 using namespace std;
15
16 int main()
17 {
18    // set floating-point output formatting
19    cout << fixed << setprecision( 2 );
20
21    // create vector of four base-class pointers
22    vector < Employee * > employees( 4 );
23
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 1 of 4.)

```
24       // initialize vector with various kinds of Employees
25       employees[ 0 ] = new SalariedEmployee(
26          "John", "Smith", "111-11-1111", 800 );
27       employees[ 1 ] = new HourlyEmployee(
28          "Karen", "Price", "222-22-2222", 16.75, 40 );
29       employees[ 2 ] = new CommissionEmployee(
30          "Sue", "Jones", "333-33-3333", 10000, .06 );
31       employees[ 3 ] = new BasePlusCommissionEmployee(
32          "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
33
34       // polymorphically process each element in vector employees
35       for ( size_t i = 0; i < employees.size(); i++ )
36       {
37          employees[ i ]->print(); // output employee information
38          cout << endl;
39
40          // downcast pointer
41          BasePlusCommissionEmployee *derivedPtr =
42             dynamic_cast < BasePlusCommissionEmployee * >
43                ( employees[ i ] );
44
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 2 of 4.)

```
45          // determine whether element points to base-salaried
46          // commission employee
47          if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
48          {
49             double oldBaseSalary = derivedPtr->getBaseSalary();
50             cout << "old base salary: $" << oldBaseSalary << endl;
51             derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
52             cout << "new base salary with 10% increase is: $"
53                << derivedPtr->getBaseSalary() << endl;
54          } // end if
55
56          cout << "earned $" << employees[ i ]->earnings() << "\n\n";
57       } // end for
58
59       // release objects pointed to by vector's elements
60       for ( size_t j = 0; j < employees.size(); j++ )
61       {
62          // output class name
63          cout << "deleting object of "
64             << typeid( *employees[ j ] ).name() << endl;
65
66          delete employees[ j ];
67       } // end for
68    } // end main
```

Fig. 13.25 | 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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

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 HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Fig. 13.25 | Demonstrating downcasting and runtime type information. (Part 4 of 4.)

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

- Operator `typeid` 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` returns a pointer-based string that contains the type name (e.g., "`class BasePlusCommissionEmployee`") of the argument passed to `typeid`.
- To use `typeid`, the program must include header file `<typeinfo>`.

Portability Tip 13.1

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

13.9 Virtual Destructors

- If a derived-class object with a nonvirtual destructor is destroyed explicitly 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 `virtual` destructor (i.e., a destructor that is declared with keyword `virtual`) in the base class.

Error-Prevention Tip 13.2

If a class has `virtual` *functions, 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 13.5

Constructors cannot be `virtual`*. Declaring a constructor* `virtual` *is a compilation error.*